Pleasantly Consuming Linked Data with RDF Data Descriptions

Michael Schmidt[1⋆] and Georg Lausen[2]

[1] fluid Operations AG
Altrottstraße 31, 69190 Walldorf, Germany
michael.schmidt@fluidops.com

[2] University of Freiburg, Institute for Computer Science
Georges-Köhler-Allee, 79110 Freiburg, Germany
lausen@informatik.uni-freiburg.de

Abstract. Although the intention of RDF is to provide an open, minimally constraining way for representing information, there exists an increasing number of applications for which guarantees on the structure and values of an RDF data set become desirable if not essential. What is missing in this respect are mechanisms to tie RDF data to quality guarantees akin to schemata of relational databases, or DTDs in XML, in particular when translating legacy data coming with a rich set of integrity constraints – like keys or cardinality restrictions – into RDF. Addressing this shortcoming, we present the RDF Data Description language (RDD), which makes it possible to specify instance-level data constraints over RDF. Making such constraints explicit does not only help in asserting and maintaining data quality, but also opens up new optimization opportunities for query engines and, most importantly, makes query formulation a lot easier for users and system developers. We present design goals, syntax, and a formal, First-order logics based semantics of RDDs and discuss the impact on consuming Linked Data.

1 Introduction

Since the early days of relational databases, constraints have been considered essential to specify the intended states of the data sets representing the information of certain applications [1, 2]. In recent years the number of applications that are based on large scale distributed data available on the Internet has been constantly increasing. Many of them are based on RDF [3] and the question arises, whether relational database like constraints can be considered essential for such applications, as well. RDF data often comes together with RDFS [3] or even OWL [4], and it is well-known that these languages are not intended to cover relational constraints [5]. Further, designed as rule languages, they do not offer mechanisms to express constraints explicitly over the instance data [6–8].

⋆ *Current affiliation:* imc information multimedia communication AG, Scheer Tower, Uni-Campus Nord, 66123 Saarbrücken, Germany.

As an example, consider the work on mapping relational databases to RDF from the W3C's Direct Mapping and R2RML initiatives [9, 10]. While they may exploit relational integrity constraints to increase the mapping quality, these constraints are at most *implicit* in the resulting RDF database: for a data consumer, who may not be aware of the underlying mappings, no *explicit* guarantees about properties and structure of the data are available. For instance, in the Direct Mapping approach [9] primary keys are exploited to generate unique IRIs for objects using the key column names and values; yet, there is no constraint in RDF describing that the properties derived from the key columns are single-valued and identify the resulting objects. In fact, designed as rule languages neither RDF(S) nor OWL allow to express constraints [5]. Although their built-in semantics may imply certain constraints (such as type inheritance at instance-level for *rdfs:subClass* relationships), constraints are only *implicit* and, moreover, may not hold when the data is published under ground semantics – a common scenario in the Linked Data context.

Contributions. After motivating the need to enable end users in writing precise SPARQL queries in Section 2, we present the *RDF Data Description* language, RDD, to define constraints over RDF, akin to DTDs for XML. We then discuss design decisions and related work in Section 3, identifying the need for RDDs to be both *user-readable* and *machine-processable*. Next, we elaborate on the conflict between the Open World Assumption underlying RDF(S) and the requirements of a hard constraint language, concluding that RDDs shall support a *pay-as-you-go paradigm* in constraining RDF(S). Section 4 formalizes RDDs by means of a *user-friendly syntax* that captures a broad range of constraints including keys, cardinalities, subclass, and subproperty restrictions. Section 5 presents a *First-order Logics semantics*, making it easy to implement RDD checkers and clearing the way for optimizations. Finally, in Section 6 we discuss *directions of future research*, including the implementation, coverage, extensibility, and relationship to standards like VoID [11].

2 RDD by Example

As a motivating example, assume a developer wants to write a SPARQL query that extracts information about persons in an RDF document, described by properties `rdfs:label` (denoting the name), `foaf:age`, and `foaf:mbox` (mail address) – where every person shall be represented by exactly one row of the result table.[3] While this sounds like a fairly trivial

[3] In fact, the problem is a slightly modified example one of the authors recently encountered in the context of an industrial project.

task (in SQL, with a reasonable schema, this could probably be expressed by a simple query like `SELECT id, name, age, email FROM Person`), with the unconstrained RDF model this may become quite tricky, even if the schema (i.e., FOAF and RDF(S) vocabulary) is well known to the developer: without further knowledge about the *instance data*, the developer cannot be sure which predicates are present at all, and which of them may be multi-valued. Making guesses that `rdfs:label` and `foaf:age` are single-valued, the developer may finally come up with the following query:

```
SELECT ?person ?name ?age (GROUP_CONCAT(?mail; separator=", ")  AS ?mail)
WHERE {  ?person rdf:type foaf:Person .
         OPTIONAL { ?person rdfs:label ?name }
         OPTIONAL { ?person foaf:age ?age }
         OPTIONAL { ?person foaf:mbox ?mail }   } GROUP BY ?name ?age
```

The OPTIONAL clauses ensure that persons with incomplete information are included in the result; to group persons with multiple email addresses, the developer used `GROUP BY` combined with `GROUP_CONCAT` in the `SELECT` clause, thus concatenating all email addresses of a single person. The crucial point here is that even this simple task leads to a quite complex query covering the "worst case scenario" anticipated by the developer, requiring the use of advanced SPARQL 1.1 constructs (which, as a matter of fact, are hard to optimize by query engines). And even this carefully designed query leads to multiple result rows for the same person in the presence of multiple labels (e.g., with different language tags).

What is needed to ease SPARQL query development is a data description that describes the structural constraints of the instance data *beyond* the schema information contained in the underlying RDF(S) specification and ontologies, which the developer can consult when writing queries. The RDD language advocated in this paper was designed with exactly this goal in mind. RDD would allow the data publisher to express the instance data constraints by means of a well-defined, human readable language.

Figure 1 depicts an example RDD that, when tied to a specific RDF database, helps the developer in understanding the constraints that hold on instance level. With respect to the concept *foaf:Person*, the first part of the RDD in Figure 1 (left) specifies a set of constraints that are known to hold for every instance of the class. Summarizing the relevant part of the RDD, it tells the developer that the property *rdfs:label* serves as a key for persons, every person has exactly one *foaf:email* property (keyword `TOTAL`), and every person has at most one *foaf:age* (keyword `FUNCTIONAL`). Further, all theses three properties point to literals, the latter being of type *xsd:integer* – this may be useful information when writing e.g. aggregation queries over the age, or when post-formatting the results.

```
PREFIX ex: <http://www.example.com#>
...

CWA CLASSES {
 OWA CLASS foaf:Person SUBCLASS ex:Student {
   KEY rdfs:label : LITERAL
   TOTAL foaf:email : LITERAL
   FUNCTIONAL foaf:age : LITERAL(xsd:integer)
   RANGE(foaf:Person) foaf:knows : IRI }
 OWA CLASS ex:Student {
   TOTAL ex:matricNr : LITERAL(xsd:integer)
   MIN(1), RANGE(ex:Course) ex:course : RESOURCE
   PATH(ex:course/ex:givenBy), RANGE(foaf:Person)
     ex:taughtBy : IRI }
}

OWA PROPERTIES {
  TOTAL rdfs:label
  foaf:knows SUBPROPERTY ex:taughtBy }
```

Fig. 1. Example RDF Data Description

With the RDD specification at hand – which can be understood in few seconds – the developer can considerably simplify the query:

```
SELECT ?person ?name ?age ?mail .
WHERE { ?person rdf:type foaf:Person ; rdfs:label ?name ; foaf:mbox ?mail .
        OPTIONAL { ?person foaf:age ?age } }
```

Even if the developer is not aware of the RDD and comes up with a query that uses, e.g., redundant OPTIONAL blocks, the RDD may still be used by the optimizer to simplify the query and speed up evaluation.

3 Design Decisions and Related Work

Philosophy. RDDs specify constraints that hold in an RDF data set. However, not to loose RDF's minimally constraining way for representing information – where one may interlink and extend data sets by adding new information – they shall not require the structure of RDF to be defined *completely*, but give a pragmatic answer to these two conflicting design goals in that they adhere to RDF's Open World character following a *pay-as-you-go paradigm*, which allows users to impose constraints only on a subset of classes, or to constrain classes and properties only partially.

Designed for Humans. To make it easy for humans to understand, write, and use RDDs as a guide when writing queries, RDDs shall come with a *user-understandable* syntax. To this end, we use an object-oriented approach closely aligned to the RDF(S) data model, reusing concepts like classes, properties, and subclass/subproperty relationships. An RDF serialization of RDDs is out of the scope of this paper (cf. Section 6).

Scope. The importance of constraints for RDF(S) has recently been emphasized in the context of REST-based enterprise applications [12]. With the goal to provide a *machine-readable* language, *OSLC Resource Shape* defines an RDF vocabulary to encode qualified property constraints (such as cardinality, range, or value restrictions). While no formal semantics is given, the authors propose an implementation via SPARQL ASK

queries. RDDs, in contrast, are designed for humans, come with a formal semantics, and go far beyond what can be expressed with OSLC (e.g., expressing completeness guarantees and unqualified property constraints).

Enabling the targeted restriction of RDF(S) constructs, RDDs provide built-in constructs to express constraints over classes, subclasses, and properties such as domain, range, or cardinality restrictions. In the light of the Direct Mapping and R2RML standards [9, 10], RDDs shall also cover constraints from the relational databases domain, in order to carry over integrity information when translating relational data.

RDF data is often equipped with RDFS or OWL axioms and may be interpreted in different entailment regimes. Dedicated studies of constraints in the context of OWL have been presented in [6, 13]. Adhering to the different semantics under which RDF can be published, RDDs should be independent from the entailment regime. Our approach is similar to SPARQL [14], which also supports different entailment regimes (and is defined independently): if, e.g., RDF is published under ground semantics, an associated RDD spec would specify the constraints that hold on the bare instance data; if, e.g., RDFS inferencing is turned on, an RDD specification would take inferred facts into account – in both cases, an end user can transparently rely on the RDD spec when accessing the data.

Formal Semantics. While a SPARQL-based semantics may seem like a natural choice (cf. [12, 8]), we argue that is desirable to choose a semantics that can easily be mapped to existing work on integrity constraints from the relational database community, e.g. to carry over Semantic Query Optimization techniques (e.g., the seminal work [15]). We therefore decided for a First-order Logics (FOL) based semantics, representing constraints as First-order sentences known as tuple-generating and equality-generating dependencies [16], which are well understood from previous investigations (e.g. [15]). A possible implementation of our FOL based semantics by means of SPARQL will be discussed later in Section 6.

4 RDD Syntax and Model

Figure 1 provides an example RDD. The definition for class *foaf:Person* contains the constraints for predicates *rdfs:label*, *foaf:email*, and *foaf:age* discussed in Sec. 2, plus a constraint expressing that predicate *foaf:knows*, when used for an instances of type *foaf:Person*, points to instances of type *foaf:Person*, which are always IRIs (i.e., not blank nodes). In the spirit of RDF, this does not enforce referred objects to be *exclusively* typed as *foaf:Person*, but only that one edge typing the object as *foaf:Person* is present. The `OWA` keyword (short for *Open World Assumption*) in front

of the `CLASS` definition allows persons to have further properties not listed in the class specification; its counterpart, keyword `CWA`, would enforce that the class is *closed* in the sense that class instances are *completely* described by the properties occurring in the `CLASS` section. Further, *foaf:Person* has a subclass *ex:Student* (keyword `SUBCLASS`). With RDDs focusing on instance-level constraints, this does **not** enforce a triple (*ex:Student*, *rdfs:subClassOf*, *foaf:Person*) in the data, but guarantees that every *ex:Student* satisfies the same constraints as *foaf:Person*s.

Similar in spirit, the `CLASS` definition for *ex:Student* guarantees that (i) *ex:matricNr* is a `TOTAL` property, i.e. every student has exactly one matriculation number, (ii) *ex:course* occurs at least once and has `RANGE` *ex:Course*, and (iii) the `PATH` and `RANGE` constraints defined for property *ex:taughtBy* asserts that, for every *ex:Student*, there is a property path along the edges *ex:course* followed by *ex:givenBy* pointing to the same value of type *foaf:Person* as property *ex:taughtBy*. Path constraints are special kinds of inclusion constraint, similar in spirit to foreign keys.

The surrounding `CLASSES` section is defined as `CWA`, guaranteeing completeness in the sense that the data contains only instances of the two classes *foaf:Person* and *ex:Student*. Subsequently, the RDD contains a `PROPERTIES` section constraining properties in an *unqualified*, *global* way. The first entry, `TOTAL rdfs:label`, ensures that *every Resource* has exactly one label – a useful information for query authors. The `SUBPROPERTY` spec, targeting the instance level again, ensures that, for every triple (X,*ex:taughtBy*,Y) in the data, there is an implied triple (X,*foaf:knows*,Y). Finally, the `OWA` keyword of the `PROPERTIES` section expresses that there may be other properties than those listed in the section.

Rather than presenting the concrete syntax (see the Technical Report [17] for this level of detail), Figure 2 visualizes the structure and concepts of the RDD language in a UML-style notation. Boxes denote concepts, arrowed lines sub-concept relationships, and the second line type indicates that concept A uses B. At top-level, RDDs consist of a ClassConstraintSec and a PropConstraintSec, which contain lists of ClassConstraints and PropConstraints, respectively, plus a boolean flag indicating whether the sections should be interpreted under Open or Closed World Assumption (i.e., whether the classes and properties in the sections describe the RDF data completely or nor) – the `OWA` keyword enables RDDs to be written in a pay-as-you-go fashion, where known constraints are specified, while unknown parts are left unspecified.

Central to the RDD concept is the notion of PropConstraints, an abstract concept that is further subclassed into specific subclasses:

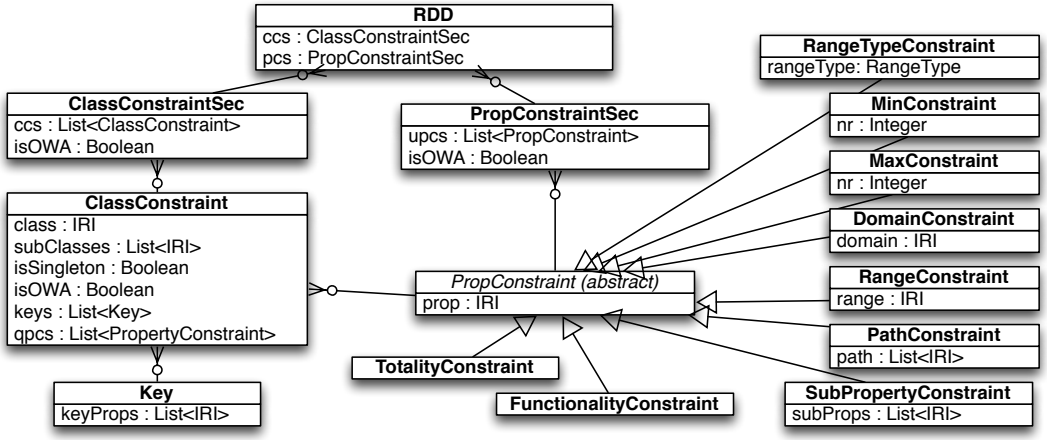

Fig. 2. Structural Overview of the RDD Language

- A RangeTypeConstraint indicates that the property *prop* points to either a URI, BlankNode, Resource, or a (possibly typed) Literal.
- A Min/MaxConstraint indicates that the property *prop* occurs at least or at most *nr* times, respectively.
- A Domain/RangeConstraint indicates a guaranteed *domain* or *range* for subject and objects associated with *prop*, respectively.
- A PathConstraint indicates that the value of *prop* can as well be reached by following a given *path* of properties.
- A SubPropertyConstraint indicates that for every triple using property *subProp*, there is also an identical triple using property *prop*.
- Functionality/TotalityConstraints express that *prop* occurs at most or exactly one time, respectively.

PropConstraints are used in two different contexts: (1) The PropConstraintSec (cf. keyword `PROPERTIES`) contains a list *upcs* of PropConstraints, implementing *unqualified*, global characteristics of properties. For instance, the TotalityConstraint in Fig. 1 (keyword `TOTAL`) for *prop* :=*rdfs:label* asserts that *every resource* has exactly one label. (2) Variable *qpcs* inside ClassConstraints represents *qualified*, class-specific PropConstraints, e.g. the MinConstraint (keyword `MIN`) for *prop* :=*ex:course* and *nr* := 1 in Fig. 1 in the class section of *ex:Student* ensures that *every instance of ex:Student* visits at least one course.

In addition to *qpcs*, a ClassConstraint contains (i) a list of subclasses (keyword `SUBCLASS`), enforcing that instances of the subclasses inherit inner constraints of the superclass, (ii) a boolean flag *isSingleton*, enforcing that exactly one instance of the class exists, (iii) the *isOWA* flag, and (iv) a list of keys. We sketch their semantics in the next section.

5 RDD Semantics

The semantics, denoted by $[\![\mathrm{r}]\!]_E$, decomposes an RDD r into constraints that can be checked individually and independently. It uses an environment E capturing `SUBCLASS` and `SUBPROPERTY` relations specified in the RDD. The result is a set of in First-order Logics (FOL) constraints over relation $T_D(s, p, o)$ representing the RDF triples in RDF document D. With $[\![.]\!]_E$ at hand, we define the notion of *consistency* as follows.

Definition 1. Let D be an RDF document and r be an RDD specification. Further let $cs := [\![\mathrm{r}]\!]_E$ be the set of first-order logic constraints defined by r. Document D is *consistent* w.r.t. r if and only if for all constraints $c \in cs$ it holds that c is valid in T_D, i.e. $T_D \models c$.

The evaluation function $[\![.]\!]_E$ is defined by about 40 rules along the structure of RDDs. At top-level, an RDD is decomposed into its ClassConstraintSec and PropConstraintSec (i.e., the members of class RDD, cf. Fig. 2), which are then further decomposed by dedicated rules. At the core are inference rules mapping the individual constraints – e.g., the PropConstraint subclasses – into FOL. We sketch the idea of the evaluation and refer to the TR [17] for a complete listing of the rules.

Let us exemplarily discuss the inference rule for ClassConstraints, which derives constraints for its key and property constraints, as well as its global "configuration", namely, the subclass hierarchy, whether it is defined as singleton, and whether it is defined under Open or Closed World Assumption. According to Figure 2, the ClassConstraint is represented by a structure (*class*,*subClasses*,*isSingleton*,*keys*,*qpcs*,*isOWA*).

$$\frac{\begin{array}{c} cs_{singleton} := [\![(class,\ isSingleton) : \mathrm{Singleton}]\!]_E \\ cs_{sc} := \bigcup_{c_{sc} \in subClasses} [\![(c_{sc}, \mathbf{false}, E.C(c_{sc}), keys, cpcs, \mathbf{true}) : \mathrm{ClassConstraint}]\!]_E \\ cs_{key} := \bigcup_{key \in keys} [\![(class, key) : \mathrm{ClassKey}]\!]_E \\ cs_{qpcs} := \bigcup_{cpc \in cpcs} [\![(class, cpc) : \mathrm{ClassPropConstraint}]\!]_E \\ cs_{wa} := [\![(class,\ E.A(class),\ isOWA) : \mathrm{OWA}_P]\!]_E \end{array}}{\begin{array}{c} [\![(class : \mathrm{IRI},\ subClasses : \mathrm{List}\langle IRI\rangle,\ isSingleton : \mathrm{Boolean},\ keys : \mathrm{List}\langle Key\rangle, \\ qpcs : \mathrm{List}\langle PropConstraint\rangle,\ isOWA : \mathrm{Boolean}) : \mathrm{ClassConstraint}]\!] \vdash \\ cs_{singleton} \cup cs_{sc} \cup cs_{key} \cup cs_{qpcs} \cup cs_{wa} \end{array}}$$

Starting with the conclusion, the result of evaluating the ClassConstraint is the union of the constraint sets $cs_{singleton}$, cs_{sc}, ...; the premise of the rule describes how these constraint sets are calculated. The

constraint set $cs_{singleton}$, for instance, is obtained by evaluating a substructure *Singleton* with *class* IRI and the *isSingleton* flag as argument – if *isSingleton*=**true**, this substructure generates a constraint enforcing that the class has exactly one instance. The scheme for computing cs_{wa}, cs_{key} and the qualified PropConstraints cs_{qpcs} are analogous. Most interesting is the computation of cs_{sc}, which captures the inheritance of constraints to subclasses. It is obtained by evaluating a replicated version of the class constraint for every subclass c_{sc}. In these replicas, we pass the *keys* and *cpcs* constraints, and consult environment E to obtain the subclasses of the c_{sc}. Note that we neither inherit the singleton constraint (passing *isSingleton*:=**false**) nor impose a `CWA` constraint on the subclass (passing *isOWA*=**true**); this gives the RDD designer greater flexibility.

To conclude, let us sketch the evaluation of a qualified RangeConstraint as one of the constraint-generating rules. The rule below implements such a constraint for property p with range R for class C: the resulting formula enforces that for every instance s of class C, every value o referenced by p is of type R. Note the difference toward RDFS: while predicate *rdfs:range* is similar by idea, it does not enforce the presence of such a triple, but sets up a rule that generates/completes the data – the constraint *guarantees* the presence of such a triple in the database.

$$\frac{}{[\![(C : \text{IRI}, (p : \text{IRI}, R : \text{IRI}) : \text{RangeConstraint})]\!]_E \vdash \{\forall s, o(T_D(s, \textbf{rdf:type}, C) \land T_D(s, p, o) \rightarrow T_D(o, \textbf{rdf:type}, R))\}}$$

6 Discussion and Future Research Direction

RDDs constitute a powerful mechanism to describe instance-level constraints and help both humans – when writing SPARQL queries – and engines – which may exploit RDDs with its concise semantics to assert data consistency and optimize queries (cf. [8]). Our approach opens up a new research field, which we will shortly discuss in the following.

Implementation. Given the First-order semantics, it is straightforward to build a constraint checker using existing FOL engines. In the context of SPARQL engines, though, it may be favorable to implement checkers by means of SPARQL Ask queries. The close connection between SPARQL and logics-based formalisms has been pointed out in several works [18, 19]. In [8] it was shown how to encode constraints such as keys and cardinalities in SPARQL. Further, it was proven in [19] by a constructive proof that every First-order sentence can be expressed in SPARQL. These results make it easy to map the FOL semantics to

SPARQL. The efficient implementation of constraint checking, though, is a challenging task left for future work. A simple approach based on a one-by-one execution of Ask queries may have limitations, in particular when it comes to scenarios with frequent data updates. Here, incremental constraint checking approaches would be required. However, given that integrity constraints are an integral part of other data models (such as relational data and DTDs for XML) and that the logical structure of RDD constraints – EGDs and TGDs, which are well understood in theory – is very similar, we are convinced that efficient implementations are possible.

Deriving RDDs. An interesting topic is the derivation of RDDs from instance data or – when the data has been obtained from relational systems [9, 10] – from constraints in the original data sources. RDFS or OWL specifications, for instance, that are included in the data, or query logs may give valuable hints about candidate constraints that hold in the instance data, allowing to automatically build RDDs that could be refined manually. In this line, it would also be interesting to study interrelations between different entailment regimes and their implications for RDDs.

Coverage and Extensibility. Since the early years of database research, various classes of constraints have been investigated (see e.g. [20, 15]). Based on the design goals from Section 3, we selected a reasonable set of constraints that (i) may be encountered in typical RDF(S) scenarios and (ii) may be of benefit when writing SPARQL queries. There are, of course, constraint types that are currently not supported by RDDs. One extension would be user-defined constraints in the form of arbitrary SPARQL Ask queries – they could be easily added through a new `USER` in RDDs, containing a list of queries including the expected results. While such custom constraints may be hard to understand by users, they could help to model complex data consistency scenarios. Other types of constraints that are candidates for extension include functional dependencies [21], EGDs and TGDs with disjunction [22],[4] or value restriction (e.g. expressing that property *foaf:gender* maps to either *male* or *female*).

As a side note, we want to point out that we intentionally did not include foreign keys as proposed in [8]. Although they play an important role in relational modeling, they are an artificial construct arising due to the relational structure: when mapping relational database into RDF(S), foreign keys typically result in range specifications over object properties (cf. [23, 9]), which RDDs can easily capture using RangeConstraints.

[4] In RDDs, some constraints like the CWA restriction implicitly contain disjunction. We may want to express disjunction in other contexts, e.g. over range restrictions.

Other applications. RDDs can be exploited for use cases beyond query formulation, semantic query optimization, and quality assurance. For instance, the constraints encoded in RDDs may give valuable input for schema mapping and alignment. As another example, RDDs could be used to derive precise data input forms, and thus help in producing data.

RDD and Linked Open Data. Constraint checking, at first glance, may look like a local task. However, as RDF resources being identified by IRIs are globally unique, local checking of constraints for certain constraints classes may not be sufficient. Assume an inclusion dependency, say a range constraint r, is violated in data set R_1, however the violating resource exists and is typed accordingly in a data set R_2. According to the principles of Linked Open Data, r should not be declared to be violated.

We conclude that constraint checking in an LOD environment in general requires us to consider the constraints of all involved data sets. Even though this does not make constraint checking inapplicable, it may require novel ways and paradigms to specify and check constraints in the context of an open world, with possibly incomplete knowledge. We leave a closer investigation of these issues for future work.

Publishing of RDDs. As a bridge between the publishers and users of RDF data, W3C proposes the VoID vocabulary (*Vocabulary of Interlinked Datasets*) [11, 24]. We suggest to develop a canonical RDF representation for RDDs (coexisting with the user-friendly syntax presented in the paper), with tooling to convert between the two syntaxes. This would make it quite easy to, e.g., publish RDD descriptions as part of VoID. A detailed study of the relationships toward VoID and an RDF serialization for the RDD language are interesting topics for future work.

Standardization. The next steps we plan are the implementation of the RDD language by means of a SPARQL query generator that outputs the Ask queries for checking RDDs, which would make RDDs immediately usable by any SPARQL engine. Further, we are investigating different ways to standardize the proposed language, e.g. as part of the W3C standardization activities in the semantic technology space.

Acknowledgments. This work was supported by the German Federal Ministry of Economics and Technology as part of the project "*Durchblick*", grant KF2587503BZ2, and by Deutsche Forschungsgesellschaft as part of the project "*CORSOS*", grant LA 598/7-1.

The authors want to thank Peter Haase and Michael Meier for fruitful discussions and the anonymous reviewers for their constructive feedback.

References

1. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.
2. M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," in *SIGMOD Conference*, 1975, pp. 65–78.
3. "RDF Specification Overview (W3C)," http://www.w3.org/standards/techs/.
4. "OWL 2 Web Ontology Language Document Overview (Second Edition)," http://www.w3.org/TR/owl2-overview/.
5. B. Motik, I. Horrocks, and U. Sattler, "Adding Integrity Constraints to OWL," in *OWLED*, 2007.
6. B. Motik, I. Horrocks, and U. Sattler, "Bridging the Gap Between OWL and Relational Databases," *J. Web Sem.*, vol. 7, no. 2, pp. 74–89, 2009.
7. J. Sequeda, M. Arenas, and D. P. Miranker, "On Directly Mapping Relational Databases to RDF and OWL," in *WWW*, 2012, pp. 649–658.
8. G. Lausen, M. Meier, and M. Schmidt, "SPARQLing Constraints for RDF," in *EDBT*, 2008, pp. 499–509.
9. "A Direct Mapping of Relational Data to RDF," http://www.w3.org/TR/rdb-direct-mapping/.
10. "R2RML: RDB to RDF Mapping Language," http://www.w3.org/TR/r2rml/.
11. K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao, "Describing Linked Datasets - on the Design and Usage of VoID," in *LDOW*, 2009.
12. A. Ryman, A. L. Hors, and S. Speicher, "OSLC Resource Shape: A Language for Defining Constraints on Linked Data," in *LDOW*, 2013.
13. J. Tao, "Adding integrity constraints to the semantic web for instance data evaluation," in *International Semantic Web Conference (2)*, 2010, pp. 330–337.
14. "SPARQL 1.1 Entailment Regimes," www.w3.org/TR/2013/REC-sparql11-entailment-20130321/.
15. D. Maier, A. O. Mendelzon, and Y. Sagiv, "Testing Implications of Data Dependencies," *ACM Trans. Database Syst.*, vol. 4, no. 4, pp. 455–469, 1979.
16. C. Beeri and M. Vardi, "Formal Systems for Tuple and Equality Generating Dependencies," *SIAM Journal on Computing*, vol. 13, no. 1, pp. 76–98, 1984.
17. M. Schmidt and G. Lausen, "Pleasantly Consuming Linked Data with RDF Data Descriptions," 2013, TR, arXiv (submit/0758082).
18. A. Polleres, "From SPARQL to Rules (and Back)," in *WWW*, 2007, pp. 787–796.
19. M. Schmidt, M. Meier, and G. Lausen, "Foundations of SPARQL Query Optimization," in *ICDT*, 2010, pp. 4–33.
20. W. Armstrong, "Dependency Structures of Data Base Relationships," in *IFIP*, 1974, pp. 580–583.
21. E. F. Codd, "Further Normalization of the Data Base Relational Model," *Data base systems*, pp. 33–64, 1972.
22. M. Meier, M. Schmidt, F. Wei, and G. Lausen, "Semantic Query Optimization in the Presence of Types," *J. Comput. Syst. Sci.*, vol. 79, no. 6, pp. 937–957, 2013.
23. J. Sequeda, S. H. Tirmizi, Ó. Corcho, and D. P. Miranker, "Survey of Directly Mapping SQL Databases to the Semantic Web," *Knowledge Eng. Review*, vol. 26, no. 4, pp. 445–486, 2011.
24. "Describing Linked Datasets with the VoID Vocabulary," http://www.w3.org/TR/void/.