CONSTRUCT Queries in SPARQL

Egor V. Kostylev[1], Juan L. Reutter[2], and Martín Ugarte[2]

1 **University of Oxford**
egor.kostylev@cs.ox.ac.uk

2 **PUC Chile**
jreutter@ing.puc.cl, martinugarte@puc.cl

Abstract

SPARQL has become the most popular language for querying RDF datasets, the standard data model for representing information in the Web. This query language has received a good deal of attention in the last few years: two versions of W3C standards have been issued, several SPARQL query engines have been deployed, and important theoretical foundations have been laid. However, many fundamental aspects of SPARQL queries are not yet fully understood. To this end, it is crucial to understand the correspondence between SPARQL and well-developed frameworks like relational algebra or first order logic. But one of the main obstacles on the way to such understanding is the fact that the well-studied fragments of SPARQL do not produce RDF as output.

In this paper we embarrk on the study of SPARQL CONSTRUCT queries, that is, queries which output RDF graphs. This class of queries takes rightful place in the standards and implementations, but contrary to SELECT queries, it has not yet attracted a worth-while theoretical research. Under this framework we are able to establish a strong connection between SPARQL and well-known logical and database formalisms. In particular, the fragment which does not allow for blank nodes in output templates corresponds to first order queries, its well-designed sub-fragment corresponds to positive first order queries, and the general language can be re-stated as a data exchange setting. These correspondences allow us to conclude that the general language is not composable, but the aforementioned blank-free fragments are. Finally, we enrich SPARQL with a recursion operator and establish fundamental properties of this extension.

Digital Object Identifier 10.4230/LIPIcs.ICDT.2015.1

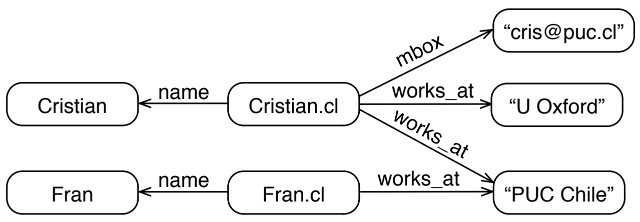
1 Introduction

The Resource Description Framework (RDF) [25] is the World Wide Web consortium (W3C) standard for representing linked data on the Web. Intuitively, an RDF graph is a set of triples of internationalized resource identifiers (IRIs), where the first and last IRI in the triples represent entity resources, and the middle one relates these resources.

SPARQL is a language for querying RDF datasets. Originally introduced in 2006 [34], SPARQL was officially made the recommended language to query RDF data by W3C in 2008 [33]. A recent version of the standard, denoted SPARQL 1.1, was issued in 2013 [40]. Nowadays this language is recognised as one of the key standards of the Semantic Web initiative and there are several SPARQL engines available to industry (e.g., [12, 18, 38]).

The theoretical foundations of SPARQL were laid by Pérez et al. in their seminal work [28], and a body of research has followed covering a variety of issues such as complexity of query evaluation [4, 24, 30, 37], query optimisation [8, 9, 22, 31], federation [7], expressive power [2, 32], and provenance tracking [15, 17]. The impact of these studies in the Semantic Web community has been astonishing, even influencing in the definition of the SPARQL standards.

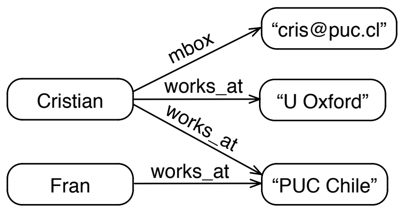
18th International Conference on Database Theory (ICDT'15).
Editors: Marcelo Arenas and Martin Ugarte; pp. 1–26
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a)

	?n	?w	?e
μ_1	Fran	PUC Chile	
μ_2	Cristian	U Oxford	cris@puc.cl
μ_3	Cristian	PUC Chile	cris@puc.cl

(b)



(c)

Figure 1 RDF graph G_{ex} **(a)**; answer of $\boldsymbol{q}_{\text{sel}}$ over G_{ex} is the set of mappings $\{\mu_1, \mu_2, \mu_3\}$ **(b)**; answer of $\boldsymbol{q}_{\text{cons}}$ over G_{ex} is RDF graph **(c)**.

Despite the key importance of SPARQL, the fundamental aspects of this language are still not fully understood. Compared to the knowledge we have on other query languages such as SQL, Datalog or even XPath, very little is known about SPARQL queries. To this end, it is of particular importance to understand the correspondence between SPARQL and other well-developed formalisms such as first order logic or relational algebra. One of the main obstacles on the way to such understanding is the fact that the queries from well-studied fragments of SPARQL produce not RDF graphs as answers, but sets of mappings (partial evaluations), which is a different form for representing data.

▶ **Example 1.** As a classical example of SPARQL, let us consider the following query $\boldsymbol{q}_{\text{sel}}$:[1]

```
SELECT ?n, ?w, ?e
WHERE (
((?p, name, ?n) AND (?p, works_at, ?w))
         OPT   (?p, mbox, ?e)).
```

This query is intended to extract all names and affiliations of people for which a working place is known, appending their emails when available in the RDF graph. Thus, when evaluated on the RDF graph G_{ex} from Figure 1(a), it gives as result a set of partial mappings from the variables of $\boldsymbol{q}_{\text{sel}}$ to IRIs in the RDF graph, as depicted in Figure 1(b), where each row represents a mapping.

Returning mappings instead of tuples might appear just as a slight difference between SPARQL and other query languages such as SQL, but it is known to lead to several complications (see, e.g., [28,32]). For example, when studying the expressive power of SPARQL in [2, 32], the authors need some rather technical machinery to be able to even compare SPARQL with relational query languages. The result is that, even if we now know that the SELECT fragment of SPARQL is equivalent in expressive power to relational algebra, this is

[1] In this paper we follow the SPARQL syntax of [28], in particular, we shorten `OPTIONAL` to OPT.

shown using proofs that are much more complicated than other similar results in database theory, and it has been difficult to build upon this proofs to produce new results.

There are also practical consequences: while recursive queries have been part of SQL for more than twenty years, we are still left without a comprehensive operator to define recursive queries in SPARQL (SPARQL 1.1 includes the property paths primitive [40], but this additional feature is very restrictive in expressing recursive queries [23]).

However, this complication is relevant only to the SELECT queries of SPARQL, which have been considered in the theoretical literature almost exclusively. But there is also a class of queries that output RDF graphs, namely the class of CONSTRUCT queries. The following example illustrates how a user can specify such a query.

▶ **Example 2.** Let $\boldsymbol{q}_{\text{cons}}$ be the following SPARQL CONSTRUCT query:

```
CONSTRUCT {(?n, works_at, ?w), (?n, mbox, ?e)}
WHERE (
((?p, name, ?n) AND (?p, works_at, ?w))
         OPT   (?p, mbox, ?e)).
```

This query has the same WHERE clause as $\boldsymbol{q}_{\text{sel}}$, but the form of the output is different. The RDF graph resulting from the evaluation of this query over the dataset G_{ex} is depicted in Figure 1(c).

CONSTRUCT queries in SPARQL shape the class of effective queries whose inputs and answers are RDF graphs, so it is conceivable that much more insight can be obtained by comparing them to well-established query languages. But rather surprisingly, and despite being an important part of the SPARQL standard, these queries have received almost no theoretical attention. This can be partially explained by the fact that, as the examples above suggest, the difference between these classes of queries might seem negligible. However, as we show in this paper, this resemblance is often deceptive, and in many cases the properties of these queries are different. For example, CONSTRUCT queries allow for blank nodes in the templates specifying the answer triples, which is a feature unavailable in SELECT queries. Trying to fill this gap, we conduct a thorough study of CONSTRUCT queries. We concentrate on the AND-UNION-OPT-FILTER fragment, which is the core of SPARQL [28].

The first question studied in the paper is the expressive power of CONSTRUCT queries. In particular, we show that if blank nodes are not allowed in the templates, then this language is equivalent in expressive power to first order logic. Furthermore, if the underlying graph patterns are enforced to belong to the class of well designed patterns (see [28]) then we obtain a correspondence with positive first order logic. If, in turn, blank nodes in templates are allowed, we establish that the expressive power of these queries is equivalent to that of a well known class of mappings in data exchange.

These expressivity results lead to important conclusions on the composability of the aforementioned classes of queries, that is, whether the composition of two queries can always be expressed by another query in the same class. We show that the fragments without blank nodes are composable, but if blank nodes are allowed in construct templates then this important property is lost.

We also obtain results on the computational complexity of the evaluation of such queries: for the blank-free language it is the same as for SELECT queries (PSPACE-complete), but for the well-designed sublanguage there is a difference—it is Σ_2^p-complete for the SELECT case ([22]), but drops to NP-complete in CONSTRUCT case.

Finally, the properties of CONSTRUCT queries allow us to develop an extension of SPARQL with a form of recursion that resembles that of SQL. This proposal unifies several formalisms for querying RDF data such as SPARQL 1.1 property paths [40], c-query answering over OWL 2 RL entailment regime [16, 20], navigational SPARQL [29], GraphLog [11], and TriAL [23]. We are also able to pinpoint the expressivity of this extension to SPARQL by comparing it with a fragment of Datalog.

Due to the space limitations, only ideas of most important proofs are exposed in the main body of this paper. Complete proofs shall be given in the full version of this paper.

2 Preliminaries

RDF Graphs and Datasets

RDF graphs can be seen as edge-labeled graphs where edge labels can be node themselves, and an RDF dataset is a collection of RDF graphs. Formally, let $\mathbf{I}$ and $\mathbf{B}$ be infinite pairwise disjoint sets of *IRIs* and *blank nodes*,[2] respectively, and $\mathbf{T} = \mathbf{I} \cup \mathbf{B}$ be the set of *terms*. Then an *RDF triple* is a tuple (s, p, o) from $\mathbf{T} \times \mathbf{I} \times \mathbf{T}$, where s is called the *subject*, p the *predicate*, and o the *object*. An *RDF graph* is a finite set of RDF triples, and an *RDF dataset* is a set $\{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$, where $G_0, \ldots, G_n$ are RDF graphs and $u_1, \ldots, u_n$ are distinct IRIs, such that the graphs G_i use pairwise disjoint sets of blank nodes. The graph G_0 is called *default graph*, and $G_1, \ldots, G_n$ are called *named graphs* with *names* $u_1, \ldots, u_n$, respectively. For a dataset D and IRI u we define $\mathsf{gr}_D(u) = G$ if $\langle u, G\rangle \in D$ and $\mathsf{gr}_D(u) = \emptyset$ otherwise. We also use $\mathcal{G}$ and $\mathcal{D}$ to denote the sets of all RDF graphs and datasets, correspondingly, as well as $\mathsf{blank}(S)$ to denote the set of blank nodes appearing in S, which can be a triple, a graph, etc.

SPARQL Syntax

SPARQL is the standard pattern-matching language for querying RDF datasets. Let $\mathbf{V}$ be an infinite set $\{?x, ?y, \ldots\}$ of *variables*, disjoint from $\mathbf{T}$. Similarly to $\mathsf{blank}(S)$, let $\mathsf{var}(S)$ denote the set of variables appearing in S. SPARQL *graph patterns* are recursively defined as follows:

1. a triple in $(\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V})$ is a graph pattern, called a *triple pattern*;
2. if P_1 and P_2 are graph patterns then $(P_1 \mathsf{\ AND\ } P_2)$, $(P_1 \mathsf{\ OPT\ } P_2)$, and $(P_1 \mathsf{\ UNION\ } P_2)$ are graph patterns, called AND-, OPT-, and UNION-*patterns*, correspondingly;
3. if P is a graph pattern and $g \in \mathbf{I} \cup \mathbf{V}$ then $(g \mathsf{\ GRAPH\ } P)$ is a graph pattern, called a GRAPH-*pattern*;
4. if P is a graph pattern and R is a filter condition then $(P \mathsf{\ FILTER\ } R)$ is a graph pattern, called a FILTER-*pattern*, where SPARQL *filter conditions* are constraints of the form:
   - $?x = u$, $?x = ?y$, $isBlank(?x)$ or $bound(?x)$ for $?x, ?y \in \mathbf{V}$ and $u \in \mathbf{I}$ (called *atomic constraints*[3]),
   - $\neg R$, $R_1 \wedge R_2$, or $R_1 \vee R_2$, for filter conditions R, R_1 and R_2.

The fragment of SPARQL graph patterns, as well as its generalisation to SELECT queries, has drawn most of the attention in the Semantic Web community. In this paper we concentrate on another class of queries, formalized next.

[2] For the sake of simplicity we do not consider literals, but all the results in this paper hold if we introduce them explicitly.

[3] We use a simplified list of SPARQL atomic constraints, for the complete one see [40].

A SPARQL CONSTRUCT query, or *c-query* for short, is an expression

$$\mathsf{CONSTRUCT}\ H\ \mathsf{WHERE}\ P,$$

where H is a set of triples from $(\mathbf{T} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{T} \cup \mathbf{V})$, called a *template*, and P is a graph pattern. We also distinguish c-queries without blank nodes in templates, called *blank-free*, and c-queries without GRAPH-subpatterns in their patterns, called *graph-free*. We use c-SPARQL to denote the class of all c-queries, and specify these restrictions with subscripts bf and gf for the blank- and graph-free subclasses. For instance, $\text{c-SPARQL}_{\mathsf{bf,gf}}$ denotes the class of blank-free and graph-free c-queries.

SPARQL Semantics

The semantics of graph patterns is defined in terms of *mappings*; that is, partial functions from variables $\mathbf{V}$ to terms $\mathbf{T}$. The *domain* $\mathsf{dom}(\mu)$ of a mapping μ is the set of variables on which μ is defined. Two mappings μ_1 and μ_2 are *compatible* (written as $\mu_1 \sim \mu_2$) if $\mu_1(?x) = \mu_2(?x)$ for all variables $?x$ in $\mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2)$. If $\mu_1 \sim \mu_2$, then we write $\mu_1 \cup \mu_2$ for the mapping obtained by extending μ_1 according to μ_2 on all the variables in $\mathsf{dom}(\mu_2) \setminus \mathsf{dom}(\mu_1)$.

Given two sets of mappings M_1 and M_2, the *join*, *union* and *difference* between M_1 and M_2 are defined respectively as follows:

$$\begin{array}{rcl} M_1 \bowtie M_2 & = & \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1,\ \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\}, \\ M_1 \cup M_2 & = & \{\mu \mid \mu \in M_1 \text{ or } \mu \in M_2\}, \\ M_1 \setminus M_2 & = & \{\mu_1 \mid \mu_1 \in M_1 \text{ and there is no } \mu_2 \in M_2 \text{ such that } \mu_1 \sim \mu_2\}. \end{array}$$

Based on these, the *left outer join* operation is defined as

$$M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2).$$

Given a dataset $D = \{G_0, \langle u_1, G_1\rangle, \dots, \langle u_n, G_n\rangle\}$, and a graph G among $G_0, \dots, G_n$, the *evaluation* $[\![P]\!]_G^D$ of a graph pattern P over D with respect to G is defined as follows:

1. if P is a triple pattern, then $[\![P]\!]_G^D = \{\mu : \mathsf{var}(P) \to \mathbf{T} \mid \mu(P) \in G\}$;
2. if $P = (P_1\ \mathsf{AND}\ P_2)$, then $[\![P]\!]_G^D = [\![P_1]\!]_G^D \bowtie [\![P_2]\!]_G^D$;
3. if $P = (P_1\ \mathsf{OPT}\ P_2)$, then $[\![P]\!]_G^D = [\![P_1]\!]_G^D ⟕ [\![P_2]\!]_G^D$;
4. if $P = (P_1\ \mathsf{UNION}\ P_2)$, then $[\![P]\!]_G^D = [\![P_1]\!]_G^D \cup [\![P_2]\!]_G^D$;
5. if $P = (g\ \mathsf{GRAPH}\ P')$, then

$$[\![P]\!]_G^D = \begin{cases} [\![P']\!]^D_{\mathsf{gr}_D(g)} & \text{if } g \in \mathbf{I} \\ \bigcup_{u \in \mathbf{I}} \left([\![P']\!]^D_{\mathsf{gr}_D(u)} \bowtie \{\mu_{g \mapsto u}\}\right) & \text{if } g \in \mathbf{V} \end{cases}$$

   where $\mu_{g \mapsto u}$ is the mapping with domain $\{g\}$ and where $\mu_{g \mapsto u}(g) = u$;
6. if $P = (P'\ \mathsf{FILTER}\ R)$, then $[\![P]\!]_G^D = \{\mu \mid \mu \in [\![P']\!]_G^D \text{ and } \mu \models R\}$, where a mapping μ *satisfies* a built-in condition R, denoted by $\mu \models R$, if one of the following holds:
   - R is $?x = u$, $?x \in \mathsf{dom}(\mu)$ and $\mu(?x) = u$; or
   - R is $?x = ?y$, $?x \in \mathsf{dom}(\mu)$, $?y \in \mathsf{dom}(\mu)$ and $\mu(?x) = \mu(?y)$; or
   - R is $isBlank(?x)$ and $?x \in \mathsf{dom}(\mu)$ and $\mu(?x) \in \mathbf{B}$; or
   - R is $bound(?x)$ and $?x \in \mathsf{dom}(\mu)$; or
   - R is a Boolean combination of other filter conditions and this combination is satisfied according to the usual notions of $\{\neg, \vee, \wedge\}$.

The evaluation $[\![P]\!]^D$ of a pattern P over a dataset D with default graph G_0 is $[\![P]\!]^D_{G_0}$.

Next we define the semantics of c-queries. We concentrate for now on the class c-SPARQL$_{\mathsf{bf}}$ of queries, and discuss the semantics for full c-SPARQL in Section 5. The answer $\mathsf{ans}(\boldsymbol{q}, D)$ of a c-query $\boldsymbol{q} = \mathsf{CONSTRUCT}\ H\ \mathsf{WHERE}\ P$ in c-SPARQL$_{\mathsf{bf}}$ over an input dataset D is defined as

$$\mathsf{ans}(\boldsymbol{q}, D) = \{\mu(t) \mid \mu \in [\![P]\!]^D, t \text{ is a triple in } H \text{ and } \mu(t) \text{ is well-formed}\},$$

Note that the well-formedness condition disallows triples with blank nodes in predicate positions. Next we provide an example to illustrate the use of the operators GRAPH and CONSTRUCT. See [5] for examples on the rest of the operators.

▶ **Example 3.** Let G and G_1 be the graphs depicted in Figure 1(a) and Figure 2, respectively. Suppose we want to query the dataset $D = \{G, \langle \text{country}, G_1\rangle\}$ to obtain a new graph with information about where workers live. This would be achieved by the next SPARQL CONSTRUCT query:

```
CONSTRUCT {(?name, lives_in, ?country)} WHERE (
(?worker, name, ?name) AND (?worker, works_at, ?university) AND
(country GRAPH (?university, in_country, ?country)) ).
```

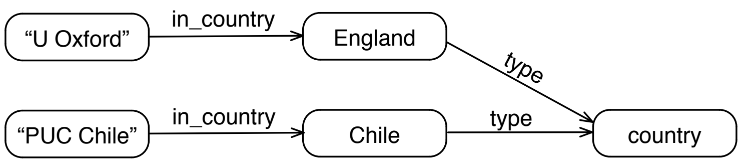


Figure 2 RDF graph containing information about location of universities.

3 Blank-free c-Queries

We start our study with c-SPARQL$_{\mathsf{bf}}$, the language of c-queries without blank nodes in their construct templates. This fragment has simple syntax and clear semantics, and it is of fundamental importance in our study. In particular, it resembles SPARQL SELECT queries in the sense that all the blank nodes in the answer graph of a c-SPARQL$_{\mathsf{bf}}$ query already appear in the input dataset.

The first problem we consider is the expressive power of c-SPARQL$_{\mathsf{bf}}$. As usual in databases our yardstick is first order logic (FO) with safe negation. However, since we are dealing with c-queries that input RDF graphs and datasets, it is only fair to compare them with FO over a signature that corresponds to these entities. Formally, we specify the following query language. Consider relational predicates $Default$, $Named$ and $IsBlank$, of arities 3, 4 and 1, respectively. Then the language FO$_{\mathsf{rdf}}$ consists of all well-formed ternary FO formulas over this signature. We always assume that the domain of FO structures is the set $\mathbf{T}$ of terms, and that for all structures we have that $IsBlank(b)$ holds for some b if and only if $b \in \mathbf{B}$, and $IsBlank(b)$ implies that none of $Default(a, b, c)$, $Named(b, a, c, d)$ and $Named(d, a, b, c)$ hold for any $a, c,$ and d. Thus the answers for this language are sets of triples from $\mathbf{T} \times \mathbf{I} \times \mathbf{T}$, essentially RDF graphs. Finally, the evaluation function for FO$_{\mathsf{rdf}}$ is the usual FO entailment $\models_{\text{adom}}$ over active domain semantics. This means that quantification is realized over the finite set of all the terms from $\mathbf{T}$ appearing in the input database and query (see [1] for formal definitions).

Note that the set of input databases of $\mathrm{FO}_{\mathsf{rdf}}$ have a straightforward one-to-one correspondence with the set of input datasets of c-SPARQL$_{\mathsf{bf}}$ queries, and the same holds for answers of queries in these languages. This allows us to compare their expressive power, for which we need the following definitions. A query language $\mathcal{Q}_1$ is *contained* in a language $\mathcal{Q}_2$ if and only if there are bijections $\mathsf{trans}_{\mathcal{I}} : \mathcal{I}_1 \to \mathcal{I}_2$, $\mathsf{trans}_{\mathcal{O}} : \mathcal{O}_1 \to \mathcal{O}_2$ between their input sets $\mathcal{I}_i$ and answer sets $\mathcal{O}_i$, and a function $\mathsf{trans}_{\mathcal{Q}} : \mathcal{Q}_1 \to \mathcal{Q}_2$ such that $\mathsf{trans}_{\mathcal{O}}(\mathsf{eval}_1(\boldsymbol{q}, I)) = \mathsf{eval}_2(\mathsf{trans}_{\mathcal{Q}}(\boldsymbol{q}), \mathsf{trans}_{\mathcal{I}}(I))$ holds for any $\boldsymbol{q} \in \mathcal{Q}_1$ and $I \in \mathcal{I}_1$, where eval_i are the evaluation functions of the languages. Two languages are *equivalent* if and only if they contain each other.

We are ready to present our first result, claiming that the language of blank-free construct queries is subsumed by first order logic.

▶ **Lemma 4.** *The language* c-SPARQL$_{\mathsf{bf}}$ *is contained in* $\mathrm{FO}_{\mathsf{rdf}}$.

To show this lemma one can use ideas similar to the ones presented in the reductions from the language of SPARQL SELECT queries to non-recursive Datalog with safe negation developed in [2] and [32]. Starting with a query Q, the idea of these reductions is to assemble an extensional predicate for each subpattern of Q in a way such that the evaluation of that predicate contains all the tuples that correspond to a mappings in the evaluation of the subpattern. Since some of the variables of these mappings may not be assigned, the undefined value is modelled by a special constant *Null*. We present a simpler reduction where *Null* is not used, but instead we create a predicate for each subset of the set of variables of Q. Avoiding predicate *Null* makes our proof much more simple and intuitive, and we make use of this proof to obtain several results in the following section.

Proof (idea). First we establish an equivalence between graph patterns and $\mathrm{FO}_{\mathsf{rdf}}$. Once this is done we just need to project out those variables that are not on the construct template and generate the corresponding triple. We do it as follows.

Given a graph pattern P, for every $X \subseteq \mathsf{var}(P)$ we construct a formula φ^P_X with X as free variables, such that a mapping μ is in $[\![P]\!]^D$ for a dataset D if and only if the variable assignment defined by μ satisfies $\varphi^P_{\mathsf{dom}(\mu)}$ in the $\mathrm{FO}_{\mathsf{rdf}}$ structure corresponding to D. Having such a formula for each set of variables makes it easier to define an inductive construction. We illustrate this construction with the translation of patterns P of the form $(P_1 \text{ AND } P_2)$. Consider, for every subset X of $\mathsf{var}(P)$, the formula

$$\varphi^P_X = \bigvee_{X_1 \subseteq \mathsf{var}(P_1), X_2 \subseteq \mathsf{var}(P_2), X_1 \cup X_2 = X} \varphi^{P_1}_{X_1} \wedge \varphi^{P_2}_{X_2},$$

where $\varphi^{P_i}_{X_i}$ are the formulas constructed on the previous inductive step.

Finally, the ternary formula $\varphi_{\boldsymbol{q}}$ producing, for every dataset D, the set of triples which correspond to the answer graph to the c-query $\boldsymbol{q} = \text{CONSTRUCT } H \text{ WHERE } P$ over D can be simply obtained from all φ^P_X by means of disjunction, existential quantification and checking that all the second arguments are not blank nodes. ◀

We illustrate this proof by means of the following example.

▶ **Example 5.** Recall the query $\boldsymbol{q}_{\mathrm{cons}}$ from Example 2. By simple inspection we see that the domain of every mapping in the evaluation of the graph pattern is either $\{?p, ?n, ?w\}$ or $\{?p, ?n, ?w, ?e\}$. Hence, we only need to construct a formula for each of these sets, as the formulas corresponding to other subsets of $\mathsf{var}(\boldsymbol{q}_{\text{cons}})$ will be unsatisfiable. Following the construction process, we obtain

$$\begin{aligned}
\varphi_{\{?p,?n,?w\}}(p,n,w) &= \mathit{Default}(p,\text{name},n) \wedge \mathit{Default}(p,\text{works\_at},w) \wedge \\
&\qquad\qquad \neg\exists e\ \mathit{Default}(p,\text{mbox},e), \\
\varphi_{\{?p,?n,?w,?e\}}(p,n,w,e) &= \mathit{Default}(p,\text{name},n) \wedge \mathit{Default}(p,\text{works\_at},w) \wedge \\
&\qquad\qquad \mathit{Default}(p,\text{mbox},e),
\end{aligned}$$

where '?' is omitted before variables to resemble the conventional FO notation. Having these, we need to create the formula $\varphi_{\boldsymbol{q}_{\text{cons}}}$ that always outputs exactly the same graph as $\boldsymbol{q}_{\text{cons}}$. As discussed above, this formula can be constructed by projecting out the non-relevant variables and checking that the triples are well-formed. In particular, we obtain

$$\begin{aligned}
\varphi_{\boldsymbol{q}_{\text{cons}}}(x,y,z) &= \neg \mathit{IsBlank}(y) \wedge \\
&\Big(\ \ \exists p,n,w \left[\varphi_{\{?p,?n,?w\}}(p,n,w) \wedge (x = n \wedge y = \text{works\_at} \wedge z = w)\right] \vee \\
&\quad\ \exists p,n,w,e \left[\varphi_{\{?p,?n,?w,?e\}}(p,n,w,e) \wedge (x = n \wedge y = \text{mbox} \wedge z = e)\right] \quad \Big).
\end{aligned}$$

Our next result is the inclusion in the other direction.

▶ **Lemma 6.** *The language* FO_{rdf} *is contained in* c-SPARQL$_{\mathsf{bf}}$.

Proof (idea). The proof of this lemma is an inductive construction that exploits the idea that the difference operation on mappings can be expressed in SPARQL by means of the following application of optional matching [28]. Let

$$P_1 \ \mathsf{MINUS}\ P_2 = (P_1\ \mathsf{OPT}\ (P_2\ \mathsf{AND}\ (?x_1, ?x_2, ?x_3)))\ \mathsf{FILTER}\ \neg \mathit{bound}(?x_1),$$

where $?x_1, ?x_2$ and $?x_3$ are mentioned neither in P_1 nor in P_2. It is readily verified that $[\![P_1\ \mathsf{MINUS}\ P_2]\!]_G^D = [\![P_1]\!]_G^D \setminus [\![P_2]\!]_G^D$ for any dataset D and its graph G. Our construction is again similar to the one in [2], where a reduction from non-recursive Datalog with safe negation to SPARQL graph patterns is provided. ◀

Having these lemmas at hand we conclude the following theorem.

▶ **Theorem 7.** *The languages* c-SPARQL$_{\mathsf{bf}}$ *and* FO_{rdf} *are equivalent in expressive power.*

This result and its proof have a couple of immediate important consequences. First of them is that the language c-SPARQL$_{\mathsf{bf,gf}}$ of graph-free and blank-free c-queries is equivalent to the fragment of FO_{rdf} which does not allow for the quaternary predicate Named (we use $\text{FO}_{\mathsf{rdf}}^{\text{ternary}}$ to denote this fragment). This comes from a straightforward inspection of the proofs of the previous lemmas.

▶ **Corollary 8.** *The languages* c-SPARQL$_{\mathsf{bf,gf}}$ *and* $\text{FO}_{\mathsf{rdf}}^{\text{ternary}}$ *are equivalent in expressive power.*

As a side remark we note that even if the syntax of manipulating graph names in SPARQL is very different from the syntax for manipulating subjects, predicates and objects, semantically the values are treated very similarly, and they are equivalent in terms of expressivity.

The second important consequence is that the blank-free fragment of c-SPARQL is composable. Formally, a query language $\mathcal{Q}$ with the same input and answer sets $\mathcal{I}$, and evaluation function eval, is *composable* if and only if for every pair of queries $\boldsymbol{q}_1, \boldsymbol{q}_2 \in \mathcal{Q}$ there is another query $\boldsymbol{q} \in \mathcal{Q}$ such that $\mathsf{eval}(\boldsymbol{q}_1, \mathsf{eval}(\boldsymbol{q}_2, I)) = \mathsf{eval}(\boldsymbol{q}, I)$ for any input database $I \in \mathcal{I}$. According to this definition, it makes no sense to talk about composability of the language c-SPARQL$_{\mathsf{bf}}$ of blank-free c-queries, because it does not satisfies the condition that the sets of inputs and answers coincide. But queries in c-SPARQL$_{\mathsf{bf,gf}}$ enjoy such a property, and we can obtain composability of c-SPARQL$_{\mathsf{bf,gf}}$ from composability of $\mathrm{FO}^{\mathrm{ternary}}_{\mathsf{rdf}}$.

▶ **Corollary 9.** *The language* c-SPARQL$_{\mathsf{bf,gf}}$ *is composable.*

We conclude this section with the complexity of the evaluation of blank-free c-queries. The lower bounds of the following result carries almost verbatim from the lower bounds in [28]. The upper bound is also very similar, the only additional step is guessing the values of all the variables which are not mentioned in the template.

▶ **Proposition 10.** *The problem of checking whether a triple is in the answer to a c-query from* c-SPARQL$_{\mathsf{bf}}$ *over a dataset is* PSPACE*-complete in general and in* NLOGSPACE *if the c-query is fixed.*[4] *The bounds hold also for c-queries from* c-SPARQL$_{\mathsf{bf,gf}}$.

Hence the complexity of evaluating blank-free c-queries is the same as that of SPARQL graph patterns, as well as of SPARQL SELECT queries.

4 OPT-free and Well-designed CONSTRUCT queries

The Semantic Web community has adopted the fragment of unions of well-designed graph patterns as a good practice for writing SPARQL queries. This is mainly because enforcing this property prevents users from writing graph patterns that do not agree with the open-world nature of the Semantic Web (see [28] for a more detailed discussion). Furthermore, restricting to unions of well-designed graph patterns drops the (combined) complexity of evaluation from PSPACE-complete to coNP-complete, and to Σ^p_2-complete if projection is allowed. Also, several optimization techniques have been developed for the evaluation of well-designed queries (see [22, 28]). In this section we study the properties of c-queries whose graph patterns are unions of well-designed patterns. We concentrate on the sublanguages of c-SPARQL$_{\mathsf{bf,gf}}$, leaving c-queries with blank nodes in templates for the next section, and restricting to graph-free c-queries for brevity. Note, however, that all the relevant results in this section hold also for c-queries with GRAPH-patterns.

We start with the definition of *well-designed* graph patterns, which are patterns using:

1. no UNION-subpatterns,
2. only FILTER-subpatterns (P FILTER R) such that all variables in R are mentioned in P,
3. only OPT-subpatterns (P_1 OPT P_2) such that all variables in P_2 which appear outside this subpattern are mentioned in P_1.

In this section we consider c-queries with graph patterns that are unions of well-designed patterns. We also consider c-queries without OPT-subpatterns, called *opt-free*. We will use superscripts uwd and of to specify sublanguages satisfying these restrictions, such as c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$. Note that c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ contains c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$, since although graph patterns in opt-free c-queries are not a union of well-designed patterns per se, they can be easily transformed into such by applying distributivity rules to push UNION outside and techniques of [2] to enforce the condition on FILTER subpatterns. Somewhat surprisingly, the next lemma shows that this containment holds in other direction as well, which means that adding well-designed OPT to patterns does not increase the expressive power of c-queries.

▶ **Lemma 11.** *The language* c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ *is contained in* c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$.

[4] The latter setting is known as *data complexity* of the problem (see [39]).

This result relies on the following fact: Consider a c-query CONSTRUCT H WHERE P in c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$. Then no triple in the answer to this query is generated by those mappings obtained from the evaluation of P in which some of the variables from H are not defined. This obviously does not hold for graph patterns themselves nor for SPARQL SELECT queries, which explains the importance of well-designed OPT in those classes of queries.

An important corollary from the proof of the previous lemma is that a c-query in c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ can be efficiently transformed into an opt-free c-query. In other words, in the context of c-queries well-designed OPT is not just dispensable, but also syntactic sugar.

▶ **Corollary 12.** *Every c-query from* c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ *can be transformed to an equivalent c-query from* c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$ *in* LOGSPACE.

We also relate the described languages to a fragment of first order logic, defined next. A formula $\varphi \in \mathrm{FO}^{\text{ternary}}_{\mathsf{rdf}}$ is $\exists$-*positive* if it is in the $\{\exists, \neg, \wedge, \vee\}$ fragment of FO where negation is atomic and only appear over equalities or $IsBlank$ atomic predicates. The language of all such $\exists$-positive formulas is denoted $\exists$pos-FO$^{\text{ternary}}_{\mathsf{rdf}}$.

The following result comes from a straightforward inspection of the reduction from c-SPARQL$_{\mathsf{bf}}$ to FO$_{\mathsf{rdf}}$ (Lemma 4), as the only way to generate negation over the $Default$ predicate is by means of OPT patterns.

▶ **Lemma 13.** *The language* c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$ *is contained in* $\exists$pos-FO$^{\text{ternary}}_{\mathsf{rdf}}$.

Quite similarly, an inspection of the proof of Lemma 6 shows that a transformation of an existential formula in which the predicate $Default$ only appears positively gives us a c-query which does not use the OPT operator. Note, however, that this c-query can have negations in the filter expressions.

▶ **Lemma 14.** *The language* $\exists$pos-FO$^{\text{ternary}}_{\mathsf{rdf}}$ *is contained in* c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$.

We can now state the main theorem of this section.

▶ **Theorem 15.** *The languages* c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$, c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$ *and* $\exists$pos-FO$^{\text{ternary}}_{\mathsf{rdf}}$ *are equivalent in expressive power.*

We obtain the composability of c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ as a corollary.

▶ **Corollary 16.** *The language* c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ *is composable.*

We conclude this section with the complexity of evaluation of c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$. As it is with expressive power, the complexity of evaluation for this fragment is lower than the complexity of evaluation of SELECT queries with well-designed patterns, which is, as already mentioned, Σ^p_2-complete.

▶ **Proposition 17.** *The problem of checking whether a triple is in the answer to a c-query from* c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ *over a dataset is* NP*-complete.*

5 c-Queries with Blank Nodes in Templates

In this section we study the properties of c-queries with blank nodes in templates. Like in the previous section we concentrate on c-SPARQL$_{\mathsf{gf}}$ queries, that is, c-queries that do not use GRAPH operator, and work with RDF graphs but not datasets. However, all relevant results of this section transfer easily to the full class of c-SPARQL queries.

In order to define the semantics of c-queries with blank nodes, for every template H and dataset D we fix a family $F(H, D)$ of *renaming functions*. This family contains, for every mapping μ from $\text{var}(H)$ to the domain of D, an injective function $f_\mu : \mathsf{blank}(H) \to \mathbf{B} \setminus \mathsf{blank}(D)$. These functions must have pairwise disjoint ranges.

Then the *answer* $\mathsf{ans}(\boldsymbol{q}, D)$ to a c-query $\boldsymbol{q} = \mathsf{CONSTRUCT}\ H\ \mathsf{WHERE}\ P$ over an input dataset D is the RDF graph

$$\mathsf{ans}(\boldsymbol{q}, D) = \{\mu(f_\mu(t)) \mid \mu \in [\![P]\!]^D, t \text{ is a triple in } H \text{ and } \mu(f_\mu(t)) \text{ is well formed}\},$$

where f_μ is the corresponding renaming function for μ in $F(H, D)$.

▶ **Example 18.** Recall again the dataset from Figure 1 and consider the c-query

```
CONSTRUCT {(_:b, manages, ?n), (?n, mbox, ?e)}
WHERE (
((?p, name, ?n) AND (?p, works_at, ?w))
        OPT   (?p, mbox, ?e)),
```

where `_:b` is a blank node. This blank node is intended to create a new blank node for each person, representing his manager. However, one must be cautious: the semantics of blank nodes in c-queries creates one blank node per each of the mappings in the evaluation of the inner pattern, and thus two blank nodes are created for Cristian, since there are two different mappings that assign Cristian to $?w$. Recall that the evaluation of the inner pattern of this query over the graph G_{ex} of Figure 1 is the set of mappings $\{\mu_1, \mu_2, \mu_3\}$, according to Figure 1. If we set f_{μ_1}(`_:b`) =`_:b1`, f_{μ_2}(`_:b`) =`_:b2` and f_{μ_3}(`_:b`) =`_:b3` then the result of evaluating the query above over G_{ex} contains the triples `(_:b1,manages,Fran)`, `(_:b2,manages,Cristian)`,`(_:b3,manages,Cristian)` and `(Cristian,mbox,cris@puc.cl)`.

In order to understand the properties of c-SPARQL$_{\mathsf{gf}}$, we start with the study of its expressive power. Since queries from this class can create values from scratch, it does not make much sense to compare them with FO queries. Instead, we focus on the resemblance between the semantics of blank nodes in c-SPARQL$_{\mathsf{gf}}$ queries and the one of nulls in universal solutions for data exchange problems (see [3] for a good introduction to the topic). In the following we show that this resemblance is not a coincidence, since all c-queries in c-SPARQL$_{\mathsf{gf}}$ can be simulated by source-to-target dependencies in the context of data exchange, in the following sense: Given a c-query q in c-SPARQL$_{\mathsf{gf}}$, one can construct a data exchange setting such that the graph created by posing q over an RDF graph corresponds to the result of exchanging this graph according to the data exchange setting (up to renaming of blank nodes). This establishes that these two formalisms are, in a way, equivalent in expressive power, and one of the most important consequences of this result is the non-composability of queries in c-SPARQL$_{\mathsf{gf}}$, in contrast to the blank-free c-queries from the previous sections.

To state these results we recall some terminology on data-exchange. We begin by adapting the definitions of [3, 13] to our context[5]. A *dependency* is an expression of the form

$$\forall \bar{x}\, \forall \bar{y}\, (\varphi(\bar{x}, \bar{y}) \to \exists \bar{z}\, \psi(\bar{x}, \bar{z}))\,, \tag{1}$$

[5] Our definition differs slightly from the chase for RDF used in [10], but it follows the same spirit.

where $\bar{x}, \bar{y}$, and $\bar{z}$ are disjoint tuples of variables and φ and ψ are first-order formulas. We concentrate on a restricted class of dependencies, that we call *source-to-target dependencies* (or *st-dependencies*), which are those in which φ belongs to $\mathrm{FO}_{\mathsf{rdf}}^{\mathrm{ternary}}$ (i.e., formulas over *Default* and *IsBlank* relations), and ψ is a conjunction of atoms over another ternary relation *OTriple*. In data exchange terminology, sets of st-dependencies are usually known as "mappings", but since we already use this term for solutions of graph patterns, we call them *de-mappings*, and denote $\mathrm{DE}_{\mathsf{rdf}}$ the language of de-mappings.

The semantics of a de-mapping Σ in our context can be defined as follows. A first-order structure over *OTriple* (called a *target instance*) is a *solution under* Σ for a structure over *Default* and *IsBlank* (called a *source instance*), if the set Σ of dependencies holds in the union of the source and target instances (recall that the domain of our structures is always the set $\mathbf{T}$ of terms).

In data exchange one is usually interested in computing *universal solutions* for a source instance and a de-mapping Σ. These are solutions that have homomorphic images to all solutions. A typical way to compute universal solutions is by means of the chase procedure. In traditional data exchange settings, this procedure repeatedly tests and enforces the satisfaction of all dependencies that are not satisfied, instantiating each existential variable in the right hand side of dependencies with a fresh *null* value. These nulls have very similar semantics to the semantics of blank nodes in SPARQL settings, so we define the chase using blanks.

Next we define the *chase* of a source instance S under a de-mapping Σ as a target instance constructed by sequentially adding triples to *OTriple*. To compute this instance, for every st-dependency of the form (1) in Σ proceed as follows. Take every assignment $\pi : \bar{x} \cup \bar{y} \to \mathbf{T}$ such that $S \models_{\mathrm{adom}} \varphi(\pi(\bar{x}), \pi(\bar{y}))$ and extend π by assigning a distinct fresh blank node from $\mathbf{B}$ to each variable in $\bar{z}$. Then add to the target instance the fact $\mathit{OTriple}(\pi(v_1), \pi(v_2), \pi(v_3))$ for each conjunct $\mathit{OTriple}(v_1, v_2, v_3)$ in ψ, as long as $\pi(v_2)$ is not a blank node.

The result of the chase is deterministic up to renaming of the introduced blank nodes, so we can consider $\mathrm{DE}_{\mathsf{rdf}}$ as a query language with answers being the results of the chase. This enables us to compare the expressive power of c-queries with that of data exchange settings.

▶ **Theorem 19.** *The languages* c-SPARQL$_{\mathsf{gf}}$ *and* $\mathrm{DE}_{\mathsf{rdf}}$ *are equivalent in expressive power.*

It is known that de-mappings are not composable in the data exchange scenario [14]. We can adapt this argument into our context to obtain the following important negative result.

▶ **Proposition 20.** *The language* c-SPARQL$_{\mathsf{gf}}$ *is not composable.*

Next we refine the results above for the language c-SPARQL$_{\mathsf{gf}}^{\mathsf{uwd}}$. Since we have shown that such queries are equivalent to positive FO, it would be reasonable to guess that c-SPARQL$_{\mathsf{gf}}^{\mathsf{uwd}}$ is equivalent to the query language given by de-mappings where every dependency (1) is such that the formula φ is a conjunction of atoms. This last language is, in fact, a very well studied class of de-mappings, called *GLAV-mappings* (see, e.g., [13, 21]), and are denoted by $\mathrm{GLAV}_{\mathsf{rdf}}$ in this paper.

Unfortunately, the following example shows that the previous intuition is not correct.

▶ **Example 21.** Consider the c-query

```
CONSTRUCT {(_:b, p, ?x), (_:b, p, ?y)}
WHERE ((?x, p, a) OPT (?x, p, ?y)).
```

Note that here the same blank needs to be added to both of the triples in the template whenever a mapping that bounds both $?x$ and $?y$ exists. However, we also need to account for mappings that bind only $?x$. Hence, this c-query is not equivalent to the de-mapping

$$\begin{aligned} \forall x \forall y \big(Default(x,p,a) \wedge Default(x,p,y) \quad &\rightarrow \quad \exists z\, (OTriple(z,p,x) \wedge OTriple(z,p,y))\big), \\ \forall x \big(Default(x,p,a) \quad &\rightarrow \quad \exists z\, OTriple(z,p,x)\big), \end{aligned}$$

because it creates additional blank nodes whenever the same pair of IRIs witnesses both dependencies. In fact, one can show that this c-query is not equivalent to any query in $\text{GLAV}_{\mathsf{rdf}}$.

In the above example both the chase of the de-mapping and the answer to the CONSTRUCT query are *homomorphically equivalent*, in the sense of [19]. This correspondence is not accidental, and an equivalence between $\text{GLAV}_{\mathsf{rdf}}$ and $\text{c-SPARQL}^{\mathsf{uwd}}_{\mathsf{gf}}$ can be shown under this relaxed notion of equivalence between RDF graphs. We omit these results for space reasons, but we can show the following containment without introducing any additional notation.

▶ **Proposition 22.** *The language* $\text{GLAV}_{\mathsf{rdf}}$ *is contained in* $\text{c-SPARQL}^{\mathsf{uwd}}_{\mathsf{gf}}$.

Regardless, the following corollary is a consequence of the proof of Proposition 20. This again goes in line with results for data exchange, since *GLAV-mappings* are not composable in general.

▶ **Corollary 23.** *The language* $\text{c-SPARQL}^{\mathsf{uwd}}_{\mathsf{gf}}$ *is not composable.*

We conclude this section with the following observation. Example 21 is problematic as we use the same blank nodes in two triples in the CONSTRUCT template. If we disallow blank nodes we regain equivalence between c-queries and data exchange settings, since $\text{c-SPARQL}^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ is equivalent to the setting given by *GAV-mappings*, that is, GLAV-mappings of the form (1) but without existential variables.

6 Adding Recursion to SPARQL

Recursion is an integral part of most practical query languages such as SQL:1999 [35]. The recent version 1.1 of SPARQL also allows for some form of recursion, namely *property paths* [40], a binary primitive based on two way regular path queries, a well-known query language for graph databases [6]. However, as shown in e.g. [23, 29], this recursion is limited and cannot express several interesting and relevant queries. It is possible to simulate more recursion by exploiting the power of *entailment regimes* like OWL 2 RL [16]. But to put it simple, this formalism is also quite limited and, more important, not part of SPARQL 1.1 itself. The aim of this section is to develop syntax and semantics for a full recursive operator in SPARQL and study its properties.

Before starting the formal development we discuss what are the difficulties of introducing recursion in SPARQL. The semantics in the majority of query languages that allow for recursion is defined in terms of a fixed point operator. But to have such operator one needs to be able to pose a query over the result of another query, that is, the query language must have the same input and answer domains. Hence it is not possible to introduce recursion based on SPARQL SELECT queries: they evaluate over a dataset, but answer a set of mappings. On the contrary, we can do it for c-queries, since the output and input of these queries are RDF graphs.

In this section we show how to apply our study of c-queries in the development of a recursive operator for SPARQL. Our proposal resembles the syntax and semantics of such an operator in the SQL:1999 standard. Let us explain our proposal by means of an example.

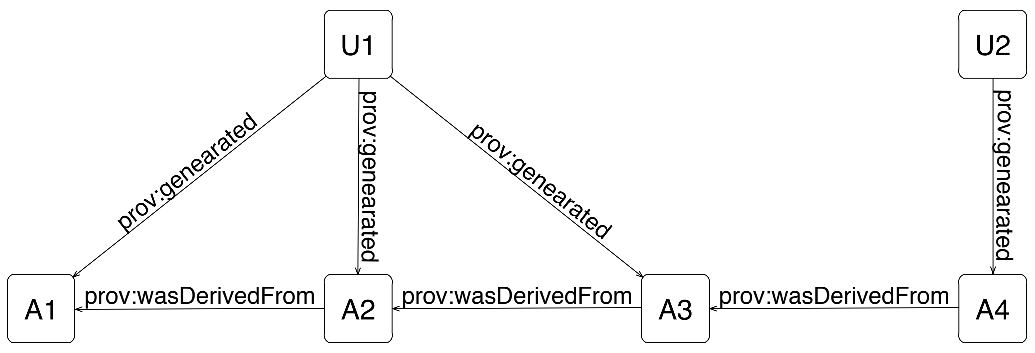


Figure 3 RDF graph storing provenance history of Wikipedia articles A1, A2, A3 and A4.

▶ **Example 24.** Consider the RDF graph in Figure 3, where a piece of the provenance information about the history of Wikipedia pages is depicted, according to PROV data model (see [26]). New articles are derived from their previous versions, and each version is linked to the user that was responsible for its generation. When inspecting this graph, one of the things we may be interested in is to obtain all triples of the form $(A, \text{same:user}, A')$ such that A' is an article derived from A by a path of revisions generated by the same user. For example, in the graph of Figure 3, we would want to obtain triple $(A3, \text{same:user}, A1)$, among others. In SPARQL we propose to obtain all such triples by means of the following query:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS
{
  CONSTRUCT {(?x, ?u, ?y)}
  WHERE
   ((?x, prov:wasDerivedFrom, ?y) AND
       (?u,prov:generated,?x) AND (?u,prov:generated,?y))
   UNION
   ((?x, prov:wasDerivedFrom, ?z) AND (?u,prov:generated,?x) AND
       (http://db.ing.puc.cl/temp GRAPH (?z, ?u, ?y)))
}
CONSTRUCT (?x, same:user, ?y)
WHERE (http://db.ing.puc.cl/temp GRAPH (?x, ?u, ?y))
```

The intention of this query is as follows. The first line is the actual fixed point operator: it specifies that the RDF graph `http://db.ing.puc.cl/temp` is a temporal graph, which is iteratively computed until the least fixed point of the subsequent query is reached. In this example, the iterated query in braces states that all the triples in `http://db.ing.puc.cl/temp` are of the form (X, U, Y), where Y is either a revision of X or is linked to X via a chain of revisions of arbitrary length, but in which all revisions involved were generated by user U. Finally, the CONSTRUCT part of the query in the end extracts the desired information from the computed temporal graph `http://db.ing.puc.cl/temp` into the output graph.

In this example, as in the rest of the paper, we deal with CONSTRUCT queries, but of course nothing prevents the main subquery to be of any other form (for example, one could retrieve mappings using SELECT in the main subquery) We also concentrate on blank-free c-queries and leave the study of recursive c-queries with blank nodes in the templates for future work.

▶ **Definition 25.** A *recursive c-query* is either a blank-free c-query from c-SPARQL$_{\mathsf{bf}}$ or an expression of the form

$$\text{WITH RECURSIVE } t \text{ AS } \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2, \tag{2}$$

where t is an IRI from $\mathbf{I}$, $\boldsymbol{q}_1$ is a c-query from c-SPARQL$_{\mathsf{bf}}$, and $\boldsymbol{q}_2$ is a recursive c-query. The set of all recursive c-queries is denoted rec-c-SPARQL$_{\mathsf{bf}}$.

We reinforce the idea that in this definition $\boldsymbol{q}_1$ is non-recursive, but $\boldsymbol{q}_2$ could be recursive by itself, which allows us to compose recursive definitions.

Having the syntax at hand we define the semantic of recursive c-queries. Given datasets D, D' with default graphs G_0 and G_0', we define $D \cup D'$ as the dataset with the default graph $G_0 \cup G_0'$ and $\mathsf{gr}_{D\cup D'}(u) = \mathsf{gr}_D(u) \cup \mathsf{gr}_{D'}(u)$ for any URI u.

▶ **Definition 26.** If a recursive query $\boldsymbol{q}$ from rec-c-SPARQL$_{\mathsf{bf}}$ is a simple c-query, then its answer $\mathsf{ans}(\boldsymbol{q}, D)$ over a dataset D is defined according to the semantics of c-queries. Otherwise, that is, if $\boldsymbol{q}$ is of the form (2), then the *answer* $\mathsf{ans}(\boldsymbol{q}, D)$ is equal to $\mathsf{ans}(\boldsymbol{q}_2, D_{\text{lfp}})$, where D_{lfp} is the least fixed point of the sequence $D_0, D_1, \ldots$ with $D_0 = D$ and

$$D_{i+1} = D \cup \{\langle t, \mathsf{ans}(\boldsymbol{q}_1, D_i)\rangle\}, \text{ for } i \geq 0.$$

Naturally, the above definition makes sense only when the sequence $D_0, D_1, \ldots$ has a (finite) fixed point. In this case, we say that the answer $\mathsf{ans}(\boldsymbol{q}, D)$ is *well-defined.* By our results on expressive power, one way to guarantee this is to require graph pattern of the c-query $\boldsymbol{q}_1$ to be a union of well-designed patterns, since this implies that the sequence is monotone. However, we can partially relax this condition and concentrate on the following fragment of rec-c-SPARQL$_{\mathsf{bf}}$. A recursive c-query $\boldsymbol{q}$ is *semi-positive* iff it is either a simple c-query, or it is of the form (2) with $\boldsymbol{q}_2$ semi-positive and every subpattern P in $\boldsymbol{q}_1$ satisfying the following conditions:

1. if P is (g GRAPH P') with $g \in \mathbf{V} \cup \{t\}$ then P' is well-designed, and
2. if P is (P_1 OPT P_2) then all subpatterns (g GRAPH P') of P_2 are such that $g \in \mathbf{I} \setminus \{t\}$.

The language of all semi-positive recursive c-queries is denoted by rec-c-SPARQL$_{\mathsf{bf}}^{\mathsf{semi}}$. They always have fixed points, as desired.

▶ **Proposition 27.** *For every recursive c-query* $\boldsymbol{q}$ *in* rec-c-SPARQL$_{\mathsf{bf}}^{\mathsf{semi}}$ *and dataset* D *the answer* $\mathsf{ans}(\boldsymbol{q}, D)$ *is well-defined.*

Next we study its expressive power of our language and show that it is equivalent to a particular class of Datalog programs (see [1] for a good introduction on Datalog).

Let V be a vocabulary of relational predicates. A *rule* is an expression of the form

$$Pr(\bar{x}) \leftarrow \varphi(\bar{x}, \bar{y}),$$

where $\bar{x}$ and $\bar{y}$ are tuples of variables, Pr is a predicate from $V \cup \{OTriple\}$, and φ is a conjunction of positive and negated atoms (including equalities) over $\mathbf{I} \cup V \cup \{Default, Named\}$, such that every variable from $\bar{x}$ appears in φ. In such a rule, $Pr(\bar{x})$ is the *head* and $\varphi(\bar{x}, \bar{y})$ is the *body*.

A *Datalog program with rule-by-rule stratification* is a sequence $\Pi_1, \ldots, \Pi_n$ of sets of rules for which there exist a set $V = \{Pr_1, \ldots, Pr_n\}$ such that the following holds:

1. the head of each rule in Π_i is the predicate Pr_i;
2. each Π_i does not mention any Pr_j with $j > i$;
3. each Π_i does not mention Pr_i in negated atoms.

Without loss of generality we assume that $Pr_n = OTriple$. The language of all Datalog programs with rule-by-rule stratification is denoted by $\text{Datalog}^{\mathsf{rbr}}_{\mathsf{rdf}}$. The semantics of these programs over structures in the signature of FO_{rdf} is the standard fixed point semantics (see, e.g., [1] for a formal definition).

▶ **Theorem 28.** *The languages* $\text{rec-c-SPARQL}^{\mathsf{semi}}_{\mathsf{bf}}$ *and* $\text{Datalog}^{\mathsf{rbr}}_{\mathsf{rdf}}$ *are equivalent in expressive power.*

We conclude this section with some discussion on the relationship of the semi-positive recursive SPARQL with other known formalisms. First, from the last theorem and the results of [11] we may conclude that $\text{rec-c-SPARQL}^{\mathsf{semi}}_{\mathsf{bf}}$ contains first order logic with transitive closure. Second, it is a technicality to check that this query formalism contains SPARQL 1.1 property paths [40], c-query answering over OWL 2 RL entailment regime [16], the algebra defined in [36], navigational SPARQL [29], as well as GraphLog [11] and TriAL [23] query languages. Also, it is possible to show that none of these formalisms can express all $\text{rec-c-SPARQL}^{\mathsf{semi}}_{\mathsf{bf}}$ queries, that is, the containment is strict in all the cases. Hence, we may conclude that $\text{rec-c-SPARQL}^{\mathsf{semi}}_{\mathsf{bf}}$ is a clean unification of all these languages, and, as we believe, it deserves a further dedicated studies, both theoretical and applied.

7 Conclusions and Future Work

We presented a thorough study of the expressive power and complexity of evaluation of SPARQL CONSTRUCT queries. By studying these queries we provided a strong bridge between SPARQL and well-developed frameworks such as first-order logic and Datalog. In particular, we gave a clean proof of the equivalence between CONSTRUCT queries without blank nodes and first-order logic, characterized well-designed CONSTRUCT queries by a reduction into a positive fragment of first-order logic, and presented a translation between the full fragment of CONSTRUCT queries and a specific setting for data exchange. Finally, having a good understanding of these queries we were able to present a proposal for extending SPARQL with recursion, which we proved to be equivalent in expressive power to Datalog with rule-by-rule stratification.

CONSTRUCT queries are an important fragment of SPARQL since they provide the standard language for querying RDF to produce RDF as output. Query languages with this property have several advantages, such as allowing for composability and recursion. The results in this paper present a first formal study of this fragment, and we believe the Semantic Web community will take good advantage of them. As future work we would like to extend our results to advance in the understanding of more expressive versions of SPARQL. There is still a good deal of research to be done in characterizing CONSTRUCT queries allowing for blank nodes in the template, as well as studying c-queries allowing for advanced SPARQL 1.1 operators. It is also left as future work to implement the recursive fragment, and to develop and apply techniques for its optimization.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

2 Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In *ISWC*, pages 114–129, 2008.

3 Marcelo Arenas, Pablo Barcelo, Leonid Libkin, and Filip Murlak. Relational and XML data exchange. *Synthesis Lectures on Data Management*, 2(1):1–112, 2010.

4 Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st international conference on World Wide Web*, pages 629–638. ACM, 2012.

5 Marcelo Arenas and Jorge Pérez. Querying semantic web data with SPARQL. In *PODS*, pages 305–316, 2011.

6 Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd symposium on Principles of database systems*, pages 175–188. ACM, 2013.

7 Carlos Buil-Aranda, Marcelo Arenas, and Oscar Corcho. Semantics and optimization of the SPARQL 1.1 federation extension. In *The Semanic Web: Research and Applications*, pages 1–15. Springer, 2011.

8 Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. SPARQL query containment under $\mathcal{SHI}$ axioms. In *AAAI*, 2012.

9 Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. SPARQL query containment under RDFS entailment regime. In *IJCAR*, pages 134–148, 2012.

10 Rada Chirkova and George HL Fletcher. Towards well-behaved schema evolution. In *WebDB*, 2009.

11 Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 404–416. ACM, 1990.

12 Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

13 Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

14 Ronald Fagin, Phokion G Kolaitis, Lucian Popa, and Wang-Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Transactions on Database Systems (TODS)*, 30(4):994–1055, 2005.

15 Floris Geerts, Grigoris Karvounarakis, Vassilis Christophides, and Irini Fundulaki. Algebraic structures for capturing the provenance of SPARQL queries. In *ICDT*, pages 153–164, 2013.

16 Birte Glimm and Chimezie Ogbuji. SPARQL 1.1 Entailment Regimes. W3C Recommendation, 2013. Available at `http://www.w3.org/TR/sparql11-entailment/`.

17 Harry Halpin and James Cheney. Dynamic provenance for SPARQL updates. In *ISWC*, 2014.

18 Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered rdf store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, pages 94–109, 2009.

19 Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. Everything you always wanted to know about blank nodes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2014.

20 Egor V. Kostylev and Bernardo Cuenca Grau. On the semantics of SPARQL queries with optional matching under entailment regimes. In *ISWC*, 2014.

21 Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM, 2002.

22 Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.*, 38(4):25, 2013.

23 Leonid Libkin, Juan Reutter, and Domagoj Vrgoč. TriAL for RDF: adapting graph query languages for RDF data. In *Proceedings of the 32nd symposium on Principles of database systems*, pages 201–212. ACM, 2013.

24 Katja Losemann and Wim Martens. The complexity of evaluating path expressions in SPARQL. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 101–112. ACM, 2012.

25 Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

26 Paolo Missier, Khalid Belhajjame, and James Cheney. The w3c prov family of specifications for modelling provenance metadata. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 773–776. ACM, 2013.

27 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. In *ISWC*, pages 30–43, 2006.

28 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

29 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A navigational language for RDF. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):255–270, 2010.

30 François Picalausa and Stijn Vansummeren. What are real SPARQL queries like? In *SWIM*, 2011.

31 Reinhard Pichler and Sebastian Skritek. Containment and equivalence of well-designed SPARQL. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 39–50. ACM, 2014.

32 Axel Polleres and Johannes Peter Wallner. On the relation between SPARQL1.1 and answer set programming. *Journal of Applied Non-Classical Logics*, 23(1-2):159–212, 2013.

33 Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Recommendation, 2008. Available at `http://www.w3.org/TR/rdf-sparql-query/`.

34 Eric Prud'hommeaux, Andy Seaborne, et al. SPARQL query language for RDF. 2006.

35 Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

36 Edward L Robertson. Triadic relations: An algebra for the semantic web. In *Semantic Web and Databases*, pages 91–108. Springer, 2005.

37 Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4–33, 2010.

38 Andy Seaborne. ARQ-A SPARQL processor for Jena. *Obtained through the Internet: http://jena. sourceforge. net/ARQ/*, 2010.

39 Moshe Y Vardi. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146. ACM, 1982.

40 W3C SPARQL Working Group. SPARQL 1.1 Query language. W3C Recommendation, 21 March 2013. Available at `http://www.w3.org/TR/sparql11-query/`.

APPENDIX: Proofs and Intermediate Results

Proof of Lemma 4

The definition of $\mathsf{trans}_{\mathcal{I}}$ and $\mathsf{trans}_{\mathcal{O}}$ are straightforward, so we concentrate on the translation $\mathsf{trans}_{\mathcal{Q}}$ for queries.

We assume an arbitrary order $\leq$ in $\mathbf{V}$. Given $X \subseteq \mathbf{V}$, we denote by $\bar{X}$ the tuple containing the variables in X ordered under $\leq$. Furthermore, for every mapping μ, if $X \subseteq \mathrm{dom}(\mu)$ we denote by $\mu(\bar{X})$ the tuple that results from replacing every component of $\bar{X}$ by its image under μ. We abuse notation by indistinctly using FO and SPARQL variables.

▶ **Lemma 29.** *For every graph pattern P there is a set of formulae $\{\varphi_X(\bar{X})\}^P_{X\subseteq var(P)}$ such that, for every mapping μ and dataset D, it is the case that $\mu \in [\![P]\!]^D$ if and only if $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^P_X(\mu(\bar{X}))$ with $X = dom(\mu)$.*

Proof. Let P be a SPARQL pattern. We proceed by induction on the structure of P.

- Let P be a triple pattern. Since for every dataset D and mapping $\mu \in [\![P]\!]^D$ we have $\mathrm{var}(P) = \mathrm{dom}(\mu)$, define $\varphi^P_X(\bar{X})$ as a contradiction for every $X \subsetneq \mathrm{var}(P)$. For $X = \mathrm{var}(P)$ define $\varphi^P_X(\bar{X})$ as $Default(P)$, where P is considered as a first order tuple. As usual, we are assuming that every IRI can be referred to as a constant. It readily follows that a mapping μ belongs to $[\![P]\!]_G$ if and only if $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^P_{\mathrm{dom}(\mu)}(\mu(\bar{X}))$.
- Let $P = P_1$ UNION P_2. For every $X \subseteq \mathrm{var}(P)$ define the formula $\varphi^P_X(\bar{X})$ as
$$\varphi^P_X(\bar{X}) = \varphi^{P_1}_X(\bar{X}) \vee \varphi^{P_2}_X(\bar{X}).$$
Let μ be a mapping and $X = \mathrm{dom}(\mu)$. By semantics, $\mu \in [\![P]\!]^D$ if and only if $\mu \in [\![P_1]\!]^D \cup [\![P_2]\!]^D$. Hence, by hypothesis we have $\mu \in [\![P]\!]^D$ if and only if $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^{P_1}_X(\mu(\bar{X}))$ or $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^{P_2}_X(\mu(\bar{X}))$, which is the semantic definition of $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^{P_1}_X(\mu(\bar{X})) \vee \varphi^{P_2}_X(\mu(\bar{X}))$.
- Let $P = P_1$ AND P_2. For every $X \subseteq \mathrm{var}(P)$ define the formula $\varphi^P_X(\bar{X})$ as
$$\varphi^P_X(\bar{X}) = \bigvee_{X_1 \cup X_2 = X} \left[\varphi^{P_1}_{X_1}(\bar{X}_1) \wedge \varphi^{P_2}_{X_2}(\bar{X}_2)\right].$$
Let D and μ be a dataset and a mapping, respectively, and let $X = \mathrm{dom}(\mu)$. If μ belongs to $[\![P]\!]^D$, then there are two compatible mappings $\mu_1 \in [\![P_1]\!]^D$ and $\mu_2 \in [\![P_2]\!]^D$ such that $\mu = \mu_1 \cup \mu_2$. Let $X_1 = \mathrm{dom}(\mu_1)$ and $X_2 = \mathrm{dom}(\mu_2)$. By hypothesis, we know that $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^{P_1}_{X_1}(\mu_1(\bar{X}_1))$ and $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^{P_2}_{X_2}(\mu_2(\bar{X}_2))$ which is equivalent to $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^{P_1}_{X_1}(\mu_1(\bar{X}_1)) \wedge \varphi^{P_2}_{X_2}(\mu_2(\bar{X}_2))$. As $X_1 \cup X_2 = X$ we have $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^P_X(\mu(\bar{X}))$.
For the converse, if $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^P_{\mathrm{dom}(\mu)}(\mu(\bar{X}))$ then there are two sets X_1 and X_2 such that $X_1 \cup X_2 = \mathrm{dom}(\mu)$ and both $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^{P_1}_{X_1}(\mu(\bar{X}_1))$ and $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^{P_1}_{X_2}(\mu(\bar{X}_2))$ hold. Define μ_i as μ restricted to X_i ($i \in \{1, 2\}$). It follows from the hypothesis that $\mu_1 \in [\![P_1]\!]^D$ and $\mu_2 \in [\![P_2]\!]^D$. Since μ_1 and μ_2 are compatible and $\mu = \mu_1 \cup \mu_2$, this implies $\mu \in [\![P]\!]^D$.
- Let $P = P_1$ OPT P_2. For every $X \subseteq \mathrm{var}(P)$ define the formula $\varphi^P_X(\bar{X})$ as
$$\varphi^P_X(\bar{X}) = \varphi^{P_1 \mathsf{AND} P_2}_X(\bar{X}) \vee \varphi^P_{\mathsf{MINUS},X}(\bar{X})$$
Where
$$\varphi^P_{\mathsf{MINUS},X}(\bar{X}) = \left[\varphi^{P_1}_X(\bar{X}) \wedge \neg \bigvee_{X' \subseteq \mathrm{var}(P_2)} \exists (X' \setminus X)\varphi^{P_2}_{X'}(\bar{X}')\right].$$

Let D and μ be a dataset and a mapping, respectively, and let $X = \text{dom}(\mu)$. By definition, μ belongs to $[\![P_1 \ \mathsf{AND}\ P_2]\!]^D$ or to $[\![P_1]\!]^D \setminus [\![P_2]\!]^D$. In the first case, we know that $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_X^{P_1 \mathsf{AND} P_2}(\mu(\bar{X}))$. Next we show that if $\mu \in [\![P_1]\!]^D \setminus [\![P_2]\!]^D$ then $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^P_{\mathsf{MINUS},X}(\mu(\bar{X}))$. As $\mu \in [\![P_1]\!]^D$, we know $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_X^{P_1}(\mu(\bar{X}))$, so we only need to prove that there is no set $X' \subseteq \text{var}(P_2)$ such that $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_{X'}^{P_2}(\mu'(\bar{X}'))$ for some μ' compatible with μ. But if this was the case then μ' would be in $[\![P_2]\!]^D$, which contradicts the fact that $\mu \in [\![P_1]\!]^D \setminus [\![P_2]\!]^D$ (since μ' and μ are compatible).
For the converse, assume $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_X^P(\mu(\bar{X}))$ where $X = \text{dom}(\mu)$. If $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_X^{P_1 \mathsf{AND} P_2}(\mu(\bar{X}))$, we know by the AND case that $\mu \in [\![P_1 \ \mathsf{AND}\ P_2]\!]^D$ and hence $\mu \in [\![P]\!]^D$. The remaining case is when $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi^P_{\mathsf{MINUS},X}(\mu(\bar{X}))$. If this is the case, by hypothesis it readily follows that $\mu \in [\![P_1]\!]^D$. Now we have to prove that μ is not compatible with any mapping in $[\![P_2]\!]^D$. Proceed now by contrapositive. Assume there is a mapping $\mu' \in [\![P_2]\!]^D$ compatible with μ. We know $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_{X'}^{P_2}(\mu'(\bar{X}'))$ where $X' = \text{dom}(\mu')$. Since μ and μ' are compatible, the assignments in μ' can be obtained by extending those in μ, and thus $\mathsf{trans}_{\mathcal{I}}(D)$ would not satisfy $\varphi^P_{\mathsf{MINUS},X}(\mu(\bar{X}))$.

- Let $P = P_1 \ \mathsf{FILTER}\ R$. For every $X \subseteq \text{var}(P)$ define $\varphi_X^P(\bar{X})$ as

$$\varphi_X^P(\bar{X}) = \varphi_X^{P_1}(\bar{X}) \wedge \varphi_R(\bar{X})$$

where $\varphi_R(\bar{X})$ is inductively defined as follows:
  - If R is an equality and $\text{var}(R) \not\subseteq X$, then $\varphi_R = False$.
  - If R is an equality and $\text{var}(R) \subseteq X$, then $\varphi_R = R$.
  - If $R = isBlank(x)$ then $\varphi_R = Blank(x)$.
  - If $R = bound(x)$ and $x \notin X$ then $\varphi_R = False$.
  - If $R = bound(x)$ and $x \in X$ then $\varphi_R = True$.
  - If R is of the form $\neg R_1$, $R_1 \wedge R_2$, or $R_1 \vee R_2$ for filter conditions R_1 and R_2, then φ_R is the corresponding boolean combination of φ_{R_1} and φ_{R_2}.

Let D and μ be a dataset and a mapping, respectively, and let $X = \text{dom}(\mu)$. It is easy to see from the definition of φ_R that $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_R(\mu(\bar{X}))$ if and only if $\mu \models R$. By hypothesis we have $\mu \in [\![P_1]\!]^D$ if and only if $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_X^{P_1}(\mu(\bar{X}))$, and hence it readily follows that $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_X^{P_1}(\mu(\bar{X})) \wedge \varphi_R(\bar{X})$ if and only if $\mu \in [\![P_1]\!]^D$ and $\mu \models R$.

- Let $P = g \ \mathsf{GRAPH}\ P_1$. We distinguish two cases: $g \in \mathbf{V}$ or $g \in \mathbf{I}$. If $g \in \mathbf{I}$, for every $X \subseteq \text{var}(P)$ define $\varphi_X^P(\bar{X})$ as the result of replacing in $\varphi_X^{P_1}(\bar{X})$ every occurence of $Default(t_1, t_2, t_3)$ by $Named(g, t_1, t_2, t_3)$. In the other case, if $g \in \mathbf{V}$, we know that g will be in the domain of every mapping in $[\![P]\!]^D$. Hence, for every $X \subseteq \text{var}(P)$ define $\varphi^P_{X \cup \{g\}}$ as the result of replacing in $\varphi_X^{P_1}(\bar{X})$ every occurence of $Default(t_1, t_2, t_3)$ by $Named(g, t_1, t_2, t_3)$. The result readily follows from the induction hypothesis and the semantic definition of the GRAPH operator.

◀

Now we are ready to prove our main result. Let

$$Q = \mathsf{CONSTRUCT}\ \{t_1, \ldots, t_n\}\ \mathsf{WHERE}\ P,$$

and let $\{\varphi_X^P(\bar{X})\}_{X \subseteq \text{var}(P)}$ be the formulas obtained by the previous lemma. Let x, y, z be three variables not mentioned in any φ_X^P. We construct a formula $\varphi_Q(x, y, z)$ which is the exact translation of Q, this is, for every dataset D, $\mathsf{ans}(Q, D)$ is the set of triples (a, b, c) such that $\mathsf{trans}_{\mathcal{I}}(D) \models \varphi_Q(a, b, c)$. We first do this separately for every t_i, and then gather them by means of disjunction. Let $t \in \{t_1, \dots, t_n\}$. We denote the i-th entry of t by $t[i]$. For every $X \subseteq \text{var}(P)$ define the formula $\psi^P_{X,t}$ as

$$\psi^P_{X,t}(x,y,z) = \begin{cases} \exists X(\varphi^P_X \wedge x = t[1] \wedge y = t[2] \wedge z = t[3]) & \text{if } \text{var}(t) \subseteq X \\ x \neq x \text{ otherwise.} \end{cases}$$

It is easy to see that this formula outputs every triple generated by t and a mapping which domain is X, including those containing blank nodes as properties. Finally, we construct the promised formula as

$$\varphi_q(x,y,z) = \neg isBlank(y) \wedge \bigvee_{t \in \{t_1 \dots, t_n\}} \bigvee_{X \subseteq \text{var}(P)} \psi^P_{X,t}.$$

This formula outputs every well-formed triple in Q generated by some t and some mapping in the answer to P, which concludes our proof.

Proof of Lemma 6

The definition of $\mathsf{trans}_{\mathcal{I}}$ and $\mathsf{trans}_{\mathcal{O}}$ are straightforward, so we concentrate on the translation $\mathsf{trans}_{\mathcal{Q}}$ for queries. We first inductively define a translation from formulas φ, which satisfy all the conditions of FO_{rdf}, except, maybe, that it can be of arbitrary arity, to graph patterns P_φ. Since $IsBlank(b)$ holds for and only for b in $\mathbf{B}$, we may assume that φ does not contain atoms of the form $IsBlank(u)$ for $u \in \mathbf{T}$. Having

$$\begin{aligned} ADom^1(?x) \quad = \quad & ((?x, ?x_1^2, ?x_1^3) \text{ UNION } (?x_2^1, ?x, ?x_2^3) \text{ UNION } (?x_3^1, ?x_3^2, ?x) \text{ UNION} \\ & (?x \text{ GRAPH } (?x_4^2, ?x_4^3, ?x_4^4)) \text{ UNION } (?x_5^1 \text{ GRAPH } (?x, ?x_5^3, ?x_5^4)) \text{ UNION} \\ & \quad (?x_6^1 \text{ GRAPH } (?x_6^2, ?x, ?x_6^4)) \text{ UNION } (?x_7^1 \text{ GRAPH } (?x_7^2, ?x_7^3, ?x_7^4))) \end{aligned}$$

for fresh variables $?x_i^j$, and $ADom(?x_1, \dots, ?x_n) = (?y_1, ?y_2, ?y_3)$ AND $ADom^1(?x_1)$ AND $\dots$ AND $ADom^1(?x_n)$ for fresh $?y_i$, we define

1. if φ is $Default(t_1, t_2, t_3)$ then $P_\varphi = (t_1, t_2, t_3)$;
2. if φ is $Named(t, t_1, t_2, t_3)$ then $P_\varphi = (t \text{ GRAPH } (t_1, t_2, t_3))$;
3. if φ is $IsBlank(?x)$ then $P_\varphi = (ADom(?x) \text{ FILTER } isBlank(?x))$;
4. if φ is $?x = ?y$ then $P_\varphi = (ADom(?x, ?y) \text{ FILTER } ?x = ?y)$;
5. if φ is $?x = u$ then $P_\varphi = (ADom(?x) \text{ FILTER } ?x = u)$;
6. if φ is $\varphi_1 \wedge \varphi_2$ then $P_\varphi = (P_{\varphi_1} \text{ AND } P_{\varphi_2})$;
7. if φ is $\varphi_1 \vee \varphi_2$ then $P_\varphi = (P_{\varphi_1} \text{ UNION } P_{\varphi_2})$;
8. if φ is $\neg\varphi'$ then $P_\varphi = (ADom(\mathsf{free}(\varphi')) \text{ MINUS } P_{\varphi'})$;
9. if φ is $\exists ?x\, \varphi'$ then $P_\varphi = P_{\varphi'}$,

where $\mathsf{free}(\varphi)$ is the set of free variables of φ. Then we define $\mathsf{trans}_{\mathcal{Q}}(\varphi)$ for an FO_{rdf} formula $\varphi(?x, ?y, ?z)$ as CONSTRUCT $\{(?x, ?y, ?z)\}$ WHERE $(P_\varphi \text{ FILTER } \neg isBlank(?y))$.

Proof of Proposition 10

The PSPACE lower bound follows directly from the PSPACE lower bound for graph patterns using only AND, FILTER and OPT presented in [28]. In that reduction, given a quantified boolean formula φ they construct a SPARQL graph pattern P_φ, an RDF graph G_φ and a mapping μ such that φ is valid if and only if $\mu \in [\![P_\varphi]\!]_{G_\varphi}$. The key facts are that the mapping μ is independent of φ and, moreover, if $\mu \in [\![P_\varphi]\!]_{G_\varphi}$ then μ is the only mapping in $[\![P_\varphi]\!]_{G_\varphi}$ assigning $\mu(?x)$ to $?x$. Hence, given a quantified boolean formula φ, we create the c-query

$$Q_\varphi = \text{CONSTRUCT } \{(?x, ?x, ?x)\} \text{ WHERE } P_\varphi$$

where $?x \in \mathrm{dom}(\mu)$. Then it readily follows that φ is valid if and only if the triple $(\mu(?x), \mu(?x), \mu(?x))$ belongs to $\mathsf{ans}(Q_\varphi, \langle G_\varphi \rangle)$.

For the PSPACE upper bound, we extend the well-known fact that evaluation of SPARQL graph patterns is in PSPACE. Let T be a triple, $Q =$ CONSTRUCT H WHERE P a c-query in c-SPARQL$_{\mathsf{bf}}$ and D a dataset. To know if T belongs to $\mathsf{ans}(Q, D)$, we simply choose a triple $t \in H$ and a mapping μ such that $\mu(t) = T$. Then, we verify that $\mu \in [\![P]\!]^D$. All of the previous steps can be done in NPSPACE and hence in PSPACE [28].

For the NLOGSPACE upper bound let $Q =$ CONSTRUCT H WHERE P be a fixed c-query. Given an RDF cataset D and a triple T, to know if $T \in \mathsf{ans}(Q, D)$ we choose a triple $t \in H$ and a mapping μ such that $\mu(t) = T$ and then we check if $\mu \in [\![P]\!]^D$. Since the query Q is fixed, all the previous steps can be done in NLOGSPACE [28].

Proof of Lemma 11

We consider every pattern in this proof to be GRAPH-free. We need to prove that for every c-query Q in c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ there is a c-query in c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$ which is equivalent to Q. To do so, we recall some definitions from [28]. Given two graph patterns P and P', P' is said to be a direct reduction of P if P' can be obtained from P by replacing a subformula P_1 OPT P_2 by P_1. The reflexive and transitive closure of this relation is denoted by $\trianglelefteq$. For every pattern P, and(P) is the result of replacing in P every OPT by AND. Given a pattern P and an RDF graph G, a mapping μ is said to be a partial solution to P over G if there is a graph pattern P' such that $P' \trianglelefteq P$ and $\mu \in [\![\mathrm{and}(P')]\!]_G$.

We need to introduce some further notation. For every pattern P, define P_{of} as the result of replacing in P every subpattern P_1 OPT P_2 by P_1 UNION (P_1 AND P_2).

▶ **Proposition 30.** *For every RDF graph G and graph pattern P, $[\![P_{\mathsf{of}}]\!]_G$ is the set of partial solutions to P over G.*

Proof. We start by showing that every partial solution to P over G is in $[\![P_{\mathsf{of}}]\!]_G$. Let μ be a partial solution to P over G. We proceed by induction over P.

- If $P = P_1$ UNION P_2, then without loss of generality we can assume that μ is a partial solution to P_1 over G. Then μ belongs to $[\![P_{1\mathsf{of}}]\!]_G$, and hence to $[\![P_{1\mathsf{of}}$ UNION $P_{2\mathsf{of}}]\!]_G = [\![P_{\mathsf{of}}]\!]_G$.
- If $P = P_1$ AND P_2, then there are two mappings μ_1 and μ_2, with $\mu = \mu_1 \cup \mu_2$, which are partial solutions to P_1 and to P_2 over G, respectively. By hypothesis, $\mu_1 \in [\![P_{1\mathsf{of}}]\!]_G$ and $\mu_2 \in [\![P_{2\mathsf{of}}]\!]_G$. Hence, $\mu \in [\![P_{1\mathsf{of}}$ AND $P_{2\mathsf{of}}]\!]_G = [\![P_{\mathsf{of}}]\!]_G$.
- If $P = P_1$ FILTER R, then μ is a partial solution to P_1 which satisfies R. Hence, μ belongs to $[\![P_{1\mathsf{of}}$ FILTER $R]\!]_G = [\![P_{\mathsf{of}}]\!]_G$.
- If $P = P_1$ OPT P_2, either μ is a partial solution to P_1 over G or μ is a partial solution to P_1 AND P_2 over G. Then, we know by hypothesis that $\mu \in [\![P_{1\mathsf{of}}]\!]_G$ or $\mu \in [\![P_{1\mathsf{of}}$ AND $P_{2\mathsf{of}}]\!]_G$. In either case we have $\mu \in [\![P_{1\mathsf{of}}$ UNION $(P_{1\mathsf{of}}$ AND $P_{2\mathsf{of}})]\!]_G = [\![P_{\mathsf{of}}]\!]_G$.

Next we prove that every mapping $\mu \in [\![P_{\mathsf{of}}]\!]_G$ is a partial solution to P over G. Again we proceed by induction on P. If P is a triple, UNION-, AND- or FILTER-pattern, the result readily follows by the induction hypothesis as in the previous case. If $P = P_1$ OPT P_2, then we know that $P_{\mathsf{of}} = P_{1\mathsf{of}}$ UNION $(P_{1\mathsf{of}}$ AND $P_{2\mathsf{of}})$. Therefore, μ may either be in $[\![P_{1\mathsf{of}}]\!]_G$ or in $[\![P_{1\mathsf{of}}$ AND $P_{2\mathsf{of}}]\!]_G$. Hence, we know μ is a partial result to P_1 over G or to P_1 AND P_2 over G. In both cases, by definition it is a partial solution to P_1 OPT P_2 over G.

◀

We recall a proposition from [28], for which we need the next definition. Given two mappings μ and μ', μ' is said to be subsumed by μ if both $\mathrm{dom}(\mu') \subseteq \mathrm{dom}(\mu)$ and $\mu_1 \sim \mu_2$ hold. Given a set of mappings M, a mapping $\mu \in M$ is said to be maximal (in M) if there is no $\mu' \in M \setminus \{\mu\}$ subsuming μ.

▶ **Proposition 31** (Pérez, Arenas, Gutiérrez [28]). *Given a UNION-free well-designed pattern P and an RDF graph G, for every mapping μ it is the case that $\mu \in [\![P]\!]_G$ if and only if μ is a maximal partial solution for P over G.*

▶ **Lemma 32.** *For every well-designed pattern P and every RDF graph G, the maximal mappings in $[\![P]\!]_G$ are the same as the maximal mappings in $[\![P_{\mathsf{of}}]\!]_G$.*

Proof. Let μ be a maximal mapping in $[\![P]\!]_G$. In particular it is a maximal mapping in $[\![P']\!]_G$ for some UNION-free disjunct P' of P. By Proposition 31, μ is a partial solution to P' over G, and hence it is a partial solution to P over G. We conclude by Proposition 30 that $\mu \in [\![P_{\mathsf{of}}]\!]_G$.

Let μ be a maximal mapping in $[\![P_{\mathsf{of}}]\!]_G$. Since $[\![P_{\mathsf{of}}]\!]_G$ is the set of partial solutions to P over G (Proposition 30), we know μ is a maximal partial solution to P over G. Hence, it must be a maximal partial solution to P' over G for some disjunct P' of P. By Proposition 31 we conclude that $\mu \in [\![P']\!]_G$ and hence $\mu \in [\![P]\!]_G$. ◀

Now we have all the necessary ingredients to prove the main result. Let Q = CONSTRUCT H WHERE P be a c-query in c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$. Define Q_{of} as CONSTRUCT H WHERE P_{of}. We prove that for every rdf graph G it is the case that $\mathsf{ans}(Q, G) = \mathsf{ans}(Q_{\mathsf{of}}, G)$.

Let T be a triple in $\mathsf{ans}(Q_{\mathsf{of}}, G)$. We know there is a mapping $\mu \in [\![P_{\mathsf{of}}]\!]_G$ and a triple $t \in H$ such that (1) $\mathrm{var}(t) \subseteq \mathrm{dom}(\mu)$, (2) $\mu(t) = T$, and (3) $\mu(t)$ does not contain a blank node as second component. It is easy to see that every mapping μ' subsuming μ satisfies the previous three properties. But by Lemma 32 there must be a mapping μ' subsuming μ in $\mu \in [\![P]\!]_G$, and hence $T = \mu'(t) \in \mathsf{ans}(Q, G)$. The other direction is obtained by the exact same analysis but changing $\mathsf{ans}(Q, G)$ by $\mathsf{ans}(Q_{\mathsf{of}}, G)$.

Proof of Proposition 17

Since we can translate between c-SPARQL$^{\mathsf{uwd}}_{\mathsf{bf,gf}}$ and c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$ in LOGSPACE, we prove the property for c-queries in c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$. We rely on the well-known fact that the problem of, given an RDF dataset D, a mapping μ and an OPT-free graph pattern P, determiing if $\mu \in [\![P]\!]^D$, is NP-complete [27]. Let T be a triple, Q = CONSTRUCT H WHERE P a c-query in c-SPARQL$^{\mathsf{of}}_{\mathsf{bf,gf}}$ and D a dataset. To know if T belongs to $\mathsf{ans}(Q, D)$, we simply choose a triple $t \in H$ and a mapping μ such that $\mu(t) = T$. Then, we verify that $\mu \in [\![P]\!]^D$. By the previously mentioned result, all of the previous steps can be done in NP.

For the lower bound, we inspect the reduction used in [27]. In that reduction, given a propositional formula φ they construct an OPT-free SPARQL graph pattern P_φ, an RDF graph G_φ and a mapping μ_φ such that φ is satisfiable if and only if $\mu_\varphi \in [\![P_\varphi]\!]_{G_\varphi}$. The key fact is that if $\mu_\varphi \in [\![P]\!]_{G_\varphi}$ then $[\![P]\!]_{G_\varphi} = \{\mu_\varphi\}$, and if $\mu_\varphi \notin [\![P]\!]_{G_\varphi}$ then $[\![P]\!]_{G_\varphi} = \emptyset$. Hence, given a propositional formula φ, we create the c-query

$$Q_\varphi = \mathsf{CONSTRUCT}\ \{(?x, ?x, ?x)\}\ \mathsf{WHERE}\ P_\varphi$$

where $?x \in \mathrm{dom}(\mu_\varphi)$. Then it readily follows that φ is satisfiable if and only if the triple $(\mu_\varphi(?x), \mu_\varphi(?x), \mu_\varphi(?x))$ belongs to $\mathsf{ans}(Q_\varphi, \langle G_\varphi \rangle)$.

Proof of Proposition 19

Assume we have a query of form $C = \mathsf{CONSTRUCT}\ \{t_1, \ldots, t_n\}\mathsf{WHERE}P$. For each $V = \{x_1, \ldots, x_n\} \subseteq \mathrm{var}(P)$, let $\mathrm{var}(P) \setminus V = \{y_1, \ldots, y_m\}$, and T_V all triples mentioning only blanks and variables from V. Furthermore, let C_V be the following pattern:

$$
\begin{aligned}
C_V = {} & \mathsf{CONSTRUCT}\ T_V\ \mathsf{WHERE}\ P\ \mathsf{FILTER} \\
& ?x_1 = ?x_1\ \mathsf{AND} \cdots ?x_n = ?x_n\ \mathsf{AND}\ \mathit{NotBound}(y_1)\ \mathsf{AND} \cdots \mathsf{AND}\ \mathit{NotBound}(y_m) \qquad (3)
\end{aligned}
$$

We now claim

▶ **Lemma 33.** *For all RDF graph G, the graph $\mathsf{ans}(C, G)$ is equivalent to the union of $\{\mathsf{ans}(C_V, G) \mid V \subseteq var(P)\}$, up to renaming of blanks*

Proof. If a triple (a, b, c) belongs to $\mathsf{ans}(C, G)$, then there is a mapping $\mu \in [\![P]\!]_G$ and a triple $t \in \{t_1, \ldots, t_n\}$ such that $\mathrm{var}(t) \subseteq \mathrm{dom}(\mu)$ and $\mu(t) = (a, b, c)$. Let $V = \mathrm{dom}(\mu)$. Notice that t must belong to T_V, and μ satisfies the filter condition of C_V. Hence, (a, b, c) also belongs to $\mathsf{ans}(C_V, G)$.

The other direction is analogous: every mapping μ witnesses at most one such C_V, and all mappings witnessing any of the C_V's must also witness C. ◀

For each such C_V, construct FO formula $\varphi^P_V(\bar{V})$ as shown in Lemma 4. Furthermore, let $f_V : \mathbf{B} \cup \mathbf{V} \to \mathbf{V}$ be a function that is the identity on $\mathbf{V}$ and that replaces each blank in T_V for a fresh variable in $\mathbf{V}$.

Let ϕ_{T_V} be the conjunction of $Default(f(a), f(b), f(c))$ for each triple (a, b, c) in T_V.

Then Σ contains, for each $V = \{x_1, \ldots, x_n\} \subseteq \mathrm{var}(P)$, the dependency

$$\forall x_1 \cdots \forall x_n \big(\varphi^P_V(\bar{X}) \to \exists \bar{z} \phi_{T_V}\big),$$

where $\bar{z}$ is the tuple of all variables created by f_V.

From Lemma 33 and Lemma 4, it is easy to see that $\mathsf{ans}(C, G)$ is equivalent to the chase of Σ over G, for every RDF graph G.

For the other direction, we assume that no dependency in Σ has any variable in common. From each dependency d in Σ of form

$$\forall \bar{x}\ \forall \bar{y}\ (\varphi(\bar{x}, \bar{y}) \to \exists \bar{z}\ \psi(\bar{x}, \bar{z}))\,, \qquad (4)$$

We create a c-query C_d as follows. Replace each variable z in ψ for a fresh blank node, and let T_ψ be a set of triples containing a triple (a, b, c) for each conjunct $OTriple(a, b, c)$ of ψ. Moreover, let P_ϕ be the pattern corresponding to ϕ, albeit without the existential quantification of the variables, as described in the proof of Lemma 6. Then note that from Lemma 6 and the definition of chase, the query $\mathsf{CONSTRUCT}\ T_\psi \mathsf{WHERE}\ P_\phi$ is actually equivalent to the mapping Σ_d containing only d.

Once again from Lemma 33 it is easy to see that the union of all graphs in $\{\mathsf{ans}(C_d, G) \mid d \in \Sigma\}$ corresponds to the chase of Σ over G, as we have shown that each C_d actually corresponds to the chase of d over G. Since none of these C_ds have variables in common, we can merge all of them into a single pattern without altering their semantics.

Proof of Proposition 20

From Proposition 19 and a simple observation of the chase algorithm, we know that the number of triples that contain a particular blank node in the result of a c-pattern C over a graph G does not depend on the size of the graph, but rather on the amount of triples in the CONSTRUCT statement of C.

However, the following example shows construct patterns C_1 and C_2 and a family of RDF graphs $\{G_i \mid i > 1\}$ so that the number of triples connected to a blank node in $C_2(C_1(G_i))$ increases as we increase i. We should note that the example is heavily inspired in a classical result about composition of st-tgds in data exchange ([13]).

C-pattern C_1 is

```
CONSTRUCT{(?n, course, ?c), (?n, student, _:s)}
WHERE {(?n, takes, ?c)}
```

And C_2 is

```
CONSTRUCT{(?s, enrollment, ?c)}
WHERE{(?n, course, ?s) AND (?n, student, ?c)}
```

Intuitively, C_1 assigns, for each course $?n$ that the student takes, a new identifier for this student. Then, afterwards, C_2 takes the RDF graph constructed by C_1, and links each of the identifiers created for this student with each of the courses he's taking.

Thus, if one defines G_i as the RDF graph containing triples $\{(A, \mathit{takes}, B_1), \ldots, (A, \mathit{takes}, B_i)\}$, then $C_1(G_i)$ is the graph containing $\{(A, \mathit{course}, B_1), \ldots, (A, \mathit{course}, B_i)\} \cup \{(A, \mathit{student}, \_ : n_1), \ldots, (A, \mathit{student}, \_ : n_i)\}$. A simple inspection then reveals that each such blank $\_ : n_j$ forms part of triples $\{(B_1, \mathit{enrollment}, \_ : n_j), \ldots, (B_i, \mathit{enrollment}, \_ : n_j)\}$ in the graph $C_2(C_1(G_i))$.

Clearly, it is not possible to build a set of st-tgds Σ that when chasing over G_i produces precisely $C_2(C_1(G_i))$. The proof then follows from Proposition 19.

Proof of Proposition 22

Follows from Theorem 15 and an inspection of the proof of Proposition 19, because the constructed mapping is in this case an OPT-free mapping.

Proof of Proposition 27

For the proof it suffices to show the following.

▶ **Lemma 34.** *Let D and D' be two datasets that differ only on the graph named T, and if $\langle T, G\rangle$ belongs to D and $\langle T, G'\rangle$ belongs to D', then $T \subseteq T'$. Moreover, consider a semi-positive recursive query of form* WITH RECURSIVE T AS $\{q_1\}q_2$. *Then* $\mathsf{ans}((, q)_1, D) \subseteq \mathsf{ans}((, q)_1, D')$.

The proof of this lemma follows immediately from the proof of Lemma 6, since the semi-positiveness of q_1 guarantees that T appears only positively in the translation of q_1 over $\mathrm{FO}_{\mathsf{rdf}}$, and that no predicate $\mathit{Named}(a, b, c, d)$ appears negated if a is not a constant.

This establishes again that the operator that we are considering is monotone, and therefore a unique fixed point exists (c.f. [1]).

Proof of Theorem 28

Let $\Pi = \{\Pi_1, \ldots, \Pi_n\}$ be a program in $\mathrm{Datalog}^{\mathsf{rbr}}_{\mathsf{rdf}}$.

For each rule R of Π_i of form

$$P_i(\bar{x}_i) \leftarrow *R^i_1(\bar{z}^i_1), \ldots, *R^i_m(\bar{z}^i_m)$$

where each $*R_j(\bar{z}_j)$ is either P_ℓ, with $\ell \leq i$, or an EDB of the program, or the negation of a P_ℓ with $\ell < i$; let $T_1, \ldots, T_n$ be fresh IRIs, and let $\tau(R)$ be the result of replacing each IDB of form $P_\ell(\bar{z})$ for $Named(T_\ell, \bar{z})$ in both the body and head of P_i.

Construct a c-query Q_{Π_i} whose set of mappings corresponds to the union of query $\phi(\bar{x}_i) = *R_1^i(\bar{z}_1^i) \wedge \cdots \wedge *R_m^i(\bar{z}_m^i)$, according to Lemma 6, of all such $\tau(R)$, for each rule in Π_i.

Further, note that such query can be actually stated as query that satisfies the conditions of semi-positiveness, as it does not use predicates $Named(a, b, c, d)$ where a is a variable, and all instances of $Named(T_i, \bar{z})$ appear positive, meaning that negation shall not be needed for anything mentioning $Named(T_i, \bar{z})$.

By induction on the length of the proof it is immediate to show that the graph computed by the query

$$\begin{aligned}&\mathsf{WITH\ RECURSIVE}\ T_1\ \mathsf{AS}\{\mathsf{CONSTRUCT}(\bar{x}_1)\ \mathsf{FROM}\ Q_{P_1}\}\\ &\quad\{\mathsf{WITH\ RECURSIVE}\ T_2\ \mathsf{AS}\{\mathsf{CONSTRUCT}(\bar{x}_1)\ \mathsf{FROM}\ Q_{P_1}\}\\ &\qquad\{\cdots\{\mathsf{WITH\ RECURSIVE}\ T_n\ \mathsf{AS}\{Q_{P_n}\}\mathsf{CONSTRUCT}(\bar{x})\ \mathsf{FROM\ NAMED}\ T_n\}\cdots\}\}\end{aligned} \tag{5}$$

corresponds to the answer of the rule P_n.

For the other direction, consider a semi-positive recursive SPARQL of form 5. We are assuming that the objective is to output the constructed graph, but without loss of generality the following argument can be extended to a query of the above form where the final non recursive query is any c-query.

We then have that none of $Q_{P_1}, \ldots, Q_{P_n}$ is recursive, and that they satisfy the restrictions of semi-positiveness. From the proof of Lemma 6 we have that each of Q_{P_i} can be transformed into an FO query over $\mathrm{FO}_{\mathsf{rdf}}$ that is of the following form: no predicate $Named(a, b, c, d)$, where a is a variable or a T_i, appears under negation.

In order to convert this into a datalog program with rule-by-rule stratified negation, there are some technicalities:

First, every instance of $Named(a, b, c, d)$, where a is a variable, must be transformed into the disjunction of $Named(a, b, c, d) \wedge a \neq T_1 \wedge \cdots \wedge T_n$ and $Named(a, b, c, d) \wedge a = T_i$, for each T_i. Afterwards we can replace all instances of $Named(a, b, c, d) \wedge a = T_i$ or $Named(T_i, b, c, d)$ for a new predicate $T_i(b, c, d)$.

Now converting the resulting FO queries into Datalog (as done, for instance, in [1]), yields that their negation is stratified, because no $Named(T_i, b, c, d)$ or $Named(a, b, c, d)$, with a a variable, was under the scope of a negation in the previous query (this is guaranteed by the semi-positiveness). The union of each of the programs gives us a datalog program that has rule-by-rule stratified negation, and is equivalent to the original query.