Schema-Agnostic Query Rewriting in SPARQL 1.1

Stefan Bischof[1], Markus Krötzsch[2], Axel Polleres[3], and Sebastian Rudolph[2]

[1] Vienna University of Technology, Austria and Siemens AG Österreich, Austria
[2] Technische Universität Dresden, Germany
[3] Vienna University of Economics and Business, Austria

Abstract. SPARQL 1.1 supports the use of ontologies to enrich query results with logical entailments, and OWL 2 provides a dedicated fragment OWL QL for this purpose. Typical implementations use the OWL QL schema to rewrite a conjunctive query into an equivalent set of queries, to be answered against the non-schema part of the data. With the adoption of the recent SPARQL 1.1 standard, however, RDF databases are capable of answering much more expressive queries directly, and we ask how this can be exploited in query rewriting. We find that SPARQL 1.1 is powerful enough to "implement" a full-fledged OWL QL reasoner in a single query. Using additional SPARQL 1.1 features, we develop a new method of schema-agnostic query rewriting, where arbitrary conjunctive queries over OWL QL are rewritten into equivalent SPARQL 1.1 queries in a way that is fully independent of the actual schema. This allows us to query RDF data under OWL QL entailment without extracting or preprocessing OWL axioms.

1 Introduction

SPARQL 1.1, the recent revision of the W3C SPARQL standard, introduces significant extensions to the capabilities of the popular RDF query language [10]. Even at the very core of the query language, we can find many notable new features, including *property paths*, *value creation* (BIND), inline data (VALUES), negation, and extended filtering capabilities. In addition, SPARQL 1.1 now supports query answering over OWL ontologies, taking full advantage of ontological information in the data [8].

Query answering in the presence of ontologies is known as *ontology-based data access* (OBDA), and has long been an important topic in applied and foundational research. Even before SPARQL provided support for this feature, several projects have used ontologies to integrate disparate data sources or to provide views over legacy databases, e.g. [5,15,16,6,11]. The W3C OWL 2 Web Ontology Language includes the OWL QL language profile, which was specifically designed for this application [12]. With the arrival of SPARQL 1.1, every aspect of OBDA is thus supported by tailor-made W3C technologies.

In practice, however, SPARQL and OWL QL are rarely integrated. Most works on OBDA address the problem of answering *conjunctive queries* (CQs), which correspond to SELECT-PROJECT-JOIN queries in SQL, and (to some degree) to Basic Graph Patterns in SPARQL. The most common approach for OBDA is *query rewriting*, where a given CQ is rewritten into a (set of) CQs that fully incorporate the schema information of the ontology. The answers to the rewritten queries (obtained without considering the

P. Mika et al. (Eds.) ISWC 2014, Part I, LNCS 8796, pp. 584–600, 2014.

ontology) are guaranteed to agree with the answers of the original queries (over the ontology). This approach separates the ontology (used for query rewriting) from the rest of the data (used for query answering), and it is typical that the latter is stored in a relational database. Correspondingly, the rewritten queries are often transformed into SQL for query answering. SPARQL and RDF do not play a role in this.

In this paper, we thus take a fresh look on the problem of OBDA query rewriting with SPARQL 1.1 as our target query language. The additional expressive power of SPARQL 1.1 allows us to introduce a new paradigm of *schema-agnostic query rewriting*, where the ontological schema is not needed for rewriting queries. Rather, the ontology is stored *together with the data* in a single RDF database. This is how many ontologies are managed today, and it corresponds to the W3C view on OWL and RDF, which does not distinguish schema and data components. The fact that today's OBDA approaches separate both parts testifies to their focus on relational databases. Our work, somewhat ironically, widens the scope of OWL QL to RDF-based applications, which have hitherto focused on OWL RL as their ontology language of choice.

Another practical advantage of schema-agnostic query rewriting is that it supports frequent updates of both data and schema. The rewriting system does not need any information on the content of the database under query, while the SPARQL processor that executes the query does not need any support for OWL. This is particularly interesting if a database can only be accessed through a restricted SPARQL query interface that does not support reasoning. For example, we have used our approach to check the consistency of DBpedia under OWL semantics, using only the public Live DBpedia SPARQL endpoint[1] (it is inconsistent: every library is inferred to belong to the mutually disjoint classes "Place" and "Agent").

Our main contributions are as follows:

- We express the standard reasoning tasks for OWL QL, including consistency checking, classification, and instance retrieval, in *single, fixed* SPARQL 1.1 queries that are independent of the ontology. For this, we use SPARQL 1.1 property paths, which support a simple form of recursion that is powerful enough for OWL QL reasoning.
- We show how to rewrite arbitrary SPARQL Basic Graph Patterns (BGPs) into single SPARQL 1.1 queries of polynomial size. This task is simplified by the fact that SPARQL does not support "non-distinguished" variables as used in general CQs.
- We present a schema-agnostic rewriting of general CQs in SPARQL 1.1, again into single queries of polynomial size. This rewriting is more involved, and we use two additional features: inline data (VALUES) and (in)equality checks in filters.
- We show the limits of schema-agnostic rewriting in SPARQL 1.1 by proving that many other OWL features cannot be supported in this way. This includes even the most basic features OWL EL and OWL RL, and mild extensions of OWL QL.

Worst-case reasoning complexity remains the same in all cases, yet our approach is much more practical in the case of standard reasoning and BGP rewriting. For general CQs, the rewritten queries are usually too complex for today's RDF databases to handle. Nevertheless, we think that our "SPARQL 1.1 implementation" of OWL QL query answering is a valuable contribution, since it reduces the problem of supporting OWL QL

[1] http://live.dbpedia.org/sparql

in an RDF database to the task of optimizing a single (type of) query. Since OWL QL subsumes RDFS, one can also apply our insights to implement query answering under RDFS ontologies, which again leads to much simpler queries.

In Section 2, we start by giving a compact introduction to the parts of SPARQL 1.1 that we require. Thereafter, in Section 3, we introduce OWL QL and relate its semantics to a *chase* procedure. In Section 4, we develop queries for implementing basic QL reasoning in SPARQL 1.1, and in Section 5, we extend this into a schema-agnostic query rewriting procedure for conjunctive queries. Finally, we investigate the limits of schema-agnostic query rewriting, and present several negative results in Section 6. We close with a short discussion and outlook in Section 7. Omitted proofs can be found in the accompanying technical report [3].

2 Preliminaries: RDF and SPARQL 1.1

We consider RDF documents based on the set **IRI** of IRIs and **BN** of blank node identifiers; we do not consider literals, since they would complicate our exposition without adding technical insights (they can mostly be treated like named individuals in OWL QL). We use Turtle syntax for denoting RDF throughout this paper.

In addition to IRIs and blank nodes, SPARQL 1.1 queries use variables as constituents, which are indicated by a preceding question mark. For compatibility with the entailment regimes, we will consider SPARQL 1.1 under the set semantics, i.e., multiplicities of solutions will be ignored, as indicated by the DISTINCT keyword. Next, we introduce syntax and semantics of the SPARQL 1.1 fragment employed in this paper.

Path expressions are defined inductively as follows: (i) Every IRI is a property path. (ii) For p and q property paths, the following expressions are property paths as well: $(\hat{}p)$ for inverse, $(p \,/\, q)$ for sequence, $(p \mid q)$ for alternative, (p^*) for Kleene star. As usual, parentheses can be omitted if there is no danger of confusion. *Triple expressions* are of the form $s\ p\ o$ where s and o are IRIs, blank nodes, or variables, whereas p is an IRI, a variable, or a path expression. *Basic graph patterns* are defined as finite sequences of triple expressions separated by a period. *Values blocks* for inline data have the shape VALUES $(?x_1 \ldots ?x_n)\{(v_{1,1} \ldots v_{1,n}) \ldots (v_{k,1} \ldots v_{k,n})\}$ for natural numbers n and k with $v_{i,j} \in$ **IRI** $\cup$ **BN**. *Filter expressions* are of the form FILTER(*boolexp*) where *boolexp* is an algebraic expression encoding the application of filter functions to variables resulting in a Boolean value (for more details see [10]). *Graph patterns* are defined inductively: (i) any basic graph pattern is a graph pattern (ii) if gp_1 and gp_2 are graph patterns then $\{gp_1\}$ UNION $\{gp_2\}$ is a graph pattern (iii) any sequence of graph patterns, values blocks and filter expressions is again a graph pattern. A SELECT-DISTINCT *query* is a SPARQL 1.1 query of the shape SELECT DISTINCT *varlist* WHERE $\{gp\}$, where gp is a graph pattern and *varlist* is a list of variables occurring in gp.

We now define the sematics of SPARQL 1.1 queries, without taking reasoning into account; this is known as *simple entailment* (as opposed to OWL DL entailment, where the OWL axioms are evaluated under OWL Direct Semantics [8]). We define the *evaluation* of path expressions w.r.t. G as a binary relation over **IRI** $\cup$ **BN** in an inductive way: $eval_G(p) = \{(u_1, u_2) \mid u_1\ p\ u_2 \in G)\}$ for $p \in$ **IRI**, inverse $eval_G(\hat{}p) = \{(u_2, u_1) \mid (u_1, u_2) \in eval_G(p)\}$, sequence $eval_G(p \,/\, q) = \{(u_1, u_3) \mid$

$(u_1, u_2) \in eval_G(p), (u_2, u_3) \in eval_G(q)\}$, alternative $eval_G(p \mid q) = eval_G(p) \cup eval_G(q)$, Kleene star $eval_G(p^*) = \bigcup_{n \geq 0} eval_G(p^n)$ where $eval_G(p^0) = \{(u, u) \mid u \in \textbf{IRI} \cup \textbf{BN}$ occurs in $G\}$ and $eval_G(p^{n+1}) = eval_G(p^n) \circ eval_G(p)$. The *evaluation* $eval_G(bgp)$ of a basic graph pattern *bgp* w.r.t. some RDF graph G is the set of all partial mappings μ from variables in *bgp* to IRIs or blank nodes of G, such that there exists some mapping σ from all blank nodes in *bgp* to terms of G for which $\mu(\sigma(bgp)) \in G$. Moreover, $eval_G(\mathsf{VALUES}\ (?x_1 \ldots ?x_n)\{(v_{1,1} \ldots v_{1,n}) \ldots (v_{k,1} \ldots v_{k,n})\}) = \{\{?x_1 \mapsto v_{1,1}, \ldots, ?x_n \mapsto v_{1,n}\}, \ldots \{?x_1 \mapsto v_{k,1}, \ldots, ?x_n \mapsto v_{k,n}\}\}$ and $eval_G(\{gp_1\}\ \mathsf{UNION}\ \{gp_2\}) = eval_G(gp_1) \cup eval_G(gp_2)$. For graph patterns *gp* that are sequences of graph patterns, values blocks and filter expressions $\mathsf{FILTER}(boolexp_1), \ldots, \mathsf{FILTER}(boolexp_\ell)$ we let $eval_G(gp) = \{\mu \mid \mu \in J \wedge \mu(boolexp_1) = true \wedge \ldots \wedge \mu(boolexp_\ell) = true\}$ where J is the join over all $eval_G(block)$ where *block* ranges over all graph patterns and values blocks of the sequence. We say a graph pattern *gp* has a *match* into a graph G if $eval_G(gp) \neq \emptyset$. Finally, the set of *answers* of a SELECT-DISTINCT query SELECT DISTINCT *varlist* WHERE $\{gp\}$ is the set obtained by restricting every partial function $\mu \in eval_G(gp)$ to the variables contained in *varlist*.

3 OWL QL: RDF Syntax and Rule-Based Semantics

OWL QL is one of the OWL 2 profiles, which restrict the OWL 2 DL ontology language to ensure that reasoning is tractable [12]. To ensure compatibility with SPARQL, we work only with the RDF representation of OWL QL here [13]. Like OWL 2 DL, OWL QL requires "standard use" of RDFS and OWL vocabulary, i.e., special vocabulary that is used to encode ontology axioms in RDF is strictly distinct from the ontology's vocabulary, and can only occur in specific triple patterns. Only a few special IRIs, such as owl:Thing, can also be used like ontology vocabulary in axioms.

OWL classes, properties, and individuals are represented by RDF elements, where complex class and property expressions are represented by blank nodes. Whether an expression is represented by an IRI or a blank node does not have an impact on ontological entailment, so we ignore this distinction in most cases. OWL 2 DL allows us to use a single IRI to represent an individual, a class, and a property in the same ontology; owing to the restrictions of standard use, it is always clear which meaning applies in a particular case. Hence we will also work with one single set of IRIs.

Next, we define the constrains that an RDF graph has to satisfy to represent an OWL QL ontology. To this end, consider a fixed RDF graph G. A *property expression in G* is an IRI or a blank node _:b that occurs in a pattern {_:b owl:inverseOf P} with $P \in \textbf{IRI}$. We use **PRP** for the set of all property elements in a given RDF graph. OWL QL further distinguishes two types of class expressions with different syntactic constraints. The set **SBC** of *subclasses in G* consists of all IRIs and all blank nodes _:b that occur in a pattern {_:b owl:onProperty P; owl:someValuesFrom owl:Thing}, where $P \in \textbf{PRP}$. The set **SPC** of *superclasses in G* is defined recursively as follows. An element x is in **SPC** if it is in **IRI**, or if it is in **BN** and G contains one of the following patterns:

- $\{x$ owl:onProperty **PRP**; owl:someValuesFrom $y\}$ where $y \in \textbf{SPC}$;
- $\{x$ owl:intersectionOf $(y_1, \ldots, y_n)\}$ where $y_1, \ldots, y_n \in \textbf{SPC}$;
- $\{x$ owl:complementOf $y\}$ where $y \in \textbf{SBC}$.

G is an *OWL QL ontology* may use the following triple patterns to encode *axioms*:

- {**IRI PRP IRI**}
- {**IRI** rdf:type **SPC**}
- {**SBC** rdfs:subClassOf **SPC**}
- {**SBC** owl:equivalentClass **SBC**}
- {**SBC** owl:disjointWith **SBC**}
- {**PRP** rdfs:range **SPC**}
- {**PRP** rdfs:domain **SPC**}
- {**PRP** rdfs:subPropertyOf **PRP**}
- {**PRP** owl:equivalentProperty **PRP**}
- {**PRP** owl:inverseOf **PRP**}
- {**PRP** owl:propertyDisjointWith **PRP**}
- {**IRI** owl:differentFrom **IRI**}

- {**BN** rdf:type owl:AllDisjointClasses; owl:members (**SBC**, . . . , **SBC**)}
- {**BN** rdf:type owl:AllDisjointProperties; owl:members (**PRP**, . . . , **PRP**)}
- {**BN** rdf:type owl:AllDifferent; owl:members (**IRI**, . . . , **IRI**)}

G is an OWL QL ontology if every triple in G is part of a unique axiom or a unique complex class or property definition used in such axioms. For simplicity, we ignore triples used in annotations or ontology headers. Moreover, we do not consider the OWL QL property characteristics symmetry, asymmetry, and global reflexivity. Asymmetry and reflexivity are not a problem, but their explicit treatment would inflate our presentation considerably. Symmetry, in contrast, cannot be supported with SPARQL 1.1, as we will show in Section 6. This is no major limitation of our approach, since symmetry can be expressed using inverses. This shows that rewritability of an ontology language does not depend on ontological expressiveness alone.

The semantics of OWL QL is inherited from OWL DL, but it can also be described by defining a *universal model*, i.e., a structure that realizes precisely the entailments of an ontology. Such a "least model" exactly captures the semantics of an ontology. To define a universal model for OWL QL, we define a set of RDF-based inference rules, similar to the rules given for OWL RL in the standard [12]. In contrast to OWL RL, however, the application of rules can introduce new elements to an RDF graph, and the universal model that is obtained in the limit is not finite in general. Indeed, our goal is not to give a practical reasoning algorithm, but to define the semantics of OWL QL in a way that is useful for analyzing the correctness of the rewriting algorithms we introduce.

The main rules for reasoning in OWL QL are defined in Table 1. A rule is *applicable* if the premise on the left matches the current RDF graph and the conclusion on the right does not match the current graph; in this case, the conclusion is added to the graph. In case of rule (2), this requires us to create a fresh blank node. In all other cases, we only add new triples among existing elements. Rules like (3) are actually schemas for an infinite number of rules for lists of any length n and any index $i \in \{1, \ldots, n\}$. Rules (15)–(16) cover owl:Thing and owl:topObjectProperty, which lead to conclusions that are true for "all" individuals. To ensure standard use, we cannot simply assert x rdf:type owl:Thing for *every* IRI x, and we restrict instead to IRIs that are used as individuals in the ontology. We define INDIVIDUAL(x) to be the SPARQL pattern {x rdf:type owl:NamedIndividual} UNION {x rdf:type ?C . ?C rdf:type owl:Class} UNION {x ?P ?Y . ?P rdf:type owl:ObjectProperty} UNION {?Y ?P x . ?P rdf:type owl:ObjectProperty}. Note that this also covers any newly introduced individuals.

Table 1. RDF inference rules for OWL QL

→ [] rdf:type owl:Thing	(1)
?X rdf:type [owl:onProperty ?P; owl:someValuesFrom ?C] → ?X ?P [rdf:type ?C]	(2)
?X rdf:type [owl:intersectionOf (?C1, . . . , ?Ci, . . . , ?Cn)] → ?X rdf:type ?Ci	(3)
?X rdf:type ?C . ?C rdfs:subClassOf ?D → ?X rdf:type ?D	(4)
?X rdf:type ?C . ?C owl:equivalentClass ?D → ?X rdf:type ?D	(5)
?X rdf:type ?C . ?D owl:equivalentClass ?C → ?X rdf:type ?D	(6)
?X ?P ?Y . ?C owl:onProperty ?P; owl:someValuesFrom owl:Thing → ?X rdf:type ?C	(7)
?X ?P ?Y . ?P rdfs:domain ?C → ?X rdf:type ?C	(8)
?X ?P ?Y . ?P rdfs:range ?C → ?Y rdf:type ?C	(9)
?X ?P ?Y . ?P owl:inverseOf ?Q → ?Y ?Q ?X	(10)
?X ?P ?Y . ?Q owl:inverseOf ?P → ?Y ?Q ?X	(11)
?X ?P ?Y . ?P rdfs:subPropertyOf ?Q → ?X ?Q ?Y	(12)
?X ?P ?Y . ?P owl:equivalentProperty ?Q → ?X ?Q ?Y	(13)
?X ?P ?Y . ?Q owl:equivalentProperty ?P → ?X ?Q ?Y	(14)
INDIVIDUAL(?X) → ?X rdf:type owl:Thing	(15)
?X rdf:type owl:Thing . ?Y rdf:type owl:Thing → ?X owl:topObjectProperty ?Y	(16)

Definition 1. *The* chase *G' of an OWL QL ontology G is a possibly infinite RDF graph obtained from G by fair application of the rules of Tables 1, meaning that every rule that is applicable has eventually been applied.*

Finally, some features of OWL QL can only make the ontology inconsistent, but not introduce any other kinds of positive entailments. According patterns are shown in Table 2. If any of these match, the ontology is inconsistent, every OWL axiom is a logical consequence, and there is no universal model.

Theorem 1. *Consider an OWL QL ontology G with chase G', and a basic graph pattern P. A variable mapping μ is a solution for P over G under the OWL DL entailment regime if and only if either (1) μ is a solution for P over G' under simple entailment, or (2) one of the patterns of Table 2 matches G'.*

4 QL Reasoning with SPARQL Property Expressions

Next, we define SPARQL 1.1 queries to solve standard reasoning tasks of OWL QL. We start with simple cases and then consider increasingly complex reasoning problems.

We first focus on the property hierarchy. An axiom of the form p rdfs:subPropertyOf q is entailed by an ontology G if, for newly introduced individuals a and b, $G \cup \{a\ p\ b\}$ entails $\{a\ q\ b\}$. By Theorem 1, the rules of Section 3 represent all possibilities for deriving this information. In this particular case, we can see that only rules (10)–(14) in Table 1 can derive a triple of the form $a\ q\ b$, where q is a regular property. The case q =owl:topObjectProperty is easy to handle, since p rdfs:subPropertyOf

Table 2. RDF inference patterns for inconsistency in OWL QL

?X owl:bottomObjectProperty ?Y	(17)
?X rdf:type owl:Nothing	(18)
?X rdf:type ?C . ?X rdf:type [owl:complementOf ?C]	(19)
?X rdf:type ?C . ?X rdf:type ?D . ?C owl:disjointWith ?D	(20)
?X rdf:type ?Ci . ?X rdf:type ?Cj.	
_:b rdf:type owl:AllDisjointClasses; owl:members (?C1, . . . , ?Ci, . . . , ?Cj, . . . , ?Cn)	(21)
?X ?P ?Y . ?X ?Q ?Y . ?P owl:propertyDisjointWith ?Q	(22)
?X ?Pi ?Y . ?X ?Pj ?Y.	
_:b rdf:type owl:AllDisjointProperties; owl:members (?P1, . . . , ?Pi, . . . , ?Pj, . . . , ?Pn)	(23)
?X owl:differentFrom ?X	(24)
_:b rdf:type owl:AllDifferent; owl:members (?I1, . . . , ?X, . . . , ?X, . . . , ?In)	(25)

owl:topObjectProperty is always true (which is also shown by rules (15) and (16)). In addition, it might be that $G \cup \{a\ p\ b\}$ is inconsistent, implied by rules of Table 2; we will ignore this case for now, since it requires more powerful reasoning.

Definition 2. *We introduce* sPO, invOf, *and* eqP *as abbreviations for* rdfs:subPropertyOf, owl:inverseOf, *and* owl:equivalentProperty, *respectively, and define the following composite property path expressions* SPOEQP := (sPO | eqP | ^eqP), INV := (invOf | ^invOf), SUBPROPERTYOF := (SPOEQP | (INV / SPOEQP* / INV))*, *as well as* SUBINVPROPERTYOF := SPOEQP* / INV / SUBPROPERTYOF. *Moreover, for an arbitrary term x, let* UNIVPROPERTY[x] *be the pattern* {owl:topObjectProperty (SPOEQP | INV)* x}.

The pattern SUBPROPERTYOF does not check for property subsumption that is caused by the inconsistency rules in Table 2, but it can be used to check for subsumptions related to owl:topObjectProperty. The corresponding correctness result is as follows:

Proposition 1. *Consider an OWL QL ontology G with properties* $p, q \in$ **PRP** *such that* $G \cup \{$_:a p _:b$\}$ *is consistent. Then G entails p* rdfs:subPropertyOf *q iff the pattern* {p SUBPROPERTYOF q} UNION UNIVPROPERTY[q] *matches G.*

We will extend this to cover the inconsistent case in Theorem 2 below. First, however, we look at entailments of class subsumptions. In this case, the main rules are (2)–(9). However, several of these rules also depend on property triples derived by rules (10)–(14), and we apply our results on property subsumption to take this into account.

Definition 3. *Let* eqC *and* sCO *abbreviate* owl:equivalentClass *and* rdfs:subClassOf, *respectively. We define property path expressions*

- INTLISTMEMBER := (owl:intersectionOf / rdf:rest* / rdf:first),
- SOMEPROP := (owl:onProperty / SUBPROPERTYOF / (^owl:onProperty | rdfs:domain)),
- SOMEPROPINV := (owl:onProperty / SUBINVPROPERTYOF / rdfs:range),
- SUBCLASSOF := (sCO | eqC | ^eqC | INTLISTMEMBER | SOMEPROP | SOMEPROPINV)*.

Table 3. Pattern EMPTYCLASS[x] for detecting empty classes

```
x (sCO | eqC | ^eqC | INTLISTMEMBER | owl:someValuesFrom |
    (owl:onProperty / (INV | SPOEQP)* / (^owl:onProperty | rdfs:domain | rdfs:range))* ?C . {
   {?C SUBCLASSOF owl:Nothing} UNION
   {?C SUBCLASSOF ?D1 {{?C SUBCLASSOF ?D2} UNION UNIVCLASS[?D2]} {
      {?D1 DISJOINTCLASSES ?D2} UNION
      {?V rdf:type owl:AllDisjointClasses . TWOMEMBERS[?V, ?D1, ?D2]}
   }} UNION
   {?C (owl:onProperty / (INV | SPOEQP)*) ?P . {
      {?P SUBPROPERTYOF owl:bottomObjectProperty} UNION
      {?P SUBPROPERTYOF ?Q1 {{?P SUBPROPERTYOF ?Q2} UNION UNIVPROPERTY[?Q2]} {
         {?Q1 (owl:propertyDisjointWith | ^owl:propertyDisjointWith) ?Q2} UNION
         {?V rdf:type owl:AllDisjointProperties . TWOMEMBERS[?V, ?Q1, ?Q2]}
      }
   }}
}
```

Moreover, we let UNIVCLASS[x] *denote the pattern* {owl:Thing SUBCLASSOF x} UNION {owl:topObjectProperty ((SPOEQP | INV)* / (^owl:onProperty | rdfs:domain | rdfs:range)/ SUBCLASSOF) x}

Proposition 2. *Consider an OWL QL ontology G with classes $c \in$* **SPC** *and $d \in$* **SBC** *such that $G \cup$* {_:a rdf:type c} *is consistent. Then G entails c* rdfs:subClassOf *d iff the pattern* {c SUBCLASSOF d} UNION UNIVCLASS[d] *matches G.*

It remains to identify classes that are incoherent, i.e., for which c rdfs:subClassOf owl:Nothing is entailed. To do this, we need to consider the patterns of Table 2.

Definition 4. *For arbitrary terms x, y, and z, let* TWOMEMBERS[x, y, z] *be the pattern* {x (owl:members / rdf:rest*) ?W . ?W rdf:first y . ?W (rdf:rest+ / rdf:first) z}*, and let* DISJOINTCLASSES *be the property path expression* (owl:disjointWith | ^owl:disjointWith | owl:complementOf | ^owl:complementOf)*. The query pattern* EMPTYCLASS[x] *is defined as in Table 3, and the query pattern* EMPTYPROPERTY[x] *is defined as in Table 4.*

We can now completely express OWL QL schema reasoning in SPARQL 1.1:

Theorem 2. *An OWL QL ontology G is inconsistent iff it has a match for the pattern*

```
{?X rdf:type ?C . EMPTYCLASS[?C]} UNION {?X ?P ?Y . EMPTYPROPERTY[?P]} UNION
{?X owl:differentFrom ?X} UNION
{?V rdf:type owl:AllDifferent . TWOMEMBERS[?V, ?X, ?X]}.                    (26)
```

G entails c rdfs:subClassOf *d for $c \in$* **SPC** *and $d \in$* **SBC** *iff G is either inconsistent or has a match for the pattern*

$$\{c \text{ SUBCLASSOF } d\} \text{ UNION } \text{UNIVCLASS}[d] \text{ UNION } \text{EMPTYCLASS}[c]. \tag{27}$$

Table 4. Pattern EMPTYPROPERTY[x] for detecting empty properties

```
x (INV | SPOEQP | (^owl:onProperty /
    (sCO | eqC | ^eqC | INTLISTMEMBER | owl:someValuesFrom)* / owl:onProperty))* ?P . {
  {?P SUBPROPERTYOF owl:bottomObjectProperty} UNION
  {?P SUBPROPERTYOF ?Q1 {{?P SUBPROPERTYOF ?Q2} UNION UNIVPROPERTY[?Q2]} {
     {?Q1 (owl:propertyDisjointWith | ^owl:propertyDisjointWith) ?Q2} UNION
     {?V rdf:type owl:AllDisjointProperties . TWOMEMBERS[?V, ?Q1, ?Q2]}
  }} UNION
  {?P ((^owl:onProperty | rdfs:domain | rdfs:range) / SUBCLASSOF) ?C . {
     {?C SUBCLASSOF owl:Nothing} UNION
     {?C SUBCLASSOF ?D1 {{?C SUBCLASSOF ?D2} UNION UNIVCLASS[?D2]} {
        {?D1 DISJOINTCLASSES ?D2} UNION
        {?V rdf:type owl:AllDisjointClasses . TWOMEMBERS[?V, ?D1, ?D2]}
     }
  }}
}
```

G entails x rdf:type *c iff G is either inconsistent or has a match for the pattern*

```
{{x (rdf:type / SUBCLASSOF) c} UNION
 {x ?P ?Y . ?P (SUBPROPERTYOF / (^owl:onProperty | rdfs:domain) / SUBCLASSOF) c} UNION
 {?Y ?P x . ?P (SUBPROPERTYOF / rdfs:range / SUBCLASSOF) c}
} UNION UNIVCLASS[c]                                                    (28)
```

G entails p rdfs:subPropertyOf *q for* $p, q \in$ **PRP** *iff G is either inconsistent or has a match for the pattern*

```
{p SUBPROPERTYOF q} UNION UNIVPROPERTY[q] UNION EMPTYPROPERTY[p].       (29)
```

G entails x p y iff G is either inconsistent or has a match for the pattern

```
{x ?R y . ?R SUBPROPERTYOF p} UNION {y ?R x . ?R SUBINVPROPERTYOF p}
                                  UNION UNIVPROPERTY[p].                (30)
```

5 OWL QL Query Rewriting with SPARQL 1.1

We now turn towards query answering over OWL QL ontologies using SPARQL 1.1. Research in OWL QL query answering typically considers the problem of answering *conjunctive queries* (CQs), which are conjunctions of OWL property and class assertions that use variables only in the place of individuals, not in the place of properties or classes. Conjunction can easily be represented by a Basic Graph Pattern in SPARQL, yet CQs are not a subset of SPARQL, since they also support existential quantification of variables. Normal query variables are called *distinguished* while existentially quantified variables are called *non-distinguished*. Distinguished variables can only bind to elements of the ontology, whereas for non-distinguished variables it suffices if the ontology implies that some binding must exist.

Example 1. Consider an OWL ontology with the assertion :peter rdf:type :Person and the axiom :Person rdfs:subClassOf [owl:onProperty :father; owl:someValuesFrom :Person]. This implies that :peter has some :father but the ontology may not contain any element of which we know that it plays this role. In this case, the SPARQL pattern {?X :father ?Y} would not have a match with ?X = :peter under OWL DL entailment. In contrast, if the variable ?Y were non-distinguished, the query would match with ?X = :peter (and ?Y would not receive any binding).

SPARQL can only express CQs where all variables are distinguished. To define this fragment of SPARQL, recall that the OWL DL entailment regime of SPARQL 1.1 requires every variable to be *declared* for a certain type (individual, object property, datatype property, or class) [8]. This requirement is the analogue of "standard use" on the level of query patterns, and it allows us to focus on instance retrieval here. We thus call a Basic Graph Pattern *P CQ-pattern* if: (1) *P* does not contain any OWL, RDF, or RDFS URIs other than rdf:type in property positions, (2) all variables in *P* are declared as required by the OWL DL entailment regime, (3) property variables occur only in predicate positions, and (4) class variables occur only in object positions of triples with predicate rdf:type. Rewriting CQ-patterns is an easy application of Theorem 2:

Definition 5. *For a triple pattern e* rdf:type *c, the rewriting* $[\![x$ rdf:type $c]\!]$ *is the graph pattern* (28) *as in Theorem 2; for a triple pattern x p y, the rewriting* $[\![x\ p\ y]\!]$ *is the graph pattern* (30)*. The rewriting* $[\![P]\!]$ *of a CQ-pattern P is obtained by replacing every triple pattern s p o in P by* $\{[\![s\ p\ o]\!]\}$.

Theorem 3. *If G is the RDF graph of a consistent OWL QL ontology, then the matches of a CQ-pattern P on G under OWL DL entailment are exactly the matches of* $[\![P]\!]$ *on G under simple entailment.*

5.1 Rewriting General Conjunctive Queries

We now explain the additional aspects that we need to take into account for computing answers to CQs with non-distinguished variables, and give an intuitive overview of our rewriting approach. A general challenge that we have to address is that classical query rewriting for OWL QL may lead to exponentially many queries, owing to the fact that many non-deterministic choices have to be made to find a query match. Some of these choices depend on the ontology, e.g., on the depth of the class hierarchy, and are naturally represented in (small) SPARQL 1.1 queries in our approach. Other choices, however, depend on the query, e.g., the decision which variables should be identified (query factorization). It is not immediately clear how to represent these choices in a polynomial query, even when using path expressions. Our solution depends on the creative use of the VALUES feature of SPARQL 1.1.

As explained before, non-distinguished variables can be matched to inferred individuals that are not named in the ontology. The chase introduced in Section 3 still captures this more general notion of query answering. The only rule to infer new individuals is (2), which introduces fresh bnodes that we call *anonymous individuals*. The elements of the original ontology (bnode or not) are *named individuals*. It is well known that a

QL ontology G entails a CQ q if and only if there is a match from q to the (possibly infinite) chase of G such that all distinguished variables are mapped to named individuals. Non-distinguished variables can be mapped to either named or anonymous individuals.

To represent the match of a query variable x in the rewritten query, we introduce a SPARQL variable ?Mx. For named individuals, ?Mx can bind to the individual in the RDF graph. However, if x is non-distinguished, then it could match to anonymous individuals, which are not represented by any individual in RDF. In this case, we bind ?Mx to the bnode _:b representing the OWL property restriction _:b owl:onProperty ?P; owl:someValuesFrom ?C that was used in rule (2) to generate the anonymous individual. Indeed, all class and property assertions that are derived for the anonymous individual can be deduced from ?P and ?C only, so this binding allows us to check query conditions.

However, the bnode _:b does not determine the identity of the anonymous individual, since infinitely many anonymous individuals can be generated from the same OWL property restriction. Example 1 illustrates this: every person has another person as is its father, *ad infinitum*. Nevertheless, the query :peter :father ?Z . ?Z :father ?Z should not have a match, even if ?Z is non-distinguished. Disregarding universal property assertions that follow from rule (16), anonymous individuals can only be related to their parent individual (represented by ?X in rule (2)) or to their children (which have the anonymous element as their parent). Therefore, to check if a triple pattern ?X p ?Y can match, we may need to know if ?X is the parent of ?Y. We capture this with auxiliary variables ?Pxy which we bind to one of two possible values (interpreted as *true* and *false*).

We thus introduce variables ?Pxy for every pair of CQ variables x and y where y is non-distinguished. This completely specifies the parenthood of the matches. Together with the generating OWL restriction represented by ?Mx, this gives us enough information to verify property assertions. To find all matches of a CQ, one has to allow for the possibility that several query variables represent the same element of the chase. To capture this, we introduce variables ?Exy that tell us if the values of x and y are equal; again we use two possible values to represent *true* and *false*. Additional conditions in our query will ensure that there are no cycles in the parenthood relation, and that equal values are indeed equal. Many of these can be encoded in propositional logic, as explained next.

5.2 Expressing Propositional Logic in SPARQL 1.1

Our intuitive explanation above uses "Boolean" variables like ?Pxy and ?Exy, which can have one of two values. Moreover, the bindings of these variables should obey further constraints. For example, if x is the parent of y and y is identified with z, then x is the parent of z. This corresponds to a propositional logic implication ?Pxz $\wedge$?Eyz $\rightarrow$?Pxz.

We express this using the VALUES feature of SPARQL 1.1, which allows us to assign a fixed set of bindings to a list of variables. For example, the pattern VALUES (?Pxy){(<http://example.org/true>)(<http://example.org/false>)} has exactly two solutions, binding ?Pxy to one of the given URIs. The URIs used here are irrelevant, and it does not even matter if they occur in the data; we thus use the abbreviations T and F to denote two distinct URIs that we use to represent Boolean values. Propositional

logic formulae can now be represented by encoding their truth table using VALUES. For example, the implication ?Pxz ∧ ?Eyz → ?Pxz can be expressed as:

$$\text{VALUES (?Pxy ?Eyz ?Pxz)\{(F F F)(T F F)(F T F)(F F T)(T F T)(F T T)(T T T)\}.} \quad (31)$$

We denote this pattern as ⟦?Pxz ∧ ?Eyz → ?Pxz⟧, and similarly for any other propositional logic formula over SPARQL variables. The solutions to (31) are exactly the truth assignments under which the implication holds. In particular, every solution requires each of the three variables to be bound to T or F (and thus to never be undefined).

5.3 A Schema-Agnostic Rewriting for Conjunctive Queries

We now specify the complete rewriting of CQs in SPARQL 1.1, which consists of rewritings for the individual triple patterns and several additional patterns to ensure that the bindings of all (auxiliary) variables are as intended. Consider a CQ q with variables $\mathsf{Var}(q)$, partitioned into the set $\mathsf{Var}_d(q)$ of distinguished variables and $\mathsf{Var}_n(q)$ of non-distinguished variables. Our encoding uses the following sets of SPARQL variables:

- for every $x \in \mathsf{Var}(q)$, a variable ?Mx (encoding the "match for x").

In addition, we use the following propositional SPARQL variables:

- for every $x \in \mathsf{Var}(q)$, a variable ?Nx ("x is a named individual").
- for every pair $x, y \in \mathsf{Var}(q)$, a variable ?Exy ("x is equal to y");
- for every pair $x \in \mathsf{Var}(q)$ and $y \in \mathsf{Var}_n(q)$, a variable ?Pxy ("x is the parent of y");
- for every pair $x, y \in \mathsf{Var}_n(q)$, a variable ?Axy ("$x$ is an ancestor of y");

The variables ?Axy are used to encode the transitive closure over the parent relations on non-distinguished variables; this is necessary to preclude cyclic ancestries. We use PropConstraints(q) to denote the SPARQL encoding of all of the following implications (for every possible combination of the above variables, if no other condition is given):

$$\text{for } x \in \mathsf{Var}_d(q)\text{: } \mathsf{T} \rightarrow \text{?Nx}$$

$$\begin{array}{rcc} \text{?Exy} \rightarrow \text{?Eyx} & \text{?Exy} \wedge \text{?Nx} \rightarrow \text{?Ny} & \text{?Pxy} \rightarrow \text{?Axy} \\ \text{?Exy} \wedge \text{?Eyz} \rightarrow \text{?Exz} & \text{?Exy} \wedge \text{?Pxz} \rightarrow \text{?Pyz} & \text{?Axy} \wedge \text{?Ayz} \rightarrow \text{?Axz} \\ \text{?Pxz} \wedge \text{?Pyz} \rightarrow \text{?Exy} & \text{?Exy} \wedge \text{?Pzx} \rightarrow \text{?Pzy} & \text{?Axx} \rightarrow \mathsf{F} \end{array}$$

The previous conditions do not ensure yet that the bindings for ?Mx and ?My are the same whenever ?Exy is true. This cannot be encoded using VALUES. Instead, we define EqualityFilter(q) to be the condition of the following filter conditions:

$$\text{FILTER(?Exy = F || ?Mx = ?My)} \qquad x, y \in \mathsf{Var}(q)$$

We can now define the rewriting of the actual query conditions. For readability, we use ⟦?V := u⟧ to abbreviate VALUES (?V){(u)}. The triple pattern x rdf:type c is rewritten into the following pattern, denoted Rewrite(x rdf:type c):

```
        {[[?Nx := T]] . [[?Mx rdf:type c]]}
UNION {univClass[c]}
UNION {[[?Nx := F]] . ?E subClassOf c
        {{?Mx owl:someValuesFrom ?E} UNION
         {?Mx (owl:onProperty / subPropertyOf / rdfs:range) ?E} UNION
         {?Mx (owl:onProperty / subInvPropertyOf / (^owl:onProperty | rdfs:domain)) ?E}}
```

A triple pattern $x\ p\ y$ is rewritten into the following pattern, denoted Rewrite($x\ p\ y$):

```
            {[[?Nx := T]] . [[?Ny := T]] . [[?Mx p ?My]]}
UNION {univProperty[p]}
UNION {[[?Ny := F]] . [[?Pxy := T]] . ?My (owl:onProperty / subPropertyOf) p
        {Rewrite(x rdf:type ?My)}
UNION {[[?Nx := F]] . [[?Pyx := T]] . ?Mx (owl:onProperty / subInvPropertyOf) p
        {Rewrite(y rdf:type ?Mx)}
```

Note that the parenthood relationship ?Pyx is only relevant for checking certain triple patterns. In each of these cases, we verify that the parent element is really capable of creating the required child. This ensures that all assumed parenthoods that are relevant to prove the query are really derived. In addition, we still need to check that all anonymous elements are really derived (from some original ancestor element in the ontology).

Example 2. Consider an OWL ontology with the assertion :peter rdf:type :Person and the axiom :Person rdfs:subClassOf [owl:onProperty :mother; owl:someValuesFrom :Woman]. Then the query {?X rdf:type :Woman} with ?X non-distinguished has a match. However, if we remove the triple :peter rdf:type :Person, then the query does not have a match. In contrast, our pattern Rewrite(*x* rdf:type :Mother) could match in either case.

To fix this, we introduce, for every non-distinguished variable x, an additional pattern MatchExists(x) that verifies that an element of the assumed type is actually derived. This pattern also ensures that named individuals are always bound to individuals. Anonymous individuals may be inferred from our assumption that the domain is not empty, or they must be derived from a named individual, which we represent by a bnode:

```
        {[[?Nx := T]] . individual(?Mx)}
UNION {[[?Nx := F]] . [[?Mx := owl:Thing]]}
UNION {[[?Nx := F]] . [[_:b rdf:type ?E]] . ?E (rdfs:subClassOf | intListMember |
        (owl:onProperty / (Inv | SpoEqp)* / (^owl:onProperty | rdfs:domain | rdfs:range)) |
        ^owl:equivalentClass | owl:equivalentClass | owl:someValuesFrom)* ?Mx}
```

We do not need to check that this derivation agrees with the guessed parenthood relations, since the check is only relevant for the elements that do not have a parent represented by a query variable.

Definition 6. *The rewriting* Rewrite(q) *a CQ q with distinguished variables $x_1, \ldots, x_n$ is the following SPARQL 1.1 query:*

```
SELECT DISTINCT ?Mx1, ..., ?Mxn WHERE {
  PropConstraints(q)
  Rewrite(x rdf:type c) for each condition x rdf:type c in q
  Rewrite(x p y) for each condition x p y in q
  MatchExists(x) for each variable x in q
  EqualityFilter(q)
}
```

Theorem 4. *The answers of a conjunctive query q over an OWL QL ontology G are exactly the answers of the SPARQL 1.1 query* Rewrite(q) *over G under simple entailment.*

6 Limits of Schema-Agnostic Query Rewriting

We have seen that schema-agnostic query rewriting works for (almost) all of OWL QL, so it is natural to ask how far this approach can be extended. In this section, we outline the natural limits of SPARQL 1.1 as a query rewriting language, point out extensions to overcome these limits.

In Section 3, we excluded owl:SymmetricProperty from our considerations. Indeed, schema-agnostic SPARQL 1.1 queries cannot support this feature. This might be surprising, given that one can write p rdf:type owl:SymmetricProperty as p rdfs:subPropertyOf [owl:inverseOf p]. To see why this problem occurs, consider the following ontology:

```
:c rdfs:subClassOf [rdf:type owl:Restriction; owl:onProperty :p; owl:someValuesFrom owl:Thing] .
[] rdf:type owl:Restriction; owl:onProperty [owl:inverseOf :p]; owl:someValuesFrom owl:Thing;
   rdfs:subClassOf :d .
:p rdf:type owl:SymmetricProperty .
```

This ontology states: every :c has an outgoing :p property; everything with an incoming :p property is a :d; and :p is symmetric. Clearly, this implies that :c is a subclass of :d. We call this ontology G(:c, :p, :d). Now assume that we have a chain of such ontologies $G_n := G$(:c1, :p1, :c2), ..., G(:cn, :pn, :d). Clearly, G_n implies that :c1 rdfs:subClassOf :d, but there is no SPARQL 1.1 graph pattern with property paths that recognizes this triple structure in an ontology. The intuitive explanation is that G_n contains a property path of length $4n$ that matches the following expression:

$$(\text{rdfs:subClassOf} \,/\, \text{owl:onProperty} \,/\, \hat{}\text{owl:onProperty} \,/\, \text{rdfs:subClassOf})^*$$

A SPARQL query that matches G_n for any n needs to use such a path expression; no other feature in SPARQL 1.1 can navigate arbitrary distances. However, it is impossible to verify that each :pi on this path is of type owl:SymmetricProperty. For the formal proof, we analyze general properties of the graphs that a SPARQL 1.1 query matches [3]. The essence of our argument is that property paths, being linear, cannot reliably

detect an arbitrary number of individuals with more than two neighbors, as found in G_n.

While this limitation is hardly more than a syntactic inconvenience, one might ask if there are query languages that can deal with this type of encoding. Indeed, one possible approach is nSPARQL, which has been proposed as an extension of SPARQL 1.0 with a form of path expressions that can test for the presence of certain side branches in property paths [14]. Similar test expressions have been considered in OBDA recently [2]. These query languages can handle the RDF encoding of symmetric properties.

Besides such "syntactic" limitations, schema-agnostic query rewriting is also restricted by complexity theoretic arguments. Simply put, the reasoning task solved in this way can not be harder (computationally speaking) than the data complexity of the underlying query language. The data complexity of the subset of SPARQL used in this paper is NLogSpace: SPARQL 1.1 patters are a variant of positive regular path queries [7], which have NLogSpace data complexity (by translation to linear Datalog [9]); inline data (VALUES) does not affect data complexity; and final filtering with equality checks can clearly be implemented in logarithmic space. Since P is widely assumed to be strictly harder than NLogSpace (though no proof has been given yet), we can exclude many lightweight ontology languages:

Theorem 5. *If* P *is strictly harder than* NLogSpace*, then reasoning for the following ontology languages cannot be expressed in SPARQL 1.1 using property paths,* UNION*,* VALUES *and (in)equality filters:*

- *any subset of OWL with* owl:intersectionOf *in subclass positions, especially OWL EL and OWL RL;*
- *any subset of OWL with unrestricted* owl:someValuesFrom *in subclasses and superclasses (not limited to* owl:Thing*);*
- *the extension of OWL QL with regular property chain axioms.*

These complexity-theoretic limitations can only be overcome by using a more complex query language. Many query languages with P-complete data complexity can be found in the Datalog family of languages, which are supported by RDF databases like OWLIM and Oracle 11g that include rule engines.

7 Conclusions and Outlook

To the best of our knowledge, our work is the first to present a query rewriting approach for ontology-based data access in OWL QL that is completely independent of the ontology. The underlying paradigm of schema-agnostic query rewriting appears to be a promising approach that can be applied in many other settings. Indeed, two previous works, nSPARQL [14] and PSPARQL [1], independently proposed query-based mechanisms for reasoning in RDFS. While these works have not considered SPARQL 1.1, OWL QL, or arbitrary conjunctive queries, they still share important underlying ideas. We think that a common name is very useful to denote this approach to query rewriting.

In this paper, we have focused on laying the foundations for this new reasoning procedure. An important next step is to study its practical implementation and optimization. Considering the size of some of the queries we obtain, one would expect them to be

challenging for RDF stores. We have started to implement our approach in a prototype system [3], and initial experiments confirm this expectation. Encouragingly, however, executing rewritten queries seems to be feasible, even in the raw, unoptimized form they have in this paper. Future work will be concerned with developing further optimizations that can be used in practical evaluations.

Indeed, while the queries we obtain might be challenging for current RDF stores, large parts of the queries are fixed and can be optimized for. Our work thus reduces the problem of adding OWL QL reasoning support to RDF stores to a query optimization problem. This can also guide future work in stores, such as OWLIM, which implement reasoning with inference rules: rather than trying to materialize (part of) an infinite OWL QL chase [4], they could materialize (sub)query results to obtain a sound and complete procedure. This provides completely new perspectives on the use of OWL QL in areas that have hitherto been reserved to OWL RL and RDFS.

Finally, our work also points into several interesting directions for foundational research, as mentioned in Section 6. Promising approaches include development of schema-agnostic rewriting procedures for languages like OWL EL that cannot be captured by SPARQL 1.1, and the development of query languages that suit this task [17].

Acknowledgements. This work has been funded by the Vienna Science and Technology Fund (WWTF, project ICT12-015), and by the DFG in project DIAMOND (Emmy Noether grant KR 4381/1-1).

References

1. Alkhateeb, F.: Querying RDF(S) with Regular Expressions. Ph.D. thesis, Université Joseph Fourier – Grenoble 1 (2008)
2. Bienvenu, M., Calvanese, D., Ortiz, M., Šimkus, M.: Nested regular path queries in description logics. CoRR abs/1402.7122 (2014)
3. Bischof, S., Krötzsch, M., Polleres, A., Rudolph, S.: Schema-agnostic query rewriting in SPARQL 1.1: Technical report (2014), `http://stefanbischof.at/publications/iswc14/`
4. Bishop, B., Bojanov, S.: Implementing OWL 2 RL and OWL 2 QL rule-sets for OWLIM. In: Dumontier, M., Courtot, M. (eds.) Proc. 8th Int. Workshop on OWL: Experiences and Directions (OWLED 2011). CEUR Workshop Proceedings, vol. 796. CEUR-WS.org (2011)
5. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. J. Automated Reasoning 39(3), 385–429 (2007)
6. Di Pinto, F., Lembo, D., Lenzerini, M., Mancini, R., Poggi, A., Rosati, R., Ruzzi, M., Savo, D.F.: Optimizing query rewriting in ontology-based data access. In: Proceedings of the 16th International Conference on Extending Database Technology, pp. 561–572. ACM (2013)
7. Florescu, D., Levy, A., Suciu, D.: Query containment for conjunctive queries with regular expressions. In: Mendelzon, A.O., Paredaens, J. (eds.) Proc. 17th Symposium on Principles of Database Systems (PODS 1998), pp. 139–148. ACM (1998)
8. Glimm, B., Ogbuji, C. (eds.): SPARQL 1.1 Entailment Regimes. W3C Recommendation (March 21, 2013), `http://www.w3.org/TR/sparql11-entailment/`
9. Gottlob, G., Papadimitriou, C.H.: On the complexity of single-rule datalog queries. Inf. Comput. 183(1), 104–122 (2003)

10. Harris, S., Seaborne, A. (eds.): SPARQL 1.1 Query Language. W3C Recommendation (March 21, 2013), http://www.w3.org/TR/sparql11-query/
11. Kontchakov, R., Rodríguez-Muro, M., Zakharyaschev, M.: Ontology-based data access with databases: A short course. In: Rudolph, S., Gottlob, G., Horrocks, I., van Harmelen, F. (eds.) Reasoning Weg 2013. LNCS, vol. 8067, pp. 194–229. Springer, Heidelberg (2013)
12. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language: Profiles. W3C Recommendation (October 27, 2009), http://www.w3.org/TR/owl2-profiles/
13. Patel-Schneider, P.F., Motik, B. (eds.): OWL 2 Web Ontology Language: Mapping to RDF Graphs. W3C Recommendation (October 27, 2009), http://www.w3.org/TR/owl2-mapping-to-rdf/
14. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A navigational language for RDF. J. Web Semantics 8, 255–270 (2010)
15. Pérez-Urbina, H., Motik, B., Horrocks, I.: Tractable query answering and rewriting under description logic constraints. J. Applied Logic 8(2), 186–209 (2010)
16. Rodríguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-based data access: Ontop of databases. In: Alani, H., et al. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 558–573. Springer, Heidelberg (2013)
17. Rudolph, S., Krötzsch, M.: Flag & check: Data access with monadically defined queries. In: Hull, R., Fan, W. (eds.) Proc. 32nd Symposium on Principles of Database Systems (PODS 2013), pp. 151–162. ACM (2013)