Discovery and Diagnosis of Wrong SPARQL Queries with Ontology and Constraint Reasoning

Jesús M. Almendros-Jiménez, Antonio Becerra-Terón

Dept. of Informatics, University of Almería, 04120, Almería (Spain)

Abstract

The discovery and diagnosis of wrong queries in database query languages have gained more attention in recent years. While for imperative languages well-known and mature debugging tools exist, the case of database query languages has traditionally attracted less attention. SPARQL is a database query language proposed for the retrieval of information in Semantic Web resources. RDF and OWL are standardized formats for representing Semantic Web information, and SPARQL acts on RDF/OWL resources allowing to retrieve answers of user's queries. In spite of the SPARQL apparent simplicity, the number of mistakes a user can make in queries can be high and their detection, localization, and correction can be difficult to carry out. Wrong queries have as consequence most of the times empty answers, but also wrong and missing (expected but not found) answers. In this paper we present two ontology and constraint reasoning based methods for the discovery and diagnosis of wrong queries in SPARQL. The first method is used for detecting wrongly typed and unsatisfiable queries. The second method is used for detecting mismatching between user intention and queries, reporting incomplete, faulty queries as well as counterexamples. We formally define the above concepts and a list of examples to illustrate the methods is shown. A Web online tool has been developed to analyze SPARQL queries according to the proposed methods.

Keywords: SPARQL; Semantic Web; Debugging; Static Analysis; Ontology Reasoning; Constraint Reasoning

Email addresses: `jalmen@ual.es` (Jesús M. Almendros-Jiménez), `abecerra@ual.es` (Antonio Becerra-Terón)

1. Introduction

The debugging of code in programming languages is one of the most costly and time consuming tasks. Modern programming languages are usually equipped with programming environments assisting compile-time debugging from static analysis of code. Additionally, type systems help to detect most programming errors coming from wrong combinations of language constructors. Type systems and debugging tools have been widely developed for imperative programming languages, however database programming languages require similar code analysis tools in order to avoid violation of integrity constraints, data protection, malicious code, etc.

SPARQL (Harris and Seaborne, 2013) is a query language designed for the retrieval of information in *Semantic Web* resources. *RDF (Resource Description Framework)* (The RDF Working Group, 2014) and *OWL (Web Ontology Language)* (The OWL Working Group, 2012) are standardized formats for representing Semantic Web information, and SPARQL acts on RDF/OWL resources to retrieve user's queries. Two main mechanisms are used in SPARQL to express queries: *triple patterns* to which *RDF/OWL triples* have to be matched, and *filter conditions*, enabling to impose ranges for *literals*.

While SPARQL is apparently a SQL-like simple language based on triple pattern matching, it is in a certain sense a type language since RDF/OWL resources are usually equipped with a type signature in the RDF Schema, and OWL type assertions, respectively. A suitable type checking tool could detect that queries lead to an *empty answer* because they are not well typed, without requiring query evaluation, and without taking into account RDF/OWL instances.

Moreover, SPARQL works with OWL datasets with a rich variety of integrity constraints using OWL classes. OWL classes can have complex definitions, involving set operators and universally and existentially quantified formulas. SPARQL queries for OWL resources can require answers of *unsatisfiable formulas (i.e., triple patterns)*. Such unsatisfiability can be checked by proving the *inconsistency* of triple patterns and OWL restrictions, without requiring query evaluation, and without taking into account OWL instances. Under an *OWL semantics*, it would serve again to detect an *empty answer* in SPARQL.

Additionally, an *entailment regime* is usually defined for OWL axioms. Such entailment relationship can be used to prove properties from OWL axioms that the programmer expects on answers from SPARQL queries. A suitable debugger could find programming errors by asking the programmer about properties on the answers and checking whether these properties are satisfied from OWL axioms without requiring query evaluation, and without taking into account OWL instances. Under an *OWL semantics*, it would serve to detect *missing (expected but not found)* or *wrong* answers from programmer intention.

Moreover, a suitable debugger should be able to give enough information to the programmer in order to rewrite the query and to remove the programming bug. Most of cases, an empty answer is solved by modifying class or property names, or by renaming variables. In case of missing or wrong answers, the query should be modified/completed or some elements should be removed. In these cases, also a *counterexample* of the property required by the programmer on answers can help to detect such mistakes.

In this paper we have designed and implemented two methods for the static analysis of SPARQL queries in which *ontology* and *constraint based reasoning* are used for detecting wrong queries. The first method (*correctness method*) is used for detecting *wrongly typed* and *unsatisfiable* queries. The second method (*type validity method*) is used for detecting *mismatching* between *programmer intention* and queries, reporting *incomplete*, *faulty* queries as well as *counterexamples*. Both methods work at compile-time and only use the *ontology schema*; that is, without taking into account the *ontology instances* and without query evaluation.

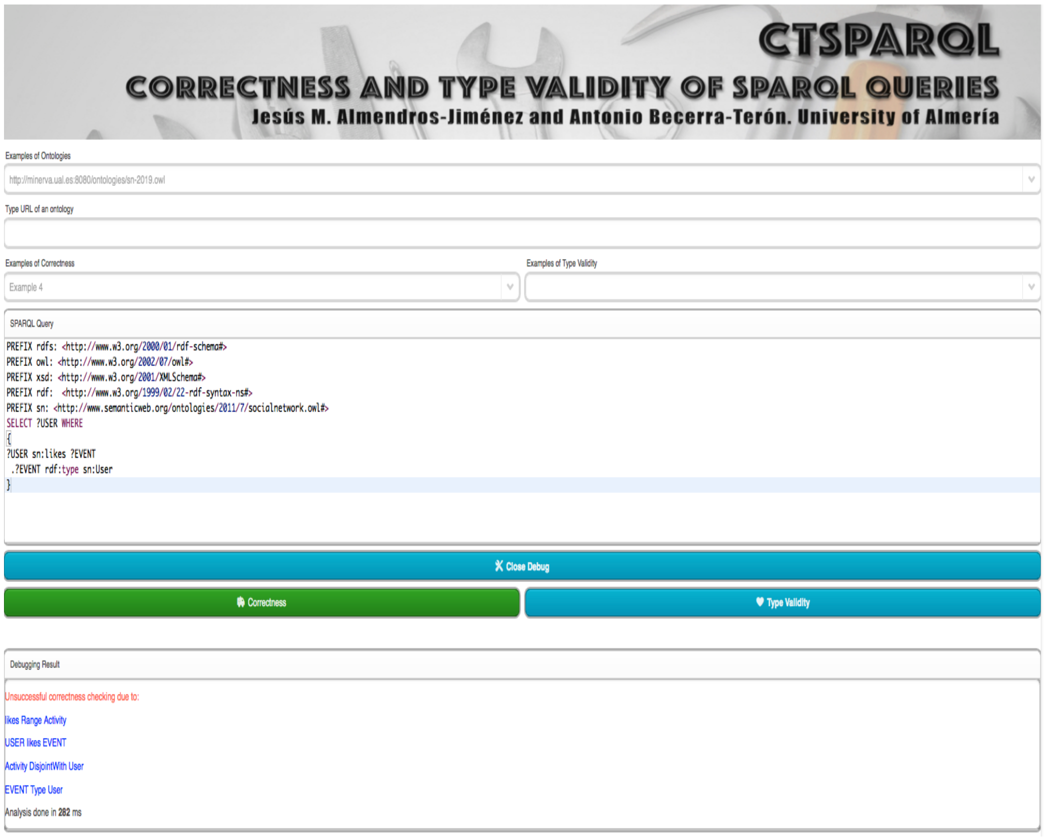
As a result of the analysis of a certain query, the correctness method is able to *automatically* detect at *compile-time* that *the set of answers is empty*, because triple patterns are wrongly typed or unsatisfiable for such filter conditions. The ill typed-ness or unsatisfiability of *triple patterns* means that such triple patterns cannot be matched to the input database which, in the case of SPARQL, it is a certain RDF/OWL ontology. Triple patterns are used in SPARQL to match variables to RDF/OWL triples but in the case such variable binding violates *RDF typing* or *OWL consistency*, we can say that triple patterns are wrongly typed or they are unsatisfiable, respectively, with regard to the input RDF/OWL ontology. *Filter conditions* can be wrongly typed when both sides of equalities and inequalities do not have a common type, and they can be unsatisfiable when no value can be found to variables satisfying all the imposed ranges. But even when filter conditions are well-typed, the *combination of both triple patterns and filter conditions* can lead to incorrectness, due to restrictions imposed by the OWL ontology on certain property values. Incorrect queries are detected from RDFS and OWL axioms using ontology and constraint reasoning.

The proposed type validity method is able to *automatically* detect at compile-time that *the set of answers is not empty but incorrect*, because even when triple patterns or filter conditions are well-typed and satisfiable, there is a mismatching between programmer intention and queries. The user/programmer of the query has a certain *intention/conviction* when he/she writes a query, and he/she has in mind an *expected set of answers*. A set of answers is incorrect from the user's point of view, when either there are *missing (expected but not found)* answers or there are *wrong* answers. Thus, the concept of incorrectness strongly depends on the programmer intention. In order to help the frustrated user, the method permits to prove a *variable typing*, which is imposed by the user. The imposed type is an *ontology class*, which can be, in the context of OWL, a complex class, involving properties, property values, and membership to (complex) classes. In particular, existentially quantified class definitions.

By proving variable typing the method automatically finds at compile-time: *incomplete queries*, with missing triple patterns and filter conditions, *faulty queries*, in which the imposed variable typing is not possible by ontology/constraint inconsistency, as well as *counterexamples* of the imposed variable typing; that is, feasible answers which do not satisfy the imposed variable typing. In order to prove the variable typing, the method also uses ontology and constraint reasoning.

Both methods are not only able to detect wrong queries but also to provide a *diagnosis/symptom* for the mistakes. In the correctness method (i.e., wrongly typed and unsatisfiable queries) the causes of the ill typed-ness and unsatisfiability are reported. In the type validity method either missing triple patterns and filter conditions; or causes of the ontology/constraint inconsistency; or counterexamples to the imposed type are reported.

Finally, the proposed methods have some limitations. Firstly, we restrict the form of SPARQL queries. We consider a subset of SPARQL containing a conjunction of triple patterns and filter conditions, which is equivalent to a SELECT expression, free of FILTER OR, MINUS, OPTIONAL, and (NOT) EXISTS, as well as aggregation (SUM, COUNT, AVG, MIN and MAX). While FILTER OR, MINUS and OPTIONAL, (NOT) EXISTS could be still added to our framework without a great effort, by adding disjunctions of formulas, the limitation of constraint reasoning to handle aggregation reduces the current applicability of the approach. An extension of the proposed meth-


CTSPARQL
CORRECTNESS AND TYPE VALIDITY OF SPARQL QUERIES
Jesús M. Almendros-Jiménez and Antonio Becerra-Terón. University of Almería

Examples of Ontologies: http://minerva.ual.es:8080/ontologies/sn-2019.owl

Type URL of an ontology:

Examples of Correctness: Example 4

Examples of Type Validity:

SPARQL Query

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sn: <http://www.semanticweb.org/ontologies/2011/7/socialnetwork.owl#>
SELECT ?USER WHERE
{
?USER sn:likes ?EVENT
 .?EVENT rdf:type sn:User
}
```

Close Debug
Correctness
Type Validity

Debugging Result

```
Unsuccessful correctness checking due to:
likes Range Activity
USER likes EVENT
Activity DisjointWith User
EVENT Type User
Analysis done in 282 ms
```

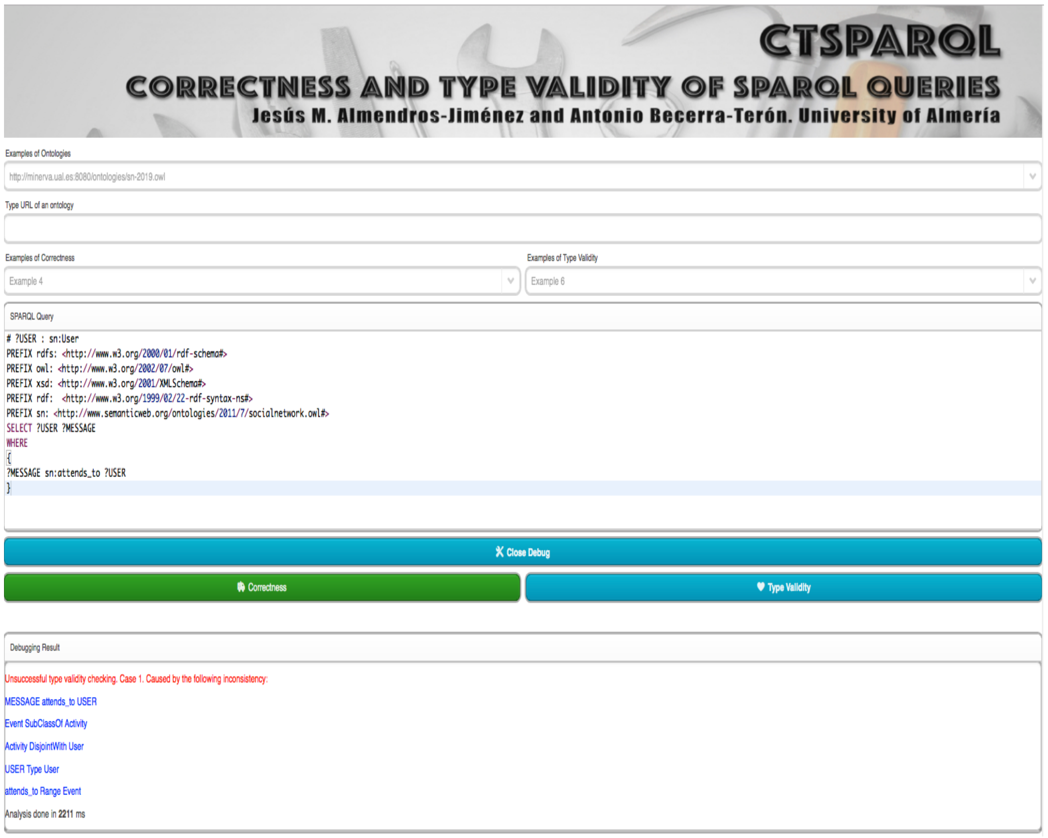
Figure 1: Web Tool of CTSPARQL: Correctness.

ods to cover richer SPARQL constructors and aggregators will be considered as future work. Additionally, the kind of ontology axioms is restricted for the same reasons as for SPARQL constructors.

An online Web tool called *CTSPARQL* (Correctness and Type Validity Tool for SPARQL) to analyze SPARQL queries has been developed (see Figures 1 and 2) and it is available at `http://minerva.ual.es:8090/CTSPARQL/`, allowing also to import and visualize ontologies.

The methods have been implemented using *ARQ Jena SPARQL API*[1], *OWL API*[2] (Horridge and Bechhofer, 2011), *Pellet OWL DL Reasoner*[3]

[1]https://jena.apache.org/documentation/query/.
[2]https://github.com/owlcs/owlapi/.
[3]https://github.com/stardog-union/pellet.


CTSPARQL
CORRECTNESS AND TYPE VALIDITY OF SPARQL QUERIES
Jesús M. Almendros-Jiménez and Antonio Becerra-Terón. University of Almería

Examples of Ontologies: http://minerva.ual.es:8080/ontologies/sn-2019.owl

Type URL of an ontology:

Examples of Correctness: Example 4

Examples of Type Validity: Example 6

SPARQL Query

```
# ?USER : sn:User
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sn: <http://www.semanticweb.org/ontologies/2011/7/socialnetwork.owl#>
SELECT ?USER ?MESSAGE
WHERE
{
?MESSAGE sn:attends_to ?USER
}
```

Close Debug
Correctness
Type Validity

Debugging Result

```
Unsuccessful type validity checking. Case 1. Caused by the following inconsistency:
MESSAGE attends_to USER
Event SubClassOf Activity
Activity DisjointWith User
USER Type User
attends_to Range Event
Analysis done in 2211 ms
```

Figure 2: Web Tool of CTSPARQL: Type Validity.

(Sirin et al., 2007), as well as *CLP(FD)*[4] (Triska, 2012) and *CLP(R)*[5] (Holzbaur, 1995) constraint solvers of *SWI-Prolog*[6]. The source code of the tool is freely available at `https://github.com/jalmenUAL/CTSPARQL`.

With regard to the performance, let us remark that all the proposed methods never reason with the ontology instances of the input ontology. Thus, even when the input ontology is large (due to high number of ontology instances), the only elements used to detect correctness and type validity are ontology axioms, which frequently are few. In our framework, the reasoner uses ontology axioms as well as triple patterns occurring in SPARQL queries

[4]https://www.swi-prolog.org/man/clpfd.html.
[5]https://www.swi-prolog.org/pldoc/man?section=clpqr.
[6]https://www.swi-prolog.org/.

(as ontology instances) which are also few. On the other hand, debugging is intended to be a compile-time process, previous and separate from query evaluation. Even when the compile-time process is costly (for a high number of ontology axioms), the debugging process ensures two main goals for programmers: semantically correct queries, and thus, free of bugs software. In any case, the time spent for the debugging process is generally not so crucial.

2. Related Work

While for imperative languages well-known and mature *debugging* tools exist, the case of *database query languages* has traditionally attracted less attention. However, proposals about SQL for stating a list of feasible *errors* (Brass and Goldberg, 2006, 2005), and studies about *test case generation* and *testing* techniques, including *mutation* methods, have been proposed (see e.g. (Javid and Embury, 2012; Shah et al., 2011; Caballero et al., 2010; De La Riva et al., 2010; Gupta et al., 2010; Suárez Cabal and Tuya González, 2009; Tuya et al., 2008, 2007, 2006; Chays et al., 2004; Zhang et al., 2001)). Also for SQL, debugging techniques have been more recently proposed (see e.g. (Guagliardo and Libkin, 2016; Dietrich and Grust, 2015; Tzompanaki, 2014; Caballero et al., 2012)). This is not the only case, because for the *XML* database query languages *XQuery* (Robie et al., 2012) and *XPath* (Clark and DeRose, 1999) test case generation and testing have been also studied (see e.g.(Almendros-Jiménez and Becerra-Terón, 2017b; Kim-Park et al., 2010; Bertolino et al., 2010, 2007b,a,c; De La Riva et al., 2006)).

SPARQL query optimization has been studied in some works (Yakovets et al., 2016; Atre, 2015; Letelier et al., 2013; Tsialiamanis et al., 2012; Le et al., 2012; Schmidt et al., 2010; Liu et al., 2010; Vidal et al., 2010), and in most of them static analysis of SPARQL is carried out. Benchmarking datasets have been also proposed (Saleem et al., 2018; Pan et al., 2018; Bagan et al., 2017; Saleem et al., 2016, 2015b,a; Qiao and Özsoyoğlu, 2015; Zhang et al., 2012; Görlitz et al., 2012; Morsey et al., 2011; Schmidt et al., 2009; Bizer and Schultz, 2009; Guo et al., 2005) in order to to analyze performance of SPARQL implementations (see in Table 3 a list of SPARQL implementations/endpoints). Also there are works about dataset analysis (Kontokostas et al., 2014a,b), but less attention has been paid to SPARQL query debugging.

Name	*URL*
ARQ SPARQL Jena	`https://jena.apache.org/documentation/query/`
Protégé SPARQL Tab	`http://protegewiki.stanford.edu/wiki/SPARQL\_Query`
Twinkle: SPARQL Tools	`http://www.ldodds.com/projects/twinkle/`
Virtuoso SPARQL Query Editor	`http://dbpedia.org/sparql`
SPARQLer - General purpose processor	`http://www.sparql.org/sparql.html`
Redland Rasqal RDF Query Demonstration	`http://librdf.org/query`
OpenUpLabs	`http://openuplabs.tso.co.uk/sparql`
wordnet.rkbexplorer.com	`http://wordnet.rkbexplorer.com/sparql/`
DBPedia SNORQL	`http://dbpedia.org/snorql/`
SPARQL editor | BNB	`http://bnb.data.bl.uk/flint-sparql`
YASGUI SPARQL Editor	`http://cliopatria.swi-prolog.org/yasgui/index.html`
SPARQL Carsten Editor	`http://sparql.carsten.io/`
ViziQuer Web tool	`https://viziquer.lumii.lv/`
VISU:Visual SPARQL query tool	`https://github.com/jiemakel/visu`
ONTOP	`https://ontop.inf.unibz.it/`

Figure 3: SPARQL Implementations/Endpoints

Related with our proposal, entailment regimes have been proposed for SPARQL (Glimm et al., 2013; Kontchakov et al., 2014; Bischof et al., 2014) The goal of entailment regimes for SPARQL is to changes the semantics of SPARQL replacing basic graph pattern matching with matching to semantically entailed relations. Queries and graphs have to be well-formed for the given regime, in order to avoid, for instance, infinite solutions and undecidability issues. The goal of our framework is different. The goal is to detect empty, missing and incorrect answers from ontology axioms whatever ontology instances are. Our framework assumes basic graph pattern matching, although also covers the case of the SPARQL under OWL 2 DL entailment regime. In other words, our framework is grounded in the following statement: Whatever the search space is (basic graph pattern or OWL 2 DL entailment regime), no answer can be found when the triple patterns of the query are unsatisfiable from ontology axioms. Unsatisfiability covers also the case of entailed relations. The fact that our framework works with basic graph pattern matching as well as SPARQL under OWL 2 DL entailment is due to the model theoretic approach we adopted. In such theoretical formulation, classes, properties and individuals are interpreted in a certain domain for a given interpretation, and thus independently of the considered triple pattern matching regime.

Finally, our group has recently worked in the design and development of a *property-based testing* method for SPARQL queries (Almendros-Jiménez and Becerra-Terón, 2017c,a). The property-based testing method generated test cases for a SPARQL query from the ontology schema and a set of values provided by the user. Once provided, the ontology schema was pruned in order to generate relevant test cases. Finally, (input and output) properties (variable typing) were validated with ontology reasoning from test cases. Thus both property-based testing and type validity methods are able to prove variable typing. However, here, the proposed type validity method uses constraint reasoning avoiding test case generation, and *improving the property-based testing method*. While the property-based testing method greatly depends on the number of test cases and the set of values provided by the user, and thus, being useless in some particular situations, here, the proposed type validity method explores all the possible test cases/values, using constraint reasoning.

The structure of the paper is as follows. Section 3 well show a list of examples to motivate the approach. Section 4 will introduce ontologies, queries, constraints and answers. Section 5 will define correct queries and valid types. Section 6 will describe the correctness method. Section 7 will show the type validity method. Section 8 will summarize additional elements of the implementation. Finally, Section 9 will draw conclusions and future work.

3. Wrong Queries

We consider OWL 2 DL ontologies, and as an example of ontology, used in the rest of the paper, we can consider the shown one in Figure 4. The ontology example defines a *social network* in which there are two main *classes* *User* and *Activity* in order to represent users and activities of users in the social network, respectively. *User* and *Activity* are disjoint classes. *Activity* has as *subclasses* *Message* and *Event* which are disjoint and whose members are *created_by* users; that is, *created_by* is an *object property* that maps *Activity* to *User* elements. *created_by* property has as *subproperties* *sent_by* and *added_by*: *sent_by* and *added_by* are used for *Message* and *Event* elements, respectively. *created_by* has an *inverse* property called *creates*. Users *likes* activities (the inverse of *likes* is *liked_by*). A *User* *attends_to* *Event*s. A *User* can be *friend_of* another one, being *friend_of* an *irreflexive* property. A *User* *shares* an *Activity*, and an *Activity* is *shared_by* a *User* (thus, *shares* and *shared_by* are the inverse of each other). With regard to *data properties*, we consider *name*, *dailyActivity* and *dailyLikes* for users. Finally, we consider four complex subclasses of *User*: *Active* users having only values of *dailyActivity* greater or equal than *5* (instance of OWL 2 DL *DataAllValuesFrom*),

Classes
Active, *Influencer*, *SocialLeader* $\sqsubseteq$ *User*
Message, *Event*, *Shared* $\sqsubseteq$ *Activity*
User $\cap$ *Activity* $= \emptyset$
Message $\cap$ *Event* $= \emptyset$
Active users have only *dailyActivity* values greater or equal than *5*:
Active $\equiv DataAllValuesFrom(dailyActivity\ (>= 5))$
Influencer users have some *dailyLikes* value greater or equal than *100*:
Influence $\equiv DataSomeValuesFrom(dailyLikes\ (>= 100))$
SocialLeader users who create some *Shared* activity:
SocialLeader $\equiv ObjectSomeValuesFrom(creates\ Shared)$
FriendOfInfluence users having as *friend_ of* an *Influencer*:
FriendOfInfluence $\equiv ObjectSomeValuesFrom(friend\_of\ Influencer)$
FriendOfActive users having as *friend_ of* an *Active* user:
FriendOfActive $\equiv ObjectSomeValuesFrom(friend\_of\ Active)$
Shared activities which are *shared_ by* some *User*:
Shared $\equiv ObjectSomeValuesFrom(shared\_by\ User)$

Properties	
created_ by : *Activity* $\rightarrow$ *User*	*added_ by*, *sent_ by* $\sqsubseteq$ *created_ by*
added_ by : *Event* $\rightarrow$ *User*	*sent_ by*: *Message* $\rightarrow$ *User*
creates inverse of *created_ by*	*likes*: *User* $\rightarrow$ *Activity*
liked_ by is the inverse of *likes*	*attends_ to* : *User* $\rightarrow$ *Event*
friend_ of: *User* $\rightarrow$ *User*	*friend_ of* is irreflexive
shares: *User* $\rightarrow$ *Activity*	*shared_ by* is the inverse of *shares*
name: *User* $\rightarrow$ *xsd:Integer*	*dailyActivity*: *User* $\rightarrow$ *xsd:Integer*
dailyLikes: *User* $\rightarrow$ *xsd:Integer*	

Figure 4: Ontology Example

Influencer having some *dailyLikes* value greater or equal than *100* (instance of OWL 2 DL *DataSomeValuesFrom*), *SocialLeader* who *creates* some *Shared* activity, *FriendOfActive* users have some *Active* friend, *FriendOfInfluencer* users have some *Influencer* friend, and *Shared* activities are activities shared by some *User* (instance of OWL 2 DL *ObjectSomeValuesFrom*).

Now, we would like to show some examples to motivate the approach. Let's start with some examples showing typical errors detected by the proposed correctness method (i.e., ill typed and unsatisfiable queries).

Example 1. *The following example illustrates the case of wrong query due to ill typed triple patterns.*

```
SELECT ?USER ?EVENT
WHERE {
  ?USER sn:attends_to ?EVENT .
  ?USER sn:friend_of ?EVENT
  }
```

Here according to the type definitions, ?EVENT cannot be matched to a value in a (consistent) ontology. From the first triple pattern (i.e., ?USER attends_to ?EVENT) the variable ?EVENT has type Event, while in the second triple pattern (i.e., ?USER friend_of ?EVENT) ?EVENT should have type User. Since Event and User are disjoint the SPARQL query is ill typed and thus the set of answers is empty.

Example 2. *The following example illustrates the case of wrong query due to unsatisfiable triple patterns.*

```
SELECT ?USER
WHERE {
  ?USER sn:friend_of ?USER
  }
```

Here according to OWL axioms, friend_of is an irreflexive property, and thus in a consistent ontology is not possible to find instances of friend_of property to match the triple pattern ?USER friend_of ?USER; therefore the set of answers is empty. In this case, we say that the SPARQL is unsatisfiable.

Now, we would like to show how the correctness method works. The idea is to transform the triple patterns in object/data property assertions. For instance, in the first example, ObjectPropertyAssertion(sn:attends_to _:USER _:EVENT) and ObjectPropertyAssertion(sn:friend_of _:USER _:EVENT) where _:USER and _:EVENT are ontology individuals/blank nodes. In the second example, ObjectPropertyAssertion(sn:friend_of _:USER _:USER). Now, once transformed, an ontology reasoner can be used in order to prove the consistency of such object/data property assertions with regard to RDF/OWL axioms.

From a theoretical point of view, a SPARQL query is an ontology instance with blank nodes representing variables, and the correctness of the SPARQL query is equivalent to the consistency of such ontology. Ill-typedness can be seen as a particular case of unsatisfiability caused by type definitions.

However, we can consider a new case of wrong queries, in which the combination of triple patterns and filter conditions cause unsatisfiability.

Example 3. *The following example illustrates how the combination of triple patterns and filter conditions can cause an empty set of answers.*

```
SELECT ?USER ?DA
```

```
WHERE {
  ?USER rdf:type sn:Active .
  ?USER sn:dailyActivity ?DA .
  FILTER (?DA <= 4)
  }
```

Here according to OWL definition of Active users, dailyActivity is greater or equal than 5, while the filter condition requests a number smaller or equal than 4. Thus the set of answers of empty.

In this case the correctness method works as follows:

(a) On one hand, the triple patterns ?USER rdf:type sn:Active and ?USER sn:dailyActivity ?DA are transformed into ClassAssertion(sn:Active _:-USER) and DataPropertyAssertion(sn:dailyActivity _:USER _:DA), respectively.

(b) On the other hand, the filter condition is transformed into a constraint ?DA <= 4.

Now, from the OWL axioms, and from ClassAssertion(sn:Active _:USER) and DataPropertyAssertion(sn:dailyActivity _:USER _:DA), we can obtain a new constraint ?DA >= 5, because Active users only have values of daily-Activity greater or equal than 5. Next, a constraint solver is used to prove the satisfiability of the constraint system ?DA <= 4 and ?DA >= 5, which is unsatisfiable.

From a theoretical point view, filter conditions introduce constraints on the values of data properties and thus, in general, a SPARQL query is a ontology instance including possibly variables (as blank nodes), together with a constraint system on the values of such variables. In this paper we will define the concept of constrained ontologies, which are combinations of ontologies with variables and constraint systems, and we will prove that the correctness of a SPARQL query is equivalent to the consistency of the constrained ontology obtained from the query. This is the main result of our proposed correctness method and it will proved in Section 6.

As previously, now we will motivate the proposed type validity method, which is a method for proving a variable typing from the query and whose goal is to detect a mismatching among user intention and query.

Example 4. *Let us suppose the following query:*

```
SELECT ?USER ?MESSAGE
WHERE {
  ?USER sn:creates ?MESSAGE .
  sn:antonio sn:likes ?MESSAGE
  }
```

And let us suppose the user/programmer wants to retrieve messages created by social leaders. Let us remark that type validity method is mainly useful when the user/programmer is not aware of the ontology axioms and instances, and thus there is mismatching between the user/programmer expresses in the query and the ontology vocabulary semantics. In this case, the query is well-typed and satisfiable, but the user/programmer finds that some answers are incorrect. In this case, the intention of user/programmer is different from the query definition and he/she can use the type validity method to check whether all the answers are correct. With this aim, the user/programmer can select the type that the variable should have. In the case, he/she selects "?USER should be SocialLeader". SocialLeader class represents users who create a Shared message. Shared messages are messages shared_by some User. In this case, from the query we cannot ensure that. It is due to the triple pattern ?MESSAGE rdf:type Shared is missing (or any other equivalent). The type validity method is able to report the fail as well as the cause of the fail. For instance, in this case, it reports that ?MESSAGE rdf:type Shared is missing. Thus, the type validity method can be seen as a query completion method according to the ontology axioms.

Example 5. *Next example shows that type validity method is also able to detect when a filter condition is missing.*

```
SELECT ?USER ?DL
WHERE {
  ?USER rdf:type sn:User .
  ?USER sn:dailyLikes ?DL
  }
```

Let us suppose now that ?USER should be an Influencer (i.e., someone that has at least one value for dailyLikes greater or equal than 100*) according to the user intention. Again, the query is well-typed and satisfiable. Now, the type validity method fails in order to prove that ?USER has type Influencer, reporting that there is a missing filter condition; i.e., ?DL >=* 100*. Thus, it helps to complete the query.*

Example 6. *The following example shows how the type validity method in some cases reports that a query is faulty, in the sense that the imposed type for the variables is incompatible with the query.*

```
SELECT ?USER ?MESSAGE
WHERE {
  ?MESSAGE sn:attends_to ?USER
  }
```

Here, we can assume, for instance, the user/programmer looks for messages send_by users. Instead of using send_by, he/she makes a mistake and uses attends_to. In this case, the user/programmer can check, for instance, that ?MESSAGE has type Message. In such a case, the method reports an inconsistency: Message and User are incompatible types. Alternatively, he/she can check the type User for ?USER, which again causes inconsistency from Event and User disjointness. Thus, the type validity method serves as query completion and repairing.

Example 7. *The following example illustrates the case of faulty query due to constraint inconsistency.*

```
SELECT ?USER ?DL
WHERE {
  ?USER rdf:type sn:User .
  ?USER sn:dailyLikes ?DL .
  FILTER (?DL < 50)
  }
```

Here, the user tries to retrieve Influencer users (i.e., having some value of dailyLikes greater or equal than 100*). In this case, the user can check the type Influencer for ?USER, and the method reports the inconsistency of ?DL < 50 and ?DL >=100. Thus, the filter condition should be modified in order to get influencer users.*

Example 8. *The following example illustrates the case of counterexample of variable typing.*

```
SELECT ?USER ?DL
WHERE {
  ?USER rdf:type sn:User .
  ?USER sn:dailyLikes ?DL .
  FILTER (?DL > 50)
  }
```

In this case let us suppose that the user requests the same variable typing (i.e., ?USER has type Influencer). The method reports counterexamples in which ?USER is not Influencer. Such values are those ones satisfying 50 < ?DL < 100.

Example 9. *Next example shows how the type validity method works for cardinality restrictions.*

```
SELECT ?USER
WHERE {
  ?USER sn:friend_of sn:jesus .
  ?USER sn:friend_of sn:antonio
  }
```

Let us suppose that Influencer is equivalent to ObjectMinCardinality(50 :friend_of User). From the given triple patterns we cannot infer that ?USER has type Influencer. The method will report in such a case "The type cannot be proved". However if Influencer is equivalent to ObjectMinCardinality(1 :friend_of User) it can be ensured. Whether Influencer is equivalent to ObjectMinCardinality(2 :friend_of User), Influencer cannot be ensured to ?USER either, since sn:jesus and sn:antonio can be equal.

The reader can wonder whether the type validity method is really needed. The user/programmer could modify the query to impose the required type.

Example 10. *For instance, the query:*

```
SELECT ?USER ?DL
WHERE {
  ?USER rdf:type sn:User .
  ?USER sn:dailyLikes ?DL
  }
```

can be modified to:

```
SELECT ?USER ?DL
WHERE {
  ?USER rdf:type sn:Influencer .
  ?USER sn:dailyLikes ?DL
  }
```

However, both queries are not equivalent if the ontology instance has not been fully materialized (i.e., not all the ontology logic consequences have been computed).

The method for type validity is similar to the correctness method. Each triple pattern becomes a class or object/data property assertion, and the filter condition is transformed into a constraint, forming a constrained ontology from ontology axioms. Additionally, the type imposed for variables also becomes a class assertion. The type validity method consists in proving the entailment of this class assertion from the constrained ontology. Again, the entailment from constrained ontologies can be reduced to entailment from ontologies and constraint systems. This is the main result of the type validity method and it will be proved in Section 7.

Example 11. *For instance, in the query:*

```
SELECT ?USER ?MESSAGE
WHERE {
  ?USER sn:creates ?MESSAGE .
  ?USER2 sn:shares ?MESSAGE
  }
```

and considering the type SocialLeader for ?USER, the type validity method proves that ClassAssertion(SocialLeader _:USER) is entailed from ObjectPropertyAssertion(sn:creates _:USER _:MESSAGE) and ObjectPropertyAssertion(sn:shares _:USER2 _:MESSAGE). And in the case:

```
SELECT ?USER ?DL
WHERE {
  ?USER rdf:type sn:User .
  ?USER sn:dailyLikes ?DL .
  FILTER (?DL > 50)
  }
```

and considering the type Influencer for ?USER, then the type validity method can prove that ?DL>=100 is entailed from ?DL > 50.

4. Ontologies, Queries and Answers

In this section we introduce the formal concepts of the paper. Firstly, we will define ontologies, ontology consistency and entailment. Next we will define queries, constraint system consistency and entailment, and finally, answers.

Definition 1 (Vocabulary of an Ontology). *A vocabulary of an ontology is a 6-tuple of the form* $VO = (VC, VOP, VDP, VI, DT, FACET)$*, where* VC*,* VOP*,* VDP*,* VI*,* DT *and* $FACET$ *are sets of classes, object properties, data properties, individuals (named and anonymous), datatype sets and facet sets, respectively.* VC *contains at least owl:Thing and owl:Nothing;* VOP *contains at least owl:topObjectProperty and owl:bottomObjectProperty; and* VDP *contains at least owl:topDataProperty and owl:bottomDataProperty. Finally,* $FACET = \{FACET^{D_i} \mid D_i \in DT\}$.

The set of resources of an ontology vocabulary, denoted by *Res(VO)*, contains the set of ontology vocabulary classes, object/data properties and individuals, that is, $Res(VO) = VC \cup VOP \cup VDP \cup VI$. The set of literals of an ontology vocabulary, denoted by *Lit(VO)*, contains the set of values of datatype sets, that is, $Lit(VO) = \cup_{D_i \in DT} D_i$.

Definition 2 (Ontology). *An ontology is* conjunction of ontology statements *about classes, object and data properties, and individuals of an ontology vocabulary. We restrict our framework to the following subset of OWL 2 DL statements*[7]*:*

- **Instances Assertions***: ClassAssertion, ObjectPropertyAssertion, DataPropertyAssertion, SameIndividual and DifferentIndividuals;*
- **Type Assertions***: ObjectPropertyDomain and ObjectPropertyRange, DataPropertyDomain and DataPropertyRange;*
- **Class Restrictions***: SubClassOf, EquivalentClasses, DisjointClasses and DisjointUnion; and finally,*

[7]We assumed the imposed OWL 2 DL restrictions (see `https://www.w3.org/TR/owl2-direct-semantics/`, Section 2.5) for avoiding undecidability issues.

- **Property Restrictions**: *SubObjectPropertyOf, EquivalentObjectProperties, SubDataPropertyOf, EquivalentDataProperties, DisjointObjectProperties,InverseObjectProperties, ReflexiveObjectProperty, IrreflexiveObjectProperty, SymmetricObjectProperty, AsymmetricObjectProperty,TransitiveObjectProperty, SubObjectPropertyOf in ObjectPropertyChain, FunctionalObjectProperty and InverseFunctionalObjectProperty;*

and they can use:

- **Object Property based Definitions**: *ObjectIntersectionOf, ObjectUnionOf, ObjectComplementOf, ObjectOneOf, ObjectSomeValuesFrom, ObjectAllValuesFrom, ObjectHasValue, ObjectHasSelf, ObjectMinCardinality,ObjectMaxCardinality and ObjectExactCardinality;*

- **Data Property based Definitions**: *DatatypeRestriction, DataIntersectionOf, DataSomeValuesFrom, DataAllValuesFrom and DataHasValue; and*

- **Object Property Definitions**: *ObjectInverseOf.*

The following OWL 2 DL statements are not considered. On one hand, NegativeDataPropertyAssertion and DisjointDataProperties assertions force to use disjunction in the constraint systems. The same can be said for DataUnionOf, DataComplementOf and DataOneOf. Finally, FunctionalDataProperty, HasKey, DataMinCardinality, DataMaxCardinality and DataExactCardinality require aggregation handling in constraint systems.

Definition 3 (Interpretation of an Ontology). *An interpretation I^O of an ontology is a 12-tuple of the form $I^O = (\Delta, \Sigma_{OP}, \Sigma_{DP}, \Gamma, I^C, I^P, I^{OP}, I^{DP}, I^{IN}, I^{LIT}, I^{DT}, I^{FACET})$, where:*

- *Δ (object domain), Σ_{OP} (object property domain), Σ_{DP} (data property domain) and Γ (literal domain) are not empty sets such that $\Sigma_{OP} \subseteq \Delta$, $\Sigma_{DP} \subseteq \Delta$ and $\Gamma \cap \Delta = \emptyset$.*

- *I^C, I^P, I^{OP}, I^{DP}, I^{IN}, I^{LIT}, I^{DT}, I^{FACET} are interpretation functions, where $I^C(C) \subseteq \Delta$ if $C \in VC$, $I^C(owl : Thing) = \Delta$ and $I^C(owl : Nothing) = \emptyset$; $I^P(p) \in \Sigma_{OP}$ if $p \in VOP$; $I^P(p) \in \Sigma_{DP}$ if $p \in VDP$; $I^{OP}(op) \subseteq \Delta \times \Delta$ if $op \in \Sigma_{OP}$, $I^{OP}(I^P(owl : topObjectProperty)) = \Delta \times \Delta$ and $I^{OP}(I^P(owl : bottomObjectProperty)) = \emptyset$;*

(CA) $I^O \models_O$ *ClassAssertion*$(C\ a)$	$I^{IN}(a) \in I^C(C)$
(OPA) $I^O \models_O$ *ObjectPropertyAssertion*$(op\ a_1\ a_2)$	$(I^{IN}(a_1), I^{IN}(a_2)) \in I^{OP}(I^P(op))$
(NOPA) $I^O \models_O$	
NegativeObjectPropertyAssertion$(op\ a_1\ a_2)$	$(I^{IN}(a_1), I^{IN}(a_2)) \notin I^{DP}(I^P(I^P(op)))$
(DPA) $I^O \models_O$ *DataPropertyAssertion*$(dp\ a\ lt)$	$(I^{IN}(a), I^{LIT}(lt)) \in I^{DP}(I^P(dp))$
(SI) $I^O \models_O$ *SameIndividual*$(a_1 \ldots a_n)$	$I^{IN}(a_j) = I^{IN}(a_k)$
	for each $1 \leq j \leq n$
	and each $1 \leq k \leq n$
(DI) $I^O \models_O$ *DifferentIndividuals*$(a_1 \ldots a_n)$	$I^{IN}(a_j) \neq I^{IN}(a_k)$
	for each $1 \leq j \leq n$
	and each $1 \leq k \leq n$
	such that $j \neq k$

Figure 5: Satisfiability of Instances Assertions

(OPD) $I^O \models_O$ *ObjectPropertyDomain*$(op\ C)$	$\forall x, y : (x, y) \in I^{OP}(I^P(op))$
	implies $x \in I^C(C)$
(OPR) $I^O \models_O$ *ObjectPropertyRange*$(op\ C)$	$\forall x, y : (x, y) \in I^{OP}(I^P(op))$
	implies $y \in I^C(C)$
(DPD) $I^O \models_O$ *DataPropertyDomain*$(dp\ C)$	$\forall x, y : (x, y) \in I^{DP}(I^P(dp))$
	implies $x \in I^C(C)$
(DPR) $I^O \models_O$ *DataPropertyRange*$(dp\ C)$	$\forall x, y : (x, y) \in I^{DP}(I^P(dp))$
	implies $y \in I^{DP}(C)$

Figure 6: Satisfiability of Type Assertions

$I^{DP}(dp) \subseteq \Delta \times \Gamma$ *if* $dp \in \Sigma_{DP}$, $I^{DP}(I^P(owl : topDataProperty)) = \Delta \times \Gamma$ *and* $I^{DP}(I^P(owl : bottomDataProperty)) = \emptyset$; $I^{IN}(a) \in \Delta$ *if* $a \in VI$; $I^{LIT}(v) \in I^{DT}(D)$ *if* $v \in D$, $D \in DT$; $I^{DT}(D) \subseteq \Gamma$ *if* $D \in DT$ *and* $I^{DT}(D_j)$ *and* $I^{DT}(D_k)$ *are disjoint for each* $i \neq j$; *and finally,* $I^{FACET}(FA^D, v) \subseteq I^{DT}(D)$ *if* $FA^D \in FACET^D$, $FACET^D \in FACET$ *and* v *of* $I^{DT}(D)$.

Definition 4 (Model of an Ontology). *An interpretation* I^O *is a model of an ontology* O *whenever* $I^O \models_O \varphi_i$ *for every* $\varphi_i \in O$, *where the satisfiability relation of ontology assertions* ($\models_O$) *is defined as shown in Figures 5, 6 and 7, and the semantics of object/data property based definitions and object property definitions in* I^O *is defined as shown in Figure 8.*

Definition 5 (Ontology Consistency and Entailment). *An ontology is* consistent *whenever there exists at least one model. An ontology* entailment *relation* $\vdash_O$ *can be defined for ontologies of the same vocabulary as follows:* $O \vdash_O O'$ *if all the models of* O *are models of* O'.

(SC) $I^O \models_O SubClassOf(C_1\ C_2)$	$I^C(C_1) \subseteq I^C(C_2)$
(EC) $I^O \models_O EquivalentClasses(C_1 \ldots C_n)$	$I^C(C_j) = I^C(C_k)$ $for\ each\ 1 \leq j \leq n$ $and\ each\ 1 \leq k \leq n$
(DC) $I^O \models_O DisjointClasses(C_1 \ldots C_n)$	$I^C(C_j) \cap I^C(C_k) = \emptyset$ $for\ each\ 1 \leq j \leq n$ $and\ each\ 1 \leq k \leq n$ $such\ that\ j \neq k$
(DU) $I^O \models_O DisjointUnion(C\ C_1 \ldots C_n)$	$I^C(C) = I^C(C_1) \cup \ldots \cup I^C(C_n)$ $and\ I^C(C_j) \cap I^C(C_k) = \emptyset$ $for\ each\ 1 \leq j \leq n$ $and\ each\ 1 \leq k \leq n$ $such\ that\ j \neq k$
(SOP) $I^O \models_O SubObjectPropertyOf(op_1\ op_2)$	$I^{OP}(I^P(op_1)) \subseteq I^{OP}(I^P(op_2))$
(EOP) $I^O \models_O EquivalentObjectProperties(op_1 \ldots op_n)$	$I^{OP}(I^P(op_j)) = I^{OP}(I^P(op_k))$ $for\ each\ 1 \leq j \leq n$ $and\ each\ 1 \leq k \leq n$
(SDP) $I^O \models_O SubDataPropertyOf(dp_1\ dp_2)$	$I^{DP}(I^P(dp_1)) \subseteq I^{DP}(I^P(dp_2))$
(EDP) $I^O \models_O EquivalentDataProperties(dp_1 \ldots dp_n)$	$I^{DP}(I^P(dp_j)) = I^{DP}(I^P(dp_k))$ $for\ each\ 1 \leq j \leq n$ $and\ each\ 1 \leq k \leq n$
(DOP) $I^O \models_O DisjointObjectProperties(op_1 \ldots op_n)$	$I^{DP}(I^P(op_j)) \cap I^{DP}(I^P(op_k)) = \emptyset$ $for\ each\ 1 \leq j \leq n$ $and\ each\ 1 \leq k \leq n$ $such\ that\ j \neq k$
(INOP) $I^O \models_O InverseObjectProperties(op_1\ op_2)$	$I^{OP}(I^P(op_1)) =$ $\{(x,y) \mid (y,x) \in I^{OP}(I^P(op_2))\}$
(ROP) $I^O \models_O ReflexiveObjectProperty(op)$	$\forall x : x \in \Delta\ implies\ (x,x) \in I^{DP}(I^P(op))$
(IROP) $I^O \models_O IrreflexiveObjectProperty(op)$	$\forall x : x \in \Delta\ implies\ (x,x) \notin I^{DP}(I^P(op))$
(SOP) $I^O \models_O SymmetricObjectProperty(op)$	$\forall x,y : (x,y) \in I^{DP}(I^P(op))$ $implies\ (y,x) \in I^{DP}(I^P(op))$
(AOP) $I^O \models_O AsymmetricObjectProperty(op)$	$\forall x,y : (x,y) \in I^{DP}(I^P(op))$ $implies\ (y,x) \notin I^{DP}(I^P(op))$
(TOP) $I^O \models_O TransitiveObjectProperty(op)\ if$	$\forall x,y,z : (x,y) \in I^{DP}(I^P(op))$ $and\ (y,z) \in I^{DP}(I^P(op))$ $imply\ (x,z) \in I^{DP}(I^P(op))$
(OPC) $I^O \models_O$ $SubObjectPropertyOf(ObjPropChain(op_1 \ldots op_n)\ op)$	$\forall y_0, \ldots, y_n : (y_0,y_1) \in I^{DP}(I^P(op_1))$ $and\ \ldots\ (y_{n-1},y_n) \in I^{DP}(I^P(op_n))$ $imply\ (y_0,y_n) \in I^{DP}(I^P(op))$
(FP) $I^O \models_O FunctionalObjectProperty(op)$	$\forall x,y_1,y_2 : (x,y_1) \in I^{DP}(I^P(op))$ $and\ (x,y_2) \in I^{DP}(I^P(op))$ $imply\ y_1 = y_2$
(IFP) $I^O \models_O InverseFunctionalObjectProperty(op)$	$\forall x_1,x_2,y : (x_1,y) \in I^{DP}(I^P(op))$ $and\ (x_2,y) \in I^{DP}(I^P(op))$ $imply\ x_1 = x_2$

Figure 7: Satisfiability of Class and Property Restrictions

Given an ontology O we consider as usual the *terminological box* of the ontology O, denoted by *TBox(O)*, and the *assertion box*, denoted by *ABox(O)*, where *ABox(O)* is the conjunction of statements in O of Figure 5, and

(OINT) ObjectIntersectionOf(C_1 ... C_n)	$= I^C(C_1) \cap ... \cap I^C(C_n)$
(OU) ObjectUnionOf(C_1 ... C_n)	$= I^C(C_1) \cup ... \cup I^C(C_n)$
(OC) ObjectComplementOf(C)	$= \Delta \backslash I^C(C)$
(OO) ObjectOneOf(a_1 ... a_n)	$= \{I^{IN}(a_1), ..., I^{IN}(a_n)\}$
(OSV) ObjectSomeValuesFrom(op C)	$= \{ x \mid \exists y : (x,y) \in I^{OP}(I^P(op))$ $\mathit{and}\ y \in I^C(C)\}$
(OAV) ObjectAllValuesFrom(op C)	$= \{ x \mid \forall y : (x,y) \in I^{OP}(I^P(op))$ $\mathit{implies}\ y \in I^C(C)\}$
(OHV) ObjectHasValue(op a)	$= \{ x \mid (x, I^{IN}(a)) \in I^{OP}(I^P(op))\}$
(OHS) ObjectHasSelf(op)	$= \{ x \mid (x,x) \in I^{OP}(I^P(op))\}$
(OMi1) ObjectMinCardinality(n op)	$= \{ x \mid \#\{ y \mid (x,y) \in I^{DP}(I^P(op))\} \geq n\}$
(OMa1) ObjectMaxCardinality(n op)	$= \{ x \mid \#\{ y \mid (x,y) \in I^{DP}(I^P(op))\} \leq n\}$
(OE1) ObjectExactCardinality(n op)	$= \{ x \mid \#\{ y \mid (x,y) \in I^{DP}(I^P(op))\} = n\}$
(OMi2) ObjectMinCardinality(n op C)	$= \{ x \mid \#\{ y \mid (x,y) \in I^{OP}(I^P(op))$ $\mathit{and}\ y \in I^C(C)\} \geq n\}$
(OMa2) ObjectMaxCardinality(n op C)	$= \{x \mid \#\{ y \mid (x,y) \in I^{DP}(I^P(op))$ $\mathit{and}\ y \in I^C(C)\} \leq n\}$
(OE2) ObjectExactCardinality(n op C)	$= \{x \mid \#\{ y \mid (x,y) \in I^{DP}(I^P(op))$ $\mathit{and}\ y \in I^C(C)\} = n\}$
(OINV) ObjectInverseOf(op)	$= \{ (x,y) \mid (y,x) \in I^{OP}(I^P(op))\}$
(DR) DatatypeRestriction(D $FA^D_1(lt_1)$ ... $FA^D_n(lt_n)$)	$= \{v \in D \mid v \in I^{FACET}(FA^D_1, lt_1)$ $\ldots, v \in I^{FACET}(FA^D_n, lt_n)\}$
(DINT) DataIntersectionOf(DR_1,..,DR_n)	$= I^{DT}(DR_1) \cap ... \cap I^{DT}(DR_n)$
(DSV) DataSomeValuesFrom(dp DR)	$= \{ x \mid \exists y_1, ..., y_n : (x, y_k) \in I^{DP}(I^P(dp))$ $\mathit{and}\ y_k \in I^C(DR)$ $\mathit{for\ each}\ 1 \leq k \leq n\}$
(DAV) DataAllValuesFrom(dp DR)	$= \{ x \mid \forall y_1, ..., y_n : (x, y_k) \in I^{DP}(I^P(dp))$ $\mathit{imply}\ y_k \in I^C(DR)$ $\mathit{for\ each}\ 1 \leq k \leq n\}$
(DHV) DataHasValue(dp lt)	$= \{ x \mid (x, lt) \in I^{DP}(I^P(dp))\}$

Figure 8: Semantics of Object/Data Property based Definitions and Object Property Definitions

TBox(O) is the conjunction of statements in O of Figures 6 and 7. We denote by $TBox^R(O)$ the class and property restrictions of Figure 7, and by $TBox^T(O)$ the type assertions of Figure 6.

The ontology assertions φ can be represented as (conjunctions of) triples $\varphi = \bigwedge t$ where t is an object/data property assertion, using RDF(S) and OWL vocabularies, and blank nodes[8]. The vocabulary VO can be extended to a vocabulary VO_0 including blank nodes and RDF(S) and OWL vocabularies, and an interpretation I^O can be extended to an interpretation I^O_0 such that $I^O_0 \models_O t$ for every t iff $I^O \models_O \varphi$. We omit here the details. From now on, we assume to work with extended vocabularies, ontologies and inter-

[8]See `https://www.w3.org/TR/owl2-mapping-to-rdf/` for more details.

pretations. Now, we will define the notion of query, constraint systems and query answers.

Definition 6 (Vocabulary of a Query). *A vocabulary of a query is a triple of the form $VQ = (VO, ORDER, FUNC)$, where VO is the extended vocabulary of an ontology; $ORDER$ is a set of partial order symbols $\{\leq^D | D \in DT\}$; and $FUNC$ is a set of function symbols $f^n : D_1 \times \cdots \times D_n \rightarrow D_{n+1}$, where n is the arity of f, $D_1, \ldots, D_n \in DT$ and $D_{n+1} \in DT$.*

A query vocabulary includes the extended ontology vocabulary for describing triple patterns and it introduces two new elements: a partial order in each datatype, and a set of function symbols acting on datatypes, to describe filter conditions in queries.

Definition 7 (Query). *A query Q of a vocabulary VQ has the form*

$$\text{SELECT } V \text{ WHERE } W$$

where V is a subset of the variables of W, and W is (1) a conjunction of triple patterns *subject-property-object (s p o), where s, p and o can be elements $Res(VO) \cup Lit(VO)$ as well as variables of the form $?x$, and (2) a (possibly empty)* conjunction of filter conditions *$l \diamond r$, where $\diamond$ can be one of* $\{=, !=, >, <, <=, >=\}$*, and l and r are expressions built from $FUNC$, $Lit(VO)$ and variables.*

Given a query Q we denote: (1) the conjunction of triple patterns by *TP(Q)*; (2) the conjunction of filter conditions by *FC(Q)*; (3) the set of variables and literals occurring in literal positions of *TP(Q)* by *Lit(Q)*; (4) the set of variables and resources occurring in (non-property) resource positions of *TP(Q)* by *Res(Q)*; (5) the set of variables and resources occurring in property positions of *TP(Q)* by *Prop(Q)*; (6) the set of variables of a query Q by $Var(Q)$.

Given a triple pattern $tp = (s\ p\ o)$ of a query Q in a vocabulary VQ, the *resource positions* of tp are the set $\{s, p, o\}$ whenever $p \in VOP$, and the set $\{s, p\}$ whenever $p \in VDP$. The *literal positions* of $tp = (s\ p\ o)$ are the set $\emptyset$ whenever $p \in VOP$, and the set $\{o\}$ whenever $p \in VDP$. Finally, the *property positions* of a triple pattern $tp = (s\ p\ o)$ are the set $\{p\}$.

Definition 8 (Interpretation of a Query). *An interpretation I^Q of a query Q in a vocabulary VQ is a 3-tuple of the form $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ where I^O is an extension of an interpretation of VO; I^{ORDER} is a partial order interpretation such that $I^{ORDER}(\leq^D)$ is a partial order on $I^{DT}(D)$, for each $D \in DT$; and finally, I^{FUNC} is the function symbol interpretation such that $I^{FUNC}(f^n)(t_1, \ldots, t_n) \in I^{DT}(D_{n+1})$ whenever $f^n : D_1 \times \cdots \times D_n \rightarrow D_{n+1} \in FUNC$, for every $t_i \in I^{DT}(D_i)$.*

Therefore, interpretations of queries provide semantics to the extended vocabulary of an ontology, the partial orders on datatypes, as well as function symbols as mappings from the defined domains to the defined ranges. Now, we define valuations of a query into an interpretation in such a way that variables occurring in resource (respectively literal) positions are mapped into the object (respectively, data) domain, and variables occurring in property positions are mapped into object and data property domains.

Definition 9 (Valuation). *A valuation θ of a query Q into an interpretation I^Q is a mapping from $Var(Q)$ into $\Delta \cup \Gamma$ such that $\theta(?x) \in \Delta$ whenever $?x \in Res(Q)$; $\theta(?x) \in \Sigma_{OP} \cup \Sigma_{DP}$ whenever $?x \in Prop(Q)$; and $\theta(?x) \in \Gamma$ whenever $?x \in Lit(Q)$.*

Given a valuation θ and a triple pattern $tp = (s\ p\ o)$ of a query, we denote by $\theta(tp)$ the replacement of variables of tp according to the substitution θ. Abusing from the notation, $\theta(tp)$ is interpreted as the object property assertion $ObjectPropertyAssertion(\theta(p)\ \theta(s)\ \theta(o))$ whenever $\theta(p) \in \Sigma_{OP}$, and the data property assertion $DataPropertyAssertion(\theta(p)\ \theta(s)\ \theta(o))$ whenever $\theta(p) \in \Sigma_{DP}$.

Now, we define the concept of constraint system in which variables and literals can be related by equality and inequality constraints (similarly to filter conditions) as well as by facet restrictions taken from ontology vocabularies. Constraint systems share vocabularies with queries.

Definition 10 (Constraint System). *A constraint system C in a query vocabulary VQ is a* conjunction of *(1)* equality and inequality constraints *of the form $l \bowtie r$, where $\bowtie$ can be one of $\{=, != , >, <, <=, >=\}$, and l, r are expressions built from $FUNC$, $Lit(VO)$ and variables, as well as (2)* membership relationships $e \in FA^D(e')$ *where e, e' are variables or literals of $Lit(VO)$. We denote by var(C) the set of variables of a constraint system C.*

In the general case constraint systems include equality and inequality constraints and membership relations. In the particular case of ontology facets for integers and real numbers, membership relations (*minInclusive*, *maxInclusive*, *minExclusive*, *maxExclusive*) become also equality and inequality constraints.

Definition 11 (Satisfiability of a Constraint System). *An interpretation of a query I^Q satisfies a constraint system C for a valuation θ, whenever $(I^Q, \theta) \models_C l \bowtie r$ for every $l \bowtie r \in C$ and $[\![e]\!]_\theta^{I^Q} \in I^{FACET}(FA^D, [\![e']\!]_\theta^{I^Q})$ for every $e \in FA^D(e')$ of C, where $(I^Q, \theta) \models_C l \bowtie r$ holds whenever $[\![l]\!]_\theta^{I^Q}, [\![l]\!]_\theta^{I^Q} \in D$ and $[\![l]\!]_\theta^{I^Q} \bowtie_D^{I^Q} [\![r]\!]_\theta^{I^Q}$.*

$[\![e]\!]_\theta^{I^Q}$ is defined as $\theta(e)$ whenever $e \in var(C)$; $I^{LIT}(e)$ whenever $e \in \Gamma$; and $I^{FUNC}(f^n)([\![e_1]\!]_\theta^{I^Q}, \ldots, [\![e_1]\!]_\theta^{I^Q})$ whenever $e = f^n(e_1, \ldots, e_n)$, and $f^n \in FUNC$; and $\bowtie_D^{I^Q}$ is defined as $e \ =_D^{I^Q} \ e'$ whenever e and e' are equals in D; $e \ != _D^{I^Q} \ e'$ whenever e and e' are distinct in D; $e \ <=_D^{I^Q} \ e'$ whenever $e \ I^{ORDER}(\leq^D) \ e'$; $e \ >=_D^{I^Q} \ e'$ whenever $e' \ I^{ORDER}(\leq^D) \ e$; $e \ <_D^{I^Q} \ e'$ whenever $e \ <=_D^{I^Q} \ e'$ and $e \ !=_D^{I^Q} \ e'$; and, finally, $e \ >_D^{I^Q} \ e'$ whenever $e \ >=_D^{I^Q} \ e'$ and $e \ !=_D^{I^Q} \ e'$.

Definition 12 (Constraint System Consistency and Entailment). *A constraint system C is* consistent *if there exists at least one interpretation I^Q satisfying C for some valuation θ. An* entailment *relation $\vdash_C$ can be defined for constraint systems of the same vocabulary as follow: $C \vdash_C C'$, where $var(C') \subseteq var(C)$, whenever every interpretation and valuation holding C satisfies C'.*

As filter conditions are a particular case of constraint systems, answers of queries can be defined as follows:

Definition 13 (Query Answer). *An* answer *of a query*

$$Q = \textsf{SELECT } V \textsf{ WHERE } W$$

in an interpretation I^Q is the set of values for the variables of V given by θ to W such that $I^O \models_O \theta(tp)$, for each $tp \in TP(Q)$, and $(I^Q, \theta) \models_C FC(Q)$.

We denote the set of answers of a query Q in all the models of O by $ANS(Q, O)$.

5. Correct Queries and Type Validity

Now, we define the concepts of correct query and valid type for a query, as well as incomplete and faulty queries, and counterexamples of a variable typing.

Definition 14 (Correct Query). *A query Q is* correct *in an ontology O whenever there exists an interpretation I^Q of the vocabulary of Q in which Q is correct in O for some valuation θ. A query Q is* correct *in an ontology O, an interpretation $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ and a valuation θ whenever I^O is a model of $TBox(O)$, $I^O \models_O \theta(tp)$, for each $tp \in TP(Q)$, and $(I^Q, \theta) \models_C FC(Q)$.*

Thus, for a correct query there exists at least one valuation in which triple patterns and filter conditions become true in a *model of the terminological box*. Next, we define the concept of satisfiable query, which is a weaker concept than correctness.

Definition 15 (Satisfiable Query). *A query Q is* satisfiable *in an ontology O whenever there exist an interpretation I^Q of the vocabulary of Q and a valuation θ in which Q is satisfiable. A query Q is* satisfiable *in an ontology O, an interpretation $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ and a valuation θ whenever I^O is a model of $TBox^R(O)$, $I^O \models_O \theta(tp)$, for each $tp \in TP(Q)$, and $(I^Q, \theta) \models_C FC(Q)$.*

For a satisfiable query there exists at least one valuation in which triple patterns and filter conditions become true in some *model of the class and property restrictions of the terminological box*. Finally, we define the concept of ill typed query.

Definition 16 (Ill Typed Query). *A query is* ill typed *in an ontology O whenever the query is satisfiable in O but not correct in O.*

Thus, a query is ill typed in an ontology whenever there exists at least one type assertion which is violated for all the models of satisfying triple patterns and filter conditions and the class and property restrictions of the terminological box, and well typed, otherwise.
The following proposition proves that incorrect queries have empty answers.

Proposition 1. *If Q is not correct in O then ANS(Q,O) $= \emptyset$.*
Proof*:*
Let us suppose that ANS(Q,O)$\neq \emptyset$. Let I^Q be an interpretation and θ be a valuation such that I^O is a model O, $I^O \models_O \theta(tp)$, for each $tp \in TP(Q)$, and $(I^Q, \theta) \models_C FC(Q)$. Thus, trivially I^O is a model of $TBox(O)$ and therefore Q is correct.

Now, we can define the notion of valid type for a variable in a query.

Definition 17 (Type Validity). *T is a* valid *type for $?x \in Res(Q)$ in a query Q and an ontology O whenever T is* valid *for $?x$ in Q for every interpretation $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ of the vocabulary of Q and valuation θ such that I^O is model of $TBox(O)$, $I^O \models_O \theta(tp)$ for every $tp \in TP(Q)$ and $(I^Q, \theta) \models_C FC(Q)$.*

T is valid *for $?x$ in Q in an interpretation $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ and a valuation θ whenever $I^O \models_O ClassAssertion(T, \theta(?x))$.*

The concept of valid type only focuses in variables occurring in resource positions (but not in property positions). A type is valid for a variable whenever it is true in all the models of the terminological box and all the valuations satisfying the triple patterns and filter conditions.

Finally, we can define the concepts of *incomplete* query Q with regard to $?x : T$ and O, *faulty* query with regard to $?x : T$ and O, and a *counterexample* θ of $?x : T$ with regard to Q and O.

Definition 18 (Incomplete and Faulty Query and Counterexample).

- *A query Q is* incomplete *with regard to a variable typing $?x : T$ and an ontology O whenever there exists an interpretation $I^Q = (I^O$, I^{ORDER}, $I^{FUNC})$ of the vocabulary of Q and valuation θ such that:*
  - *I^O is model of $TBox(O)$;*
  - *$I^O \models_O \theta(tp)$ for every $tp \in TP(Q)$, $(I^Q, \theta) \models_C FC(Q)$; and*
  - *$I^O \not\models_O ClassAssertion(T\ \theta(?x))$.*
- *A query Q is* faulty *with regard to a variable typing $?x : T$ and an ontology O, whenever there do not exist an interpretation $I^Q = (I^O$, I^{ORDER}, $I^{FUNC})$ of the vocabulary of Q and valuation θ such that:*

  - *I^O is model of $TBox(O)$;*
  - *$I^O \models_O \theta(tp)$ for every $tp \in TP(Q)$, $(I^Q, \theta) \models_C FC(Q)$; and*
  - *$I^O \models_O ClassAssertion(T\ \theta(?x))$.*

- *θ is a counterexample of $?x : T$ in Q in an ontology O whenever there exists an interpretation $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ of the vocabulary of Q such that:*
  - *I^O is model of $TBox(O)$;*
  - *$I^O \models_O \theta(tp)$ for every $tp \in TP(Q)$; and*
  - *$(I^Q, \theta) \not\models_C FC(Q)$.*

Thus, an incomplete query is a query in which $?x : T$ cannot be proved for some interpretation and valuation. A query is faulty whenever any interpretation satisfies $?x : T$. Finally, θ is counterexample whenever the filter condition cannot be ensured for some interpretation.

6. Method for Correctness

Now, we will describe the method for correctness and prove correctness and completeness results of the method.

Firstly, we have to consider the so-called *constrained ontologies*, which are pairs (O^V, C), where O^V is an ontology with variables possibly occurring in any position of the assertions (V is the set of variables of O), and C is a constraint system, where $var(C) \subseteq V$. Given a valuation θ, the ontology $\theta(O^V)$ is an ontology according to Definition 2, in which variables of O^V are replaced by their values in θ.

Definition 19 (Satisfiability of a Constrained Ontology). *An interpretation $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ of a query Q satisfies a constrained ontology (O^V, C) in a valuation θ whenever I^O is a model of $\theta(O^V)$ and $(I^Q, \theta) \models_C C$.*

Definition 20 (Constrained Ontology Consistency and Entailment). *A constrained ontology (O^V, C) is* consistent *whenever there exists an interpretation I^Q satisfying (O^V, C) for some valuation θ. An* entailment *relation $\vdash_{CO}$ can be also defined for constrained ontologies of the same vocabulary: $(O^V, C) \vdash_{CO} (O'^{V'}, C')$ whenever $var(V') \subseteq var(V)$, and all the interpretations I^Q satisfying (O^V, C) in θ also satisfy $(O'^{V'}, C')$.*

$$\begin{array}{lcl}
O_Q^{Var(Q)} & = & TBox(O) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ p \in VOP\}} ObjectPropertyAssertion(p\ s\ o) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ p \in VDP\}} DataPropertyAssertion(p\ s\ o) \\
 & & \\
C_Q & = & FC(Q)
\end{array}$$

Figure 9: Correctness. Version 1.

$$\begin{array}{lcl}
\bar{O}_Q^{Res(Q)} & = & TBox(O) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ p \in VOP\}} ObjectPropertyAssertion(p\ s\ o) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ p \in VDP,\ o \notin Var(Q)\}} DataPropertyAssertion(p\ s\ o) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ p \in VDP,\ o \in Var(Q)\}} ClassAssertion(owl:Thing\ s) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ DataPropertyDomain(p\ D) \in TBox(O),\ o \in Var(Q)\}} ClassAssertion(D\ s) \\
 & & \\
\bar{C}_Q & = & FC(Q) \\
 & & \bigwedge_{\substack{\{(s\ p\ o) \in TP(Q), \\ o \in Var(Q), \\ \bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(DataAllValuesFrom(p\ DR)\ s), \\ DR = DatatypeRestriction(D\ FA_1^D(lt_1)\ \dots\ FA_n^D(lt_n))\}}} o \in FA_i^D(lt_i) \\
 & & \bigwedge_{\substack{\{(s\ p\ o) \in TP(Q), \\ o \in Var(Q), \\ \bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(DataHasValue(p\ lt)\ s)\}}} o = lt
\end{array}$$

Figure 10: Correctness. Version 2.

In order to check the correctness of a query Q the constrained ontology $(O_Q^{Var(Q)}, C_Q)$ defined in Figure 9 is considered.

Proposition 2. *$(O_Q^{Var(Q)}, C_Q)$ is consistent iff Q is correct in O.*
Proof:
It can be deduced from $O_Q^{Var(Q)}$ includes $TBox(O)$ and ObjectPropertyAssertion(p s o) as well as DataPropertyAssertion(p s o) for each $(s\ p\ o) \in TP(Q)$, and C_Q is equal to $FC(Q)$.

Existing ontology reasoners cannot be directly used to prove the consistency of constrained ontologies. While variables occurring in resource positions can be transformed into blank nodes (i.e., individuals), by considering _:x for each variable ?x, it does not work in the case of variables in literal positions. In order to reason with constrained ontologies, we have to transform $(O_Q^{Var(Q)}, C_Q)$ into a pair $\bar{O}_Q^{Res(Q)}$, $\bar{C}_Q$ in which $\bar{O}_Q^{Res(Q)}$ is an ontology *without variables in literal positions*, and $\bar{C}_Q$ is a constraint system including constraints on variables in literal positions. The transformation is shown in Figure 10.

Basically, the transformation procedure is as follows:

(1) Removes $DataPropertyAssertion(p\ s\ o)$ from $O_Q^{Var(Q)}$ whenever $o \in Var(Q)$

(2) Adds $ClassAssertion(D\ s)$[9], for each domain of p (and also $ClassAssertion(owl:Thing\ s)$) to $\bar{O}_Q^{Res(Q)}$, and

(3) Adds membership constraints to the corresponding data property based definitions to $\bar{C}_Q$.

Therefore the set of variables of $\bar{O}_Q^{Res(Q)}$ and $\bar{C}_Q$ is $Res(Q)$ and $Lit(Q)$, respectively. Let us remark that in Figure 10 the entailment relationship $\vdash_O$ is used from the $\bar{O}_Q^{Res(Q)}$ to collect all the class assertions of s of the form $DataAllValuesFrom(p\ DR)$ and $DataHasValue(p\ lt)$ whenever $(s\ p\ o) \in TP(Q)$ and $o \in Var(Q)$.

As $(O_Q^{Var(Q)}, C_Q)$ and the pair $\bar{O}_Q^{Res(Q)}$ and $\bar{C}_Q$ can differ from $DataPropertyAssertion(p\ s\ o)$ assertions, then models of both ones do not necessarily coincide. However, a model of $(O_Q^{Var(Q)}, C_Q)$ can be rebuilt from a model of the pair $\bar{O}_Q^{Res(Q)}$ and $\bar{C}_Q$ as follows.

Let $I^Q = (I^O, I^{ORDER}, I^{FACET})$ be such that I^O is a model of $\theta(\bar{O}_Q^{Res(Q)})$ and $(I^Q, \theta) \models_C \bar{C}_Q$. We denote by $DP(I^O)$ the *extension of the interpretation* I^O built as follows:

- For each $q \in VDP$, such that $(s\ p\ o) \in TP(Q)$, $o \in Var(Q)$ and $TBox(O) \vdash_O SubPropertyOf(p\ q)$ then $(\theta(s), \theta(o)) \in I^{DP}(I^P(q))$ in $DP(I^O)$.
- I^O and $DP(I^O)$ coincide in the rest of interpretations of VDP.

The following theorem proves that $DP(I^O)$ is a model of $(O_Q^{Var(Q)}, C_Q)$ if I^O is model of $\bar{O}_Q^{Res(Q)}$ and satisfies $\bar{C}_Q$. Also each model of $(O_Q^{Var(Q)}, C_Q)$ is a model of $\bar{O}_Q^{Res(Q)}$ and satisfies $\bar{C}_Q$. However, it requires that Q satisfies the following condition. This condition is mandatory for completeness since it ensures that $\bar{C}_Q$ is a conjunction of constraints.

[9]Let us remark that $ClassAssertion(D\ s)$ is equivalent to $ObjectPropertyAssertion(rdf:type\ s\ D)$ according to the triple based representation of ontologies.

Definition 21 (Conjunctive Query). *A query Q is a* conjunctive query *in O whenever for every $(s\ p\ o) \in TP(Q)$, such that $p \in VPD$, $o \in Var(Q)$ and every data property based definition D such that $\bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(D\ s)$ then:*

(a) either $TBox(O) \vdash_O EquivalentClasses(D\ DataAllValuesFrom(p\ DR))$ where DR is a datatype restriction;

(b) or $TBox(O) \vdash_O EquivalentClasses(D\ DataHasValue(p\ lt))$.

Example 12. *Non-conjunctive queries can require a disjunction of constraints. For instance, let us suppose the query:*

```
SELECT ?X ?Y
WHERE {
  ?X rdf:type sn:Influencer .
  ?X sn:dailyLikes ?Y .
  ?X sn:dailyLikes ?Z
  }
```

Here Q is a non conjunctive query since $\bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(Data$-$SomeValuesFrom(dailyLikes\ \ DatatypeRestriction(Integer\quad >= (100)))$ $?X)$. Then the resulting constraint system is $?Y >= 100 \bigvee ?Z >= 100$.

Theorem 1 (Correctness Method). *If Q is a conjunctive query in O then:*

(a) If $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ satisfies $(O_Q^{Var(Q)}, C_Q)$ in a valuation θ then I^O is a model of $\theta(\bar{O}_Q^{Res(Q)})$ and $(I^Q, \theta) \models \bar{C}_Q$.

(b) If $I^Q = (I^O, I^{ORDER}, I^{FUNC})$ is model of $\theta(\bar{O}_Q^{Res(Q)})$ and $(I^Q, \theta) \models \bar{C}_Q$ then $DP(I^Q) = (DP(I^O), I^{ORDER}, I^{FUNC})$ satisfies $(O_Q^{Var(Q)}, C_Q)$ in θ.

Proof*:*

(a) We have to prove that: (a.1) $ClassAssertion(owl : Thing\ s)$ is satisfied by I^O and θ; (a.2) $ClassAssertion(D\ s)$ is satisfied by I^O and θ whenever $(s\ p\ o) \in TP(Q)$, $p \in VDP$, $o \in Var(Q)$ and DataProper-tyDomain$(D\ p) \in TBox(O)$; and (a.3) $(I^Q, \theta) \models_C \bar{C}_Q$.

(a.1) Since $I^C(owl:Thing) = \Delta$ *then* $I^O \models_O \theta(ClassAssertion(owl:Thing\ s))$.

(a.2) Since $TBox(O)$ *is included in* $O_Q^{Var(Q)}$ *then* $DataPropertyDomain(D\ p) \in O_Q^{Var(Q)}$. *By construction of* $DP(I^O)$ *for each* $p \in VDP$ *such that* $(s\ p\ o) \in TP(Q)$ *and* $o \in Var(Q)$ *then* $(\theta(s), \theta(o)) \in I^{DP}(I^P(p))$. *Thus,* $\theta(s) \in I^C(D)$.

(a.3) From (1) and (2), I^O *is a model of* $\theta(\bar{O}_Q^{Res(Q)})$. *We have two cases:*

(a.3.1) $(s\ p\ o) \in TP(Q)$, $\bar{O}^{Res(Q)} \vdash_O ClassAssertion(DataAllValuesFrom(p\ DR)\ s)$, $DR = DatatypeRestriction(D\ FA_1^D(lt_1) \ldots FA_n^D(lt_n))$. *Since* I^O *is model of* $\theta(\bar{O}_Q^{Res(Q)})$ *then* $I^O \models_O ClassAssertion(DataAllValuesFrom(p\ DR)\ \theta(s))$ *and thus* $\theta(o) \in I^{FACET}(FA_i^D, lt_i)$.

(a.3.2) $(s\ p\ o) \in TP(Q)$, $\bar{O}^{Res(Q)} \vdash_O ClassAssertion(DataHasValue(p\ lt)\ s)$. *Since* I^O *is model of* $\theta(\bar{O}_Q^{Res(Q)})$ *then* $I^O \models_O ClassAssertion(DataHasValue(p\ lt)\ \theta(s))$ *and thus* $\theta(o) = lt$.

Thus in both cases $(I^Q, \theta) \models_C \bar{C}_Q$.

(b) We have to prove that $DP(I^Q)$ *is a model of* $\theta(O_Q^{Var(Q)}, C_Q)$. *By construction of* $DP(I^O)$, $(\theta(s), \theta(o)) \in I^{DP}(I^P(p))$ *in* $DP(I^O)$, *whenever* I^O *is a model of* $\theta(\bar{O}_Q^{Res(Q)})$ *and* $(I^Q, \theta) \models_C \bar{C}_Q$. *Therefore* $DP(I^O) \models_O DataPropertyAssertion(p\ \theta(s)\ \theta(o))$. *However, we have to check that* $DP(I^O)$ *does not contradict the axioms of* $TBox(O)$. *We need to consider the following cases:*

(b.1) Let us suppose the case of the class restrictions (SC) SubClassOf, (EC) EquivalentClasses, (DC) DisjointClasses and (DU) DisjointUnion of the $TBox(O)$ *involving* D. *In this case, they are not violated since* $ClassAssertion(D\ s) \in \bar{O}_Q^{Res(Q)}$.

(b.2) Let us suppose the case of (DPD) DataPropertyDomain. If $DataPropertyDomain(D\ p) \in TBox(O)$ *then* $\bar{O}_Q^{Res(Q)}$ *contains ClassAssertion* $(D\ s)$, *therefore* $\theta(s) \in I^C(D)$. *Let us analyze the axioms involving* D:

(b.2.1) Axioms (OINT) ObjectIntersectionOf, (OU) ObjectUnionOf, (OC) ObjectComplementOf and (OO) ObjectOneOf are not violated trivially.

(b.2.2) Since I^O and $DP(I^O)$ coincide in object properties then the axioms (OSV) ObjectSomeValuesFrom, (OAV) ObjectAllValuesFrom, (OHV) ObjectHasValue, (OHS) ObjectHasSelf, (OMi1) / (OMi2) ObjectMinCardinality, (OMa1) / (OMa2) ObjectMaxCardinality, and (OE1)/(OE2) ObjectExactCardinality are not violated trivially.

(b.2.3) In the case of the axiom (DINT) DataIntersectionOf, if DataPropertyDomain(D p) $\in TBox(O)$, and $D = DataIntersectionOf(D_1, \ldots, D_n)$ then $ClassAssertion(D\ s) \in \bar{O}_Q^{Res(Q)}$, and thus $\bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(D_i\ s)$. Since the query is conjunctive then either $TBox(O) \vdash_O EquivalentClasses(D_i\ DataAllValuesFrom(p\ DR))$, where DR is a datatype restriction; or $TBox(O) \vdash_O EquivalentClasses(D_i\ DataHasValue(p\ lt))$. By construction of $DP(I^O)$, $(\theta(s),\ \theta(o)) \in I^{DP}(I^P(p))$ in $DP(I^O)$ whenever $(I^Q, \theta) \models_C \bar{C}_Q$. If $DataIntersectionOf\ (D_1,\ \ldots,\ D_n)$ is violated, then there exists $\theta(s)$ such that $\theta(s) \notin I^C(D_i)$, for every i, but it contradicts $(I^Q, \theta) \models_C \bar{C}_Q$.

(b.2.4) In the case of the axiom (DSV) DataSomeValuesFrom, if DataPropertyDomain(D p) $\in TBox(O)$, $D = DataSomeValuesFrom(p\ E)$ then $ClassAssertion(D\ s) \in \bar{O}_Q^{Res(Q)}$, and thus $\bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(D\ s)$. Since the query is conjunctive then: either $TBox(O) \vdash_O EquivalentClasses(D\ DataAllValuesFrom\ (p\ DR))$, where $DR = DatatypeRestriction\ (D\ FA_1^D(lt_1)\ ...\ FA_n^D(lt_n))$ or $TBox(O) \vdash_O EquivalentClasses(D\ DataHasValue(p\ lt))$. By construction of $DP(I^O)$, $(\theta(s),\ \theta(o)) \in I^{DP}(I^P(p))$ whenever $(I^Q, \theta) \models_C \bar{C}_Q$. If $\theta(s) \notin I^C(D)$ then it contradicts $(I^Q, \theta) \models_C \bar{C}_Q$.

(b.2.5) The case of axioms (DAV) DataAllValuesFrom and (DHV) DataHasValue is similar to the case of the axiom (DSV) DataSomeValuesFrom.

(b.3) The axiom (DPR) DataPropertyRange is not violated by construction of $DP(I^O)$.

(b.4) In the case of the axiom (SDP) SubDataPropertyOf, by construction of $DP(I^O)$, for each $q \in VDP$, such that $(s\ p\ o) \in TP(Q)$, $o \in Var(Q)$ and $TBox(O) \vdash_O SubPropertyOf(p\ q)$ then $(\theta(s),$

$\theta(o)) \in I^{DP}(I^P(q))$.

(b.5) The case of the axiom (EDP) EquivalentDataProperties is consequence of the previous one.

Corollary 1. *If Q is a conjunctive query in O then $(O_Q^{Var(Q)}, C_Q)$ is consistent iff $\bar{O}_Q^{Res(Q)}$ and $\bar{C}_Q$ are consistent.*

7. Method for Type Validity

Now, we will describe the method for type validity and prove correctness and completeness results.

The method for type validity is similar to the proposed for correctness, but the entailment relation is used instead of consistency relation. More concretely, $(O_Q^{Var(Q)}, C_Q) \vdash_{CO} (ClassAssertion(T\ ?x), \emptyset)$ should be proved.

Proposition 3. *$(O_Q^{Var(Q)}, C_Q) \vdash_{CO} (ClassAssertion(T\ ?x), \emptyset)$ iff T is valid for $?x$ in Q and O.*

Proof:

$(\Rightarrow)$:

Let us suppose I^O model of $TBox(O)$ such that $I^O \models_O \theta(tp)$, for each $tp \in TP(Q)$, and $(I^Q, \theta) \models_C FC(Q)$ for some θ. Since $O_Q^{Var(Q)}$ contains $TBox(O)$, ObjectPropertyAssertion(p s o), DataPropertyAssertion(p s o) for each $(s\ p\ o) \in TP(Q)$, and C_Q contains $FC(Q)$ then, by hypothesis, $I^O \models_O ClassAssertion(T\ \theta(?x))$.

$(\Leftarrow)$:

Let us suppose that T is valid for $?x$ in Q and O, and I^Q model of $(O_Q^{Var(Q)}, C_Q)$. Since $O_Q^{Var(Q)}$ contains $TBox(O)$, ObjectPropertyAssertion(p s o), DataPropertyAssertion(p s o) for each $(s\ p\ o) \in TP(Q)$, and C_Q contains $FC(Q)$, then, by hypothesis, $I^O \models_O ClassAssertion(T\ \theta(?x))$ and, thus, $(O_Q^{Var(Q)}, C_Q) \vdash_{CO} (ClassAssertion(T\ ?x), \emptyset)$.

As in the case of correctness, and given that we are using existing reasoners, instead of proving $(O_Q^{Var(Q)}, C_Q) \vdash_{CO} (ClassAssertion(T\ ?x), \emptyset)$, the following entailments are proved:

- $\bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(T\ ?x)$ and
- $\bar{C}_Q \vdash_C C_{?x:T}$.

$$
\begin{array}{lcl}
\bar{O}_Q^{Res(Q)} & = & TBox(O) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ p \in VOP\}} ObjectPropertyAssertion(p\ s\ o) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ p \in VDP,\ o \notin Var(Q)\}} DataPropertyAssertion(p\ s\ o) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ p \in VDP,\ o \in Var(Q)\}} ClassAssertion(owl:Thing\ s) \\
 & & \bigwedge_{\{(s\ p\ o) \in TP(Q),\ DataPropertyDomain(p\ D) \in TBox(O),\ o \in Var(Q)\}} ClassAssertion(D\ s) \\
 & & \\
\bar{C}_Q & = & FC(Q) \\
 & & \bigwedge_{\substack{\{(s\ p\ o) \in TP(Q), \\ o \in Var(Q), \\ \bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(DataAllValuesFrom(p\ DR)\ s), \\ DR = DatatypeRestriction(D\ FA_1^D(lt_1)\ \ldots\ FA_n^D(lt_n))\}}} o \in FA_i^D(lt_i) \\
 & & \bigwedge_{\substack{\{(s\ p\ o) \in TP(Q), \\ o \in Var(Q), \\ \bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(DataHasValue(p\ lt)\ s)\}}} o = lt \\
C_{s:T} & = & \bigwedge_{\substack{\{(s\ p\ o) \in TP(Q), \\ o \in Var(Q), \\ TBox(O) \vdash_O EquivalentClasses(D\ T), \\ D = DataSomeValuesFrom(p\ DR), \\ DR = DatatypeRestriction(D\ FA_1^D(lt_1)\ \ldots\ FA_n^D(lt_n))\}\}}} o \in FA_i^D(lt_i) \\
 & & \bigwedge_{\substack{\{(s\ p\ o) \in TP(Q), \\ o \in Var(Q), \\ TBox(O) \vdash_O EquivalentClasses(D\ T), \\ D = DataHasValue(p\ lt)\}}} o = lt \\
 & & \bigwedge_{\substack{\{(s\ p\ o) \in TP(Q), \\ p \in VOP, \\ TBox(O) \vdash_O EquivalentClasses(ObjectSomeValuesFrom(p\ D)\ T)\}}} C_{o:D}
\end{array}
$$

Figure 11: Type Validity

where $C_{?x:T}$ is defined in Figure 11.

In this case, from $?x : T$ the conjunction of constraints $C_{?x:T}$ is built. However this constraint system is recursively built. When $C_{?x:T}$ is built, from each $(s\ p\ o) \in TP(Q)$ and $p \in VOP$ (i.e., object property) the constraint system $C_{o:D}$ is also built, whenever $TBox(O) \vdash_O EquivalentClasses(Ob$-$jectSomeValuesFrom(p\ D)\ T)$. In other words, in order to ensure that $?x$ has type T, the type D for o has to be ensured. Thus, in general a constraint $C_{s:T}$ for a resource s and a type T is built.

The following conditions are mandatory for completeness since they ensure that $C_{s:T}$ is a conjunction of constraints.

Definition 22 (Linear Query). *A query Q is* linear *whenever for every s there do not exist $(s\ p\ o_1)$ and $(s\ p\ o_2) \in TP(Q)$ such that $o_1 \neq o_2$.*

This condition requires that the same resource s and property p can be not used twice. Otherwise, a disjunction of constraints can be required.

Example 13. *Let us suppose the following conjunctive query:*

```
SELECT ?X ?Y ?Z
WHERE {
  ?X sn:dailyLikes ?Y .
  ?X sn:dailyLikes ?Z
  }
```

and the variable typing ?X : Influencer. Thus, the imposed condition on ?X is $?Y >= 100 \bigvee ?Z >= 100$*.*

Definition 23 (Range Restricted).

- *A resource or variable* s *is* object range restricted *in a query* Q *whenever there exists* $(s\ p\ o) \in TP(Q)$ *such that* $p \in VOP$*.*
- *A resource or variable* s *is* data range restricted *in a query* Q *whenever there exists* $(s\ p\ o) \in TP(Q)$ *such that* $p \in VDP$ *and* $o \in Var(Q)$*.*
- *A resource or variable* s *is* range restricted *in a query* Q *whether is object or data range restricted in* Q*.*

Object/data range restricted resources/variables are those ones possibly involved in a constraint.

Definition 24 (Conjunctive Variable Typing). *A typing* $s : T$ *is* conjunctive *in a query* Q *and an ontology* O *if*

(a) when s *is object range restricted and* T *is an object property based definition then:*

(a.1) $TBox(O) \vdash_O EquivalentClasses(ObjectSomeValuesFrom(p\ D)\ T)$*; and*

(a.2) $o : D$ *is conjunctive for each* $(s\ p\ o) \in TP(Q)$ *and* $p \in VOP$*.*

(b) when s *is data range restricted and* T *is a data property based definition then:*

(b.1) either $TBox(O) \vdash_O EquivalentClasses(DataSomeValuesFrom(p\ DR)\ T)$ *where* DR *is a datatype restriction;*

(b.2) or $TBox(O) \vdash_O EquivalentClasses(DataHasValue(p\ lt)\ T)$*,*

We say that T *is conjunctive for* s *in* Q *and* O *whenever* $s : T$ *is conjunctive in* Q *and* O*.*

Thus, a type T for a range restricted s is conjunctive whenever s is involved by T in existentially quantified constraints. In the case of a non range restricted s, all the types T are trivially conjunctive.

Example 14. *Let us suppose the case:*

```
SELECT ?X ?Y ?Z
WHERE {
  ?X sn:friend_of sn:antonio .
  sn:antonio sn:dailyLikes ?Z .
  FILTER(?Z > 50)
  }
```

This query is conjunctive (i.e., antonio has type User and thus trivially conjunctive) and linear. The variable typing ?X : FriendOfInfluencer is conjunctive in the query since: (1) ?X is range restricted and FriendOfInfluencer is equivalent to ObjectSomeValuesFrom(friend_ of Influencer), and (2) antonio : Influencer is conjunctive since EquivalentClasses(Influencer DataSomeValuesFrom(dailyLikes DR)), where $DR = DatatypeRestriction(Integer\ (>= 100))$.

Example 15. `SELECT ?X ?Y ?Z`

```
WHERE {
  ?X sn:friend_of ?Y .
  ?Y sn:dailyActivity ?Z .
  FILTER(?Z > 50)
  }
```

Again, this query is conjunctive and linear. The variable typing ?X : FriendOfActive is not conjunctive in the query: (1) ?X is range restricted and FriendOfInfluencer is equivalent to ObjectSomeValuesFrom(friend_ of Active), but (2) ?Y: Active is not conjunctive since ?Y: Active is equivalent to a DataAllValuesFrom data property based definition.

In other words, ?X : FriendOfActive requires to prove that all the values for dailyActivity are greater or equal than 5, according to the definition of Active. This kind of variable typing causes incompleteness since from just one value for dailyActivity the membership of ?X to the class FriendOfActive cannot be ensured for all the interpretations.

Theorem 2 (Type Validity Method). *Let us suppose a linear and conjunctive query Q and $s : T$ a conjunctive variable typing then $(O_Q^{Var(Q)}, C_Q) \vdash_{CO} (ClassAssertion(T\ s),\ \emptyset)$ iff $\bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(T\ s)$ and $\bar{C}_Q \vdash_C C_{s:T}$, if s is range restricted.*

Proof:

$(\Rightarrow)$*:*

We proceed by induction on $\#\{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ and $b \in VOP\}$.

(I) $\#\{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ *and* $b \in VOP\} = 0$*:*
Given a model I^O of $\theta(\bar{O}_Q^{Res(Q)})$ and $(I^Q, \theta) \models_C \bar{C}_Q$ then, by Theorem 1, $DP(I^O)$ is model of $\theta(O_Q^{Var(Q)}, C_Q)$. Thus $DP(I^O) \models_O ClassAssertion(T\ \theta(s))$. Since $\#\{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ and $b \in VOP\} = 0$, then s cannot be object range restricted. Thus, we need to consider two cases: (a) s is data range restricted and (b) s is not range restricted.

(a) If s is data range restricted, we have two cases: (a.1) T is a data property based definition or (a.2) T is not a data property based definition.

(a.1) If T is a data property based definition, since $s : T$ is conjunctive then:

(a.1.1) either $TBox(O) \vdash_O EquivalentClasses(DataSomeValuesFrom\ (p\ DR)\ T)$, $DR = DatatypeRestriction(D\ FA_1^D(lt_1)\ ...\ FA_n^D\ (lt_n))$;

(a.1.2) or $TBox(O) \vdash_O EquivalentClasses(DataHasValue(p\ lt)\ T)$.

By $DP(I^O)$ construction, since $DP(I^O) \models_O ClassAssertion\ (T\ \theta(s))$, for every $(s\ p\ o) \in TP(Q)$ such that $o \in Var(Q)$, then either (a.1.1) for some i, $\theta(o) \in FA_i^D(lt_i)$ or (a.1.2) $\theta(o) = lt$, where $(\theta(s), \theta(o)) \in I^{DP}(I^P(p))$ in $DP(I^O)$. Since $\#\{(s\ p\ o)\ | (s\ p\ o) \in TP(Q)$ and $p \in VOP\} = 0$ then $C_{s:T}$ only includes either (a.1.1) $o \in FA_i^D(lt_i)$ or (a.1.2) $o = lt$, for every $(s\ p\ o) \in TP(Q)$ such that $o \in Var(Q)$, thus $\bar{C}_Q \vdash_C C_{s:T}$.

(a.2) If T is not a data property based definition, Since $\#\{(s\ p\ o) | (s\ p\ o) \in TP(Q)$ and $p \in VOP\} = 0$ then $C_{s:T}$ is empty, and thus $\bar{C}_Q \vdash_C C_{s:T}$ holds trivially.

(b) *If s is not range restricted then $O_Q^{Var(Q)}$ and $\bar{O}_Q^{Res(Q)}$ coincide in assertions about s. Thus, from $DP(I^O) \models_O$ ClassAssertion(T $\theta(s)$) we have $I^O \models_O$ ClassAssertion(T $\theta(s)$).*

(II) $\#\{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ *and* $b \in VOP\} > 0$:
Given a model I^O of $\theta(\bar{O}_Q^{Res(Q)})$ and $(I^Q, \theta) \models_C \bar{C}_Q$ then, by Theorem 1, $DP(I^O)$ is model of $\theta(O_Q^{Var(Q)}, C_Q)$. Thus $DP(I^O) \models_O$ ClassAssertion(T $\theta(s)$). We need to consider three cases: (a) s is object range restricted, (b) s is data range restricted or (c) s is not range restricted.

(a) *If s is object range restricted, since $s : T$ is conjunctive then $o : D$ is also conjunctive by definition. We have two cases: (a.1) T is an object property based definition and (a.2) T is not an object property based definition.*

(a.1) *If T is an object property based definition, since T is conjunctive for s then $TBox(O) \vdash_O$ EquivalentClasses(Object-SomeValuesFrom(D p) T). Since $TBox(O)$ is included in $O_Q^{Var(Q)}$, we have that if $(O_Q^{Var(Q)}, C_Q) \vdash_{CO}$ (ClassAssertion(T s), $\emptyset$) then $(O_Q^{Var(Q)}, C_Q) \vdash_{CO}$ (ClassAssertion(D o), $\emptyset$). Given that Q is linear, we can apply induction hypothesis on U where $U = S \backslash K$, $S = \{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ and $b \in VOP\}$ and $K = \{(s\ p\ o) |\ (s\ p\ o) \in TP(Q)$ and $p \in VOP\} \neq \emptyset$ because $\#K = 1$, in such a way that:*

(a.1.1) $\bar{O}_Q^{Res(Q)} \vdash_O$ *ClassAssertion(D o) and*

(a.1.2) $\bar{C}_Q \vdash_C C_{o:D}$ *if o is range restricted.*

In case (a.1.1), since $TBox(O)$ is included in $\bar{O}_Q^{Res(Q)}$ then $I^O \models_O$ ClassAssertion(T $\theta(s)$). In case (a.1.2) $\bar{C}_Q \vdash_C C_{s:T}$ because $C_{s:T}$ only contains $C_{o:D}$.

(a.2) *By construction of $DP(I^O)$, If T is not an object property based definition then from $DP(I^O) \models_O$ ClassAssertion(T $\theta(s)$) we have that $I^O \models_O$ ClassAssertion(T $\theta(s)$).*

(b) *If s is data range restricted we need to consider two cases: (b.1) T is a data property based definition or (b.2) T is not a data property based definition.*

(b.1) *If T is a data property based definition, since $s : T$ is conjunctive then,* *(b.1.1) either $TBox(O) \vdash_O EquivalentClasses(DataSomeValuesFrom\ (p\ DR)\ T)$, $DR = DatatypeRestriction(D\ FA_1^D(lt_1)\ ...\ FA_n^D\ (lt_n))$;*

(b.1.2) or $TBox(O) \vdash_O EquivalentClasses(DataHasValue(p\ lt)\ T)$.

By $DP(I^O)$ construction, since $DP(I^O) \models_O ClassAssertion\ (T\ \theta(s))$, for every $(s\ p\ o) \in TP(Q)$ such that $o \in Var(Q)$, then either (b.1.1) for some i, $\theta(o) \in FA_i^D(lt_i)$ or (b.1.2) $\theta(o) = lt$, where $(\theta(s), \theta(o)) \in I^{DP}(I^P(p))$ in $DP(I^O)$.
Since $TBox(O)$ is included in $O_Q^{Var(Q)}$, if $(O_Q^{Var(Q)}, C_Q) \vdash_{CO} (ClassAssertion(T\ s), \emptyset)$ then $(O_Q^{Var(Q)}, C_Q) \vdash_{CO} (ClassAssertion(D\ o), \emptyset)$.
Given that Q is linear, we can apply hypothesis induction on $U = S \backslash K$, $S = \{(a\ b\ c) |\ (a\ b\ c) \in TP(Q)$ and $b \in VOP\}$ and $K = \{(s\ p\ o) | (s\ p\ o) \in TP(Q)$ and $p \in VOP\} \neq \emptyset$, because $\#K = 1$, in such a way that:

(b.1.2.1) $\bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(D\ o)$ and

(b.1.2.2) $\bar{C}_Q \vdash_C C_{o:D}$ if o is range restricted.

In case (b.1.2.1), since $TBox(O)$ is included in $\bar{O}_Q^{Res(Q)}$ then $I^O \models_O ClassAssertion(T\ \theta(s))$. In case (b.1.2.2) $C_{s:T}$ includes $C_{o:D}$ and either $o \in FA_i^D(lt_i)$ or $o = lt$, for every $(s\ p\ o) \in TP(Q)$ such that $o \in Var(Q)$, thus $\bar{C}_Q \vdash_C C_{s:T}$.

(b.2) By construction of $DP(I^O)$, if T is not a data property based definition from $DP(I^O) \models_O ClassAssertion(T\ \theta(s))$, we have that $I^O \models_O ClassAssertion(T\ \theta(s))$.

(c) If s is not range restricted then $O_Q^{Var(Q)}$ and $\bar{O}_Q^{Res(Q)}$ coincide in assertions about s. Thus, from $DP(I^O) \models_O ClassAssertion(T\ \theta(s))$, we have that $I^O \models_O ClassAssertion(T\ \theta(s))$.

$(\Leftarrow)$*:*
Given a model I^Q of $\theta(O_Q^{Var(Q)}, C_Q)$ then, by Theorem 1, I^O is model of $\theta(\bar{O}_Q^{Res(Q)})$ and $(I^Q, \theta) \models_O \bar{C}_Q$. If s is not range restricted, we can conclude trivially the result. Otherwise, we proceed by induction on $\#\{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ and $b \in VOP\}$.

(I) $\#\{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ *and* $b \in VOP\} = 0$.
In this case, s *has to be data range restricted and since* $s : T$ *is conjunctive then:*

  (a) *either* $TBox(O) \vdash_O EquivalentClasses(DataSomeValuesFrom$ $(p\ DR)\ T)$, $DR = DatatypeRestriction(D\ FA_1^D(lt_1)\ ...\ FA_n^D$ $(lt_n))$;

  (b) *or* $TBox(O) \vdash_O EquivalentClasses(DataHasValue(p\ lt)\ T)$

Thus in case (a) $C_{s:T}$ *is* $o \in FA_i^D(lt_i)$ *and in case (b)* $C_{s:T}$ *is* $o = lt$*, and no more restrictions occur in* $C_{s:T}$ *because* $\#\{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ *and* $b \in VOP\} = 0$*. From* $\bar{C}_Q \vdash_O C_{s:T}$ *and* $(I^Q, \theta) \models_O \bar{C}_Q$ *we can conclude that* $I^O \models_O ClassAssertion(T\ \theta(s))$.

(II) $\#\{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ *and* $b \in VOP\} > 0$.

  (a) *If* s *is object range restricted, we need to consider two cases: (a.1)* T *is a object property based definition, and (a.2) is not a object property based definition.*

    (a.1) *If* T *is a object property based definition since* T *is conjunctive for* s *then* $TBox(O) \vdash_O EquivalentClasses(ObjectSomeValuesFrom(D\ p)\ T)$*. Now,* $C_{s:T}$ *contains* $C_{o:D}$ *for each* $(s\ p\ o) \in TP(Q)$ *such that* $p \in VPO$*, and* $TBox(O) \vdash_O EquivalentClasses(ObjectSomeValuesFrom(p\ D)\ T)$.
    We can apply hypothesis induction on $U = S \backslash K$, $S = \{(a\ b\ c) | (a\ b\ c) \in TP(Q)$ *and* $b \in VOP\}$ *and* $K = \{(s\ p\ o) | (s\ p\ o) \in TP(Q)$ *and* $p \in VOP\} \neq \emptyset$ *because* $\#K = 1$*, and thus from* $\bar{C}_Q \vdash_O C_{s:T}$ *and* $(I^Q, \theta) \models_O \bar{C}_Q$ *then* $I^O \models_O ClassAssertion(D\ \theta(o))$*. Finally, we can conclude that* $I^O \models_O ClassAssertion(T\ \theta(s))$.

    (a.2) *If* T *is not an object property based definition then trivially* $I^O \models_O ClassAssertion(T\ \theta(s))$.

  (b) *If* s *is data range restricted, we have two cases: (b.1)* T *is a data property based definition, and (b.2) is not a data property based definition.*

    (b.1) T *is a data property based definition, since* $s : T$ *is conjunctive then:*

(b.1.1) $TBox(O) \vdash_O EquivalentClasses(DataSomeValuesFrom$ *(p DR) T), DR* $= DatatypeRestriction(D\ FA_1^D(lt_1)\ ...\ FA_n^D\ (lt_n))$*; or*

(b.1.2) $TBox(O) \vdash_O EquivalentClasses(DataHasValue(p\ lt)\ T)$

Thus in case (b.1.1) $C_{s:T}$ *contains* $o \in FA_i^D(lt_i)$ *and in case (b.1.2)* $C_{s:T}$ *contains* $o = lt$*. From* $\bar{C}_Q \vdash_O C_{s:T}$ *and* $(I^Q, \theta) \models_O \bar{C}_Q$*, then* $I^O \models_O ClassAssertion(T\ \theta(s))$*.*

(b.2) T *is not a data property based definition, then trivially* $I^O \models_O ClassAssertion(T\ \theta(s))$*.*

When the type validity method fails, i.e., $(O_Q^{Var(Q)}, C_Q) \not\vdash_{CO} (ClassAssertion(T\ ?x), \emptyset)$, the method will report the following answers:

(1) the query Q is incomplete for $?x : T$ and O, i.e., some information is *missing: either triple pattern or filter condition.* In such case, the method will report either the triple patterns of the proof $\bar{O}_Q^{Res(Q)} \vdash_O ClassAssertion(T\ ?x)$ which cannot be proved or the constraints of the proof $\bar{C}_Q \vdash_C C_{?x:T}$ which cannot be proved.

(2) the query Q is faulty with regard to the variable typing $?x : T$ and O. In such case $\bar{O}_Q^{Res(Q)} \wedge ClassAssertion(T\ ?x)$ and $\bar{C}_Q \wedge C_{?x:T}$ are proved to be consistent and the method will report either the cause of the inconsistency of either $\bar{O}_Q^{Res(Q)} \wedge ClassAssertion(T\ ?x)$ or $\bar{C}_Q \wedge C_{?x:T}$.

(3) the variable typing $?x : T$ has a counterexample with regard to Q and O. In such case, the method will report the valuations θ satisfying $\bar{C}_Q \wedge \neg C_{?x:T}$. Instead of enumerating valuations, the method reports a solved form of constraints.

8. Implementation Details

We have proposed a framework for detecting wrong queries of SPARQL. The developed tool for analyzing SPARQL queries available at `http://minerva.ual.es:8090/CTSPARQL/` performs some additional checkins. Firstly, the tool rejects queries involving elements not declared in the ontology vocabulary.

Example 16. *For instance, the following query is rejected:*

```
SELECT ?X
WHERE {
  sn:foo sn:unknown ?X
  }
```

Secondly, the tool rejects queries with wrong use of variables, resources and literals, such as shown in the following example:

Example 17. *Queries rejected by the tool:*

```
SELECT ?USER ?P ?EVENT
WHERE {
  ?USER ?P ?EVENT .
  ?USER sn:age ?P
  }
```

```
SELECT ?USER
WHERE {
  ?USER rdf:type 10
  }
```

In the left hand side, the variable ?P occurs in both resource and literal positions, while in the right hand side the literal 10 occurs in a resource position. In both cases, any interpretation and valuation can be found satisfying the query.

In order to reject such queries, each element of the query is typed (variable, resource and literal) in *rdfs:resource* and *rdfs:literal*, rejecting those ones occurring in both types.

On the other hand, the implementation detects ill typed queries due to ill typed filter conditions.

Example 18. *For instance, the following queries are automatically rejected as ill typed:*

```
SELECT ?USER ?VALUE
WHERE {
  ?USER sn:name ?VALUE .
  FILTER (?VALUE > 10)
  }
```

```
SELECT ?USER ?NAME ?AGE
WHERE {
  sn:jesus sn:name ?NAME .
  sn:jesus sn:age ?AGE .
  FILTER (?NAME > ?AGE)
  }
```

In the left hand side, the filter condition (i.e., ?VALUE > 10) is incompatible with the triple pattern ?USER name ?VALUE due to the range of name. In the right hand side, the same happens since name and age have distinct ranges. Let us remark that according to the definitions 14 and 15 these queries are ill typed (i.e., they are satisfiable but not correct).

In order to detect such cases, each variable/literal occurring in a literal position is typed (as integer, real, string, etc) from the context and when the collected types for a variable/literal are incompatible, the tool reports the ill typedness.

Now, we give some implementation details about datatypes. In our framework, ontology vocabularies are equipped with datatypes, and facets act on datatypes. Also, queries introduce partial order relations in these datatypes, and functions that act on datatypes. The implementation of the proposed methods is currently limited to the integers and real numbers, and facets are limited to *minInclusive*, *maxInclusive*, *minExclusive* and *maxExclusive*. Additionally, the current implementation is limited to arithmetic functions. Thus, the handled constraint systems are linear constraints over integers or real numbers. In the first case (i.e., integers) we have constraints over finite domains, and the reasoning (consistency and entailment) is carried out by SWI-Prolog (i.e., CLP(FD)). In the second case, (i.e., real numbers) we have constraints over real numbers, and the reasoning (consistency and entailment) is also carried out by SWI-Prolog (i.e., CLP(R)).

Example 19. *These queries are examples which can be analyzed in the current implementation:*

```
SELECT ?USER1 ?USER2
WHERE {
  ?USER1 sn:age ?AU1 .
  ?USER2 sn:age ?AU2 .
  FILTER(?AU1-?AU2 < 10) .
  FILTER(?AU1 > 40 ) .
  FILTER (?AU2 < 18)
  }
```

```
SELECT ?USER
WHERE {
  ?USER sn:height ?HU  .
  FILTER(?HU > 130 ).
  FILTER (?HU < 131)
  }
```

The left hand side is not satisfiable, while the right hand side is satisfiable.

Moreover, the current implementation is also able to handle the (natural) order on dates and strings in filter conditions. Dates and strings are encoded as integers.

Let us remark that the current implementation checks whether the query is conjunctive (for correctness) as well as it checks whether the query is linear and the type is conjunctive (for type validity).

9. Conclusions and Future Work

In this paper we have presented two methods for detecting wrong SPARQL queries: the correctness method is used for detecting wrongly typed and unsatisfiable queries and the type validity method is used for detecting mismatching between user intention and queries. We have formally defined such concepts in terms of interpretations of ontologies, and we have proved correctness and completeness results of the proposed methods. The proposed methods have been implemented using ontology and constraint reasoning. A Web online tool has been developed (see `http://minerva.ual.es:8090/CTSPARQL/`) to analyze SPARQL queries.

While SPARQL is a mature technology, the lack of debugging tools impose limits to the frustrated users due to program bugs. The proposed methods aim to mitigate such absence, providing an automatic method for detecting wrong queries. We have also illustrated the approach with a list of examples which also allows to draw the main problems found for writing correct SPARQL queries. More query examples (with ontologies found in the Internet) can be tested at CTSPARQL web site.

As future work, we will pursue the following lines of research.

(1) We have restricted the type of assertions in ontologies and constructors of SPARQL. On one hand, we have assumed that a SPARQL query is a conjunction of triple patterns and filter conditions. It includes nested SELECT expressions and BIND statement. Introducing FILTER OR, MINUS, OPTIONAL and (NOT) EXISTS in SPARQL involves to extend the proposed methods allowing disjunction of formulas.

(2) We have restricted the kind of OWL assertions. NegativeDataPropertyAssertion and DisjointDataProperties assertions force to use disjunction of constraints. The same can be said for DataUnionOf, DataComplementOf and DataOneOf. FunctionalDataProperty, HasKey, DataMinCardinality, DataMaxCardinality and DataExactCardinality are focused on cardinality restrictions and thus the aggregation handling in constraint systems is required.

Example 20. *For instance, when age is a FunctionalDataProperty or DataMaxCardinality is 1, the following (non linear) query is .*

```
SELECT ?USER
WHERE {
  ?USER sn:age ?AGE .
  ?USER sn:age ?AGE2
  FILTER (?AGE != ?AGE2)
  }
```

From a theoretical point of view, it does not introduce significant problems, but both methods have to be adapted. A special case is the introduction of aggregation in SPARQL, which also involve aggregation handling in constraint systems (Chu and Stuckey, 2014).

(3) Non conjunctive and non linear queries as well as non conjunctive types will be handled when disjunction of constraints is allowed.

(4) We would like to incorporate to the implementation the full handling of other datatypes (string, dates, Boolean) and their facets, for instance, *length*, *minLength*, *maxLength* for strings, as well as new datatype functions (other than arithmetic) for existing and new datatypes.

(5) We are interested in providing alternative methods for query debugging. One promising direction is the study of a declarative/algorithmic debugging (Caballero et al., 2017).

10. Acknoledgements

This work is partially supported by PROYECTOS DE I+D+i UAL-FEDER under grant UAL18-TIC-A002-B1.

References

Almendros-Jiménez, J. M., Becerra-Terón, A., 2017a. A Web Tool for Type Checking and Testing of SPARQL Queries. In: 17th International Conference on Web Engineering. Springer, pp. 535–538.

Almendros-Jiménez, J. M., Becerra-Terón, A., 2017b. Automatic property-based testing and path validation of XQuery programs. Software Testing, Verification and Reliability 27 (1-2).

Almendros-Jiménez, J. M., Becerra-Terón, A., 2017c. Property-based testing of SPARQL queries. In: 16th International Symposium on Database Programming Languages. ACM, pp. 1–8.

Atre, M., 2015. Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD '15. ACM, New York, NY, USA, pp. 1793–1808.

Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G. H., Lemay, A., Advokaat, N., 2017. gMark: Schema-driven Generation of Graphs and Queries. IEEE Transactions on Knowledge and Data Engineering 29 (4), 856–869.

Bertolino, A., Gao, J., Marchetti, E., Polini, A., 2007a. Automatic test data generation for XML schema-based partition testing. In: Proceedings of the Second International Workshop on Automation of Software Test (AST). IEEE Computer Society, p. 4.

Bertolino, A., Gao, J., Marchetti, E., Polini, A., 2007b. Systematic generation of XML instances to test complex software applications. In: Rapid Integration of Software Engineering Techniques. Springer, pp. 114–129.

Bertolino, A., Gao, J., Marchetti, E., Polini, A., 2007c. TAXI–a tool for XML-based testing. In: Companion to the proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society, pp. 53–54.

Bertolino, A., Lonetti, F., Marchetti, E., 2010. Systematic XACML request generation for testing purposes. In: Software Engineering and Advanced Applications, 36th EUROMICRO Conference on. IEEE, pp. 3–11.

Bischof, S., Krötzsch, M., Polleres, A., Rudolph, S., 2014. Schema-agnostic query rewriting in SPARQL 1.1. In: International Semantic Web Conference. Springer, pp. 584–600.

Bizer, C., Schultz, A., 2009. The Berlin SPARQL benchmark. International Journal on Semantic Web and Information Systems (IJSWIS) 5 (2), 1–24.

Brass, S., Goldberg, C., 2005. Proving the safety of SQL queries. In: Fifth International Conference on Quality Software (QSIC'05). IEEE, pp. 197–204.

Brass, S., Goldberg, C., 2006. Semantic errors in SQL queries: A quite complete list. Journal of Systems and Software 79 (5), 630–644.

Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F., 2010. Applying constraint logic programming to SQL test case generation. In: International Symposium on Functional and Logic Programming. Springer, pp. 191–206.

Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F., 2012. Declarative debugging of wrong and missing answers for SQL views. In: International Symposium on Functional and Logic Programming. Springer, pp. 73–87.

Caballero, R., Riesco, A., Silva, J., 2017. A survey of algorithmic debugging. ACM Computing Surveys (CSUR) 50 (4), 60.

Chays, D., Deng, Y., Frankl, P. G., Dan, S., Vokolos, F. I., Weyuker, E. J., 2004. An AGENDA for testing relational database applications. Software Testing, verification and reliability 14 (1), 17–44.

Chu, G., Stuckey, P. J., 2014. Nested constraint programs. In: International Conference on Principles and Practice of Constraint Programming. Springer, pp. 240–255.

Clark, J., DeRose, S., 1999. Xml path language (xpath). version 1.0. https://www.w3.org/TR/1999/REC-xpath-19991116/, W3C Recommendation.

De La Riva, C., Garcia-Fanjul, J., Tuya, J., 2006. A Partition-based approach for XPath testing. In: Software Engineering Advances, International Conference on. IEEE, pp. 17–17.

De La Riva, C., Suárez-Cabal, M. J., Tuya, J., 2010. Constraint-based test database generation for SQL queries. In: Proceedings of the 5th Workshop on Automation of Software Test. ACM, pp. 67–74.

Dietrich, B., Grust, T., 2015. A SQL Debugger Built from Spare Parts: Turning a SQL: 1999 Database System into Its Own Debugger. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 865–870.

Glimm, B., Ogbuji, C., Hawke, S., Herman, I., Parsia, B., Polleres, A., Seaborne, A., 2013. Sparql 1.1 entailment regimes. `https://www.w3.org/TR/sparql11-entailment`, W3C Recommendation.

Görlitz, O., Thimm, M., Staab, S., 2012. Splodge: Systematic generation of SPARQL benchmark queries for linked open data. In: International Semantic Web Conference. Springer, pp. 116–132.

Guagliardo, P., Libkin, L., 2016. Making SQL queries correct on incomplete databases: A feasibility study. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. ACM, pp. 211–223.

Guo, Y., Pan, Z., Heflin, J., 2005. LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web 3 (2), 158–182.

Gupta, B. P., Vira, D., Sudarshan, S., 2010. X-data: Generating test data for killing SQL mutants. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). IEEE, pp. 876–879.

Harris, S., Seaborne, A., 2013. SPARQL 1.1 Query Language. `https://www.w3.org/TR/sparql11-query/`, W3C Recommendation.

Holzbaur, C., 1995. OFAI CLP(Q,R) Manual. Tech. rep., Edition 1.3. 3. Technical Report TR-95-09, Austrian Research Institute.

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

Javid, M. A., Embury, S. M., 2012. Diagnosing faults in embedded queries in database applications. In: Proceedings of the 2012 Joint EDBT/ICDT Workshops. ACM, pp. 239–244.

Kim-Park, D. S., de la Riva, C., Tuya, J., 2010. An Automated Test Oracle for XML processing programs. In: Proceedings of the First International Workshop on Software Test Output Validation. ACM, pp. 5–12.

Kontchakov, R., Rezk, M., Rodriguez-Muro, M., Xiao, G., Zakharyaschev, M., 2014. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In: International Semantic Web Conference. Springer, pp. 552–567.

Kontokostas, D., Westphal, P., Auer, S., Hellmann, S., Lehmann, J., Cornelissen, R., 2014a. Databugger: a test-driven framework for debugging the web of data. In: Proceedings of the 23rd International Conference on World Wide Web. ACM, pp. 115–118.

Kontokostas, D., Westphal, P., Auer, S., Hellmann, S., Lehmann, J., Cornelissen, R., Zaveri, A., 2014b. Test-driven evaluation of linked data quality. In: Proceedings of the 23rd international conference on World Wide Web. ACM, pp. 747–758.

Le, W., Kementsietsidis, A., Duan, S., Li, F., 2012. Scalable multi-query optimization for SPARQL. In: 2012 IEEE 28th International Conference on Data Engineering. IEEE, pp. 666–677.

Letelier, A., Pérez, J., Pichler, R., Skritek, S., Dec. 2013. Static Analysis and Optimization of Semantic Web Queries. ACM Trans. Database Syst. 38 (4), 25:1–25:45.

Liu, C., Wang, H., Yu, Y., Xu, L., 2010. Towards efficient SPARQL query processing on RDF data. Tsinghua science and technology 15 (6), 613–622.

Morsey, M., Lehmann, J., Auer, S., Ngomo, A.-C. N., 2011. DBpedia SPARQL benchmark–performance assessment with real queries on real data. In: International semantic web conference. Springer, pp. 454–469.

Pan, Z., Zhu, T., Liu, H., Ning, H., 2018. A survey of RDF management technologies and benchmark datasets. Journal of Ambient Intelligence and Humanized Computing 9 (5), 1693–1704.

Qiao, S., Özsoyoğlu, Z. M., 2015. Rbench: Application-specific RDF benchmarking. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 1825–1838.

Robie, J., Dyck, M., Spiegel, J., 2012. Xquery 3.1: An xml query language. `https://www.w3.org/TR/xquery-31/`, W3C Recommendation.

Saleem, M., Ali, M. I., Hogan, A., Mehmood, Q., Ngomo, A.-C. N., 2015a. LSQ: the linked SPARQL queries dataset. In: International semantic web conference. Springer, pp. 261–269.

Saleem, M., Hasnain, A., Ngomo, A.-C. N., 2018. Largerdfbench: a billion triples benchmark for SPARQL endpoint federation. Journal of Web Semantics 48, 85–125.

Saleem, M., Khan, Y., Hasnain, A., Ermilov, I., Ngonga Ngomo, A.-C., 2016. A fine-grained evaluation of SPARQL endpoint federation systems. Semantic Web 7 (5), 493–518.

Saleem, M., Mehmood, Q., Ngomo, A.-C. N., 2015b. Feasible: A feature-based SPARQL benchmark generation framework. In: International Semantic Web Conference. Springer, pp. 52–69.

Schmidt, M., Hornung, T., Lausen, G., Pinkel, C., 2009. SP2Bench: a SPARQL performance benchmark. In: 2009 IEEE 25th International Conference on Data Engineering. IEEE, pp. 222–233.

Schmidt, M., Meier, M., Lausen, G., 2010. Foundations of SPARQL query optimization. In: Proceedings of the 13th International Conference on Database Theory. ACM, pp. 4–33.

Shah, S., Sudarshan, S., Kajbaje, S., Patidar, S., Gupta, B. P., Vira, D., 2011. Generating test data for killing SQL mutants: A constraint-based approach. In: 2011 IEEE 27th International Conference on Data Engineering. IEEE, pp. 1175–1186.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., Katz, Y., 2007. Pellet: A practical OWL-DL reasoner. Web Semantics: science, services and agents on the World Wide Web 5 (2), 51–53.

Suárez Cabal, M. J., Tuya González, P. J., 2009. Structural coverage criteria for testing SQL queries. Journal of Universal Computer Science, 15 (3).

The OWL Working Group, 2012. Web ontology language (owl). `https://www.w3.org/OWL/`, W3C Recommendation.

The RDF Working Group, 2014. Resource Description Framework (RDF). `https://www.w3.org/RDF/`, W3C Recommendation.

Triska, M., 2012. The finite domain constraint solver of SWI-Prolog. In: International Symposium on Functional and Logic Programming. Springer, pp. 307–316.

Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V., Boncz, P., 2012. Heuristics-based query optimisation for SPARQL. In: Proceedings of the 15th International Conference on Extending Database Technology. ACM, pp. 324–335.

Tuya, J., Dolado, J., Suarez-Cabal, M. J., de la Riva, C., 2008. A controlled experiment on white-box database testing. ACM SIGSOFT Software Engineering Notes 33 (1), 8.

Tuya, J., Suárez-Cabal, M. J., De La Riva, C., 2006. A practical guide to SQL white-box testing. ACM SIGPLAN Notices 41 (4), 36–41.

Tuya, J., Suárez-Cabal, M. J., De La Riva, C., 2007. Mutating database queries. Information and Software Technology 49 (4), 398–417.

Tzompanaki, K., Sep. 2014. Semi-automatic SQL Debugging and Fixing to solve the Missing-Answers Problem. In: Very Large Databases (VLDB'14) PhD Workshop. Hangzhou, China.

Vidal, M.-E., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A., 2010. Efficiently joining group patterns in SPARQL queries. In: Extended Semantic Web Conference. Springer, pp. 228–242.

Yakovets, N., Godfrey, P., Gryz, J., 2016. Query planning for evaluating SPARQL property paths. In: Proceedings of the 2016 International Conference on Management of Data. ACM, pp. 1875–1889.

Zhang, J., Xu, C., Cheung, S.-C., 2001. Automatic generation of database instances for white-box testing. In: Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International. IEEE, pp. 161–165.

Zhang, Y., Duc, P. M., Corcho, O., Calbimonte, J.-P., 2012. SRBench: a streaming RDF/SPARQL benchmark. In: International Semantic Web Conference. Springer, pp. 641–657.