CONSTRUCT Queries in SPARQL

Egor V. Kostylev, Juan L. Reutter, and Martín Ugarte

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 embark 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 worthwhile 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 restated 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.

1998 ACM Subject Classification H.2.3 Languages – Query languages

Keywords and phrases RDF, SPARQL, Query Languages

Digital Object Identifier 10.4230/LIPIcs.ICDT.2015.212

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 [33], SPARQL was officially made the recommended language to query RDF data by W3C in 2008 [32]. A recent version of the standard, denoted SPARQL 1.1, was issued in 2013 [39]. Nowadays this language is recognized as one of the key standards of the Semantic Web initiative and there are several SPARQL engines available to industry (e.g., [12, 18, 37]).

The theoretical foundations of SPARQL were laid by Pérez et al. in their seminal work [27], and a body of research has followed covering a variety of issues such as complexity of query evaluation [4, 24, 29, 36], query optimization [8, 9, 22, 30], federation [7], expressive power
Figure 1 (a) RDF graph $G_{ex}$; (b) answer of $q_{sel}$ over $G_{ex}$ is the set of mappings $\{\mu_1, \mu_2, \mu_3\}$; (c) answer of $q_{cons}$ over $G_{ex}$ is RDF graph.

Example 1. As a classical example of SPARQL, let us consider the following query $q_{sel}$:

```
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 $q_{sel}$ 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, 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 $q_{sel}$:

```
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 $q_{sel}$ 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., [27, 31]). For example, when studying the expressive power of SPARQL in [2, 31], 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

---

1 In this paper we follow the SPARQL syntax of [27], in particular, we shorten OPTIONAL to OPT.
that the SELECT fragment of SPARQL is equivalent in expressive power to relational algebra, this is shown using proofs that are much more complicated than other similar results in database theory, and it has been difficult to build upon these 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 [39], 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 $q_{cons}$ 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 $q_{sel}$, 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 [27].

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 [27]) 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 $\Sigma^p_2$-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 [39], c-query answering over OWL 2 RL entailment regime [16,20], navigational SPARQL [28], 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 \( I \) and \( B \) be infinite pairwise disjoint sets of IRIs and blank nodes,\(^2\) respectively, and \( T = I \cup B \) be the set of terms. Then an RDF triple is a tuple \((s, p, o)\) from \( T \times I \times T \), 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 \( \{G_0, (u_1, G_1), \ldots, (u_n, G_n)\} \), 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 \( gr_D(u) = G \) if \((u, G) \in D\) and \( gr_D(u) = \emptyset \) otherwise.

We also use \( G \) and \( D \) to denote the sets of all RDF graphs and datasets, correspondingly, as well as 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 \( V \) be an infinite set \( \{?x, ?y, \ldots\} \) of variables, disjoint from \( T \). Similarly to \( \text{var}(S) \), let \( \text{var}(S) \) denote the set of variables appearing in \( S \). SPARQL graph patterns are recursively defined as follows:

1. a triple in \((I \cup V) \times (I \cup V) \times (I \cup V)\) is a graph pattern, called a triple pattern;
2. if \( P_1 \) and \( P_2 \) are graph patterns then \((P_1 \text{ AND } P_2)\), \((P_1 \text{ OPT } P_2)\), and \((P_1 \text{ UNION } P_2)\) are graph patterns, called AND-, OPT-, and UNION-patterns, correspondingly;
3. if \( P \) is a graph pattern and \( g \in I \cup V \) then \((g \text{ GRAPH } P)\) is a graph pattern, called a GRAPH-pattern;
4. if \( P \) is a graph pattern and \( R \) is a filter condition then \((P \text{ FILTER } R)\) 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 \( ?x, ?y \in V \) and \( u \in I \) (called atomic constraints\(^3\)),
   - \( \neg R, R_1 \land R_2 \), or \( R_1 \lor R_2 \), 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 [39].
A SPARQL CONSTRUCT query, or c-query for short, is an expression

CONSTRUCT H WHERE P,

where \( H \) is a set of triples from \( (T \cup V) \times (I \cup V) \times (T \cup V) \), 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, c-SPARQL\(_{bf,gf} \) 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 \( V \) to terms \( T \). The domain \( \text{dom}(\mu) \) of a mapping \( \mu \) is the set of variables on which \( \mu \) is defined. Two mappings \( \mu_1 \) and \( \mu_2 \) are compatible (written as \( \mu_1 \sim \mu_2 \)) if \( \mu_1(\tau x) = \mu_2(\tau x) \) for all variables \( \tau x \) in \( \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \). If \( \mu_1 \sim \mu_2 \), then we write \( \mu_1 \cup \mu_2 \) for the mapping obtained by extending \( \mu_1 \) according to \( \mu_2 \) on all the variables in \( \text{dom}(\mu_2) \setminus \text{dom}(\mu_1) \).

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:

\[
\begin{align*}
M_1 \Join M_2 &= \{ \mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2 \}, \\
M_1 \cup M_2 &= \{ \mu \mid \mu \in M_1 \text{ or } \mu \in M_2 \}, \\
M_1 \setminus M_2 &= \{ \mu_1 \mid \mu_1 \in M_1 \text{ and there is no } \mu_2 \in M_2 \text{ such that } \mu_1 \sim \mu_2 \}.
\end{align*}
\]

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

\[ M_1 \bowtie M_2 = (M_1 \Join M_2) \cup (M_1 \setminus M_2). \]

Given a dataset \( D = \{ G_0, (u_1, G_1), \ldots, (u_n, G_n) \} \), and a graph \( G \) among \( G_0, \ldots, G_n \), the evaluation \( \llbracket P \rrbracket_D^G \) of a graph pattern \( P \) over \( D \) with respect to \( G \) is defined as follows:

1. if \( P \) is a triple pattern, then \( \llbracket P \rrbracket_D^G = \{ \mu : \text{var}(P) \rightarrow T \mid \mu(P) \in G \} \);
2. if \( P = (P_1 \text{ AND } P_2) \), then \( \llbracket P \rrbracket_D^G = \llbracket P_1 \rrbracket_D^G \Join \llbracket P_2 \rrbracket_D^G \);
3. if \( P = (P_1 \text{ OPT } P_2) \), then \( \llbracket P \rrbracket_D^G = \llbracket P_1 \rrbracket_D^G \bowtie \llbracket P_2 \rrbracket_D^G \);
4. if \( P = (P_1 \text{ UNION } P_2) \), then \( \llbracket P \rrbracket_D^G = \llbracket P_1 \rrbracket_D^G \cup \llbracket P_2 \rrbracket_D^G \);
5. if \( P = (g \text{ GRAPH } P') \), then

\[
\llbracket P \rrbracket_D^G = \begin{cases} 
\llbracket P' \rrbracket_{gr}(g) & \text{if } g \in I \\
\bigcup_{u \in I} \left( \llbracket P' \rrbracket_{gr(u)} \Join \{ \mu_{y \mapsto u} \} \right) & \text{if } g \in V
\end{cases}
\]

where \( \mu_{y \mapsto u} \) is the mapping with domain \( \{ g \} \) and where \( \mu_{y \mapsto u}(g) = u \);

6. if \( P = (P' \text{ FILTER } R) \), then \( \llbracket P \rrbracket_D^G = \{ \mu \mid \mu \in \llbracket P' \rrbracket_D^G \text{ and } \mu \models R \} \), where a mapping \( \mu \) satisfies a built-in condition \( R \), denoted by \( \mu \models R \), if one of the following holds:

- \( R \) is \( ?x = u \), \( ?x \in \text{dom}(\mu) \) and \( \mu(?x) = u \); or
- \( R \) is \( ?x \neq ?y \), \( ?x \in \text{dom}(\mu) \), \( ?y \in \text{dom}(\mu) \) and \( \mu(?x) = \mu(?y) \); or
- \( R \) is \( \text{isBlank}(?x) \) and \( ?x \in \text{dom}(\mu) \) and \( \mu(?x) \in B \); or
- \( R \) is \( \text{bound}(?x) \) and \( ?x \in \text{dom}(\mu) ; \) or
- \( R \) is a Boolean combination of other filter conditions and this combination is satisfied according to the usual notions of \( \{ \neg, \lor, \land \} \).
The evaluation \([P]_D^D\) of a pattern \(P\) over a dataset \(D\) with default graph \(G_0\) is \([[P]_D^D]_G_0\).

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

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

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 \(\text{GRAPH}\) and \(\text{CONSTRUCT}\). 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:

\[
\text{CONSTRUCT} \{\langle ?\text{name}, \text{lives_in}, ?\text{country} \rangle \} \text{ WHERE } (\langle ?\text{worker}, \text{name}, ?\text{name} \rangle \text{ AND } \langle ?\text{worker}, \text{works_at}, ?\text{university} \rangle \text{ AND } \\
\langle \text{country GRAPH } (?\text{university}, \text{in_country}, ?\text{country}) \rangle).
\]

### 3 Blank-free c-Queries

We start our study with \(c\text{-SPARQL}_{bf}\), 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 \(\text{SPARQL SELECT}\) queries in the sense that all the blank nodes in the answer graph of a \(c\text{-SPARQL}_{bf}\) query already appear in the input dataset.

The first problem we consider is the expressive power of \(c\text{-SPARQL}_{bf}\). 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 \(\text{Default}, \text{Named}\) and \(\text{IsBlank}\), of arities 3, 4 and 1, respectively. Then the language \(\text{FO}_{bf}\) consists of all well-formed ternary FO formulas over this signature. We always assume that the domain of FO structures is the set \(T\) of terms, and that for all structures we have that \(\text{IsBlank}(b)\) holds for some \(b\) if and only if \(b \in B\), and \(\text{IsBlank}(b)\) implies that none of \(\text{Default}(a, b, c), \text{Named}(b, a, c, d)\) and \(\text{Named}(d, a, b, c)\) hold for any \(a, c,\) and \(d\). Thus the answers for this language are sets of triples from \(T \times I \times T\), essentially RDF graphs. Finally, the evaluation function for \(\text{FO}_{bf}\) is the usual FO entailment \(\models_{adom}\) over active domain semantics. This means that quantification
is realised over the finite set of all the terms from T appearing in the input database and query (see [1] for formal definitions).

Note that the set of input databases of FO$_{ef}$ have a straightforward one-to-one correspondence with the set of input datasets of c-SPARQL$_{ef}$ 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 $Q_1$ is contained in a language $Q_2$ if and only if there are bijections $\text{trans}_I : I_1 \rightarrow I_2$, $\text{trans}_O : O_1 \rightarrow O_2$ between their input sets $I_i$ and answer sets $O_i$, and a function $\text{trans}_Q : Q_1 \rightarrow Q_2$ such that $\text{trans}_Q(\text{eval}_1(q,I)) = \text{eval}_2(\text{trans}_Q(q),\text{trans}_I(I))$ holds for any $q \in Q_1$ and $I \in I_1$, where $\text{eval}_i$ 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.

\section*{Lemma 4.} The language c-SPARQL$_{ef}$ is contained in FO$_{ef}$.

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 [31]. 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 $\texttt{Null}$. We present a simpler reduction where $\texttt{Null}$ is not used, but instead we create a predicate for each subset of the set of variables of $Q$. Avoiding predicate $\texttt{Null}$ makes our proof much more simple and intuitive, and we make use of this proof to obtain several results in the following section.

\section*{Proof (idea).} First we establish an equivalence between graph patterns and FO$_{ef}$. 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 $X \subseteq \text{var}(P)$ we construct a formula $\varphi^P_X$ with $X$ as free variables, such that a mapping $\mu$ is in $[P]^D$ for a dataset $D$ if and only if the variable assignment defined by $\mu$ satisfies $\varphi^\text{dom}(\mu)$ in the FO$_{ef}$ 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 $\text{var}(P)$, the formula

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

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

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

\section*{Example 5.} Recall the query $q_{\text{cons}}$ 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 \( \var(q_{\text{cons}}) \) will be unsatisfiable. Following the construction process, we obtain

\[
\varphi([p,?,n,?w]) (p, n, w) = \text{Default}(p, \text{name}, n) \land \text{Default}(p, \text{works\_at}, w) \land \\
\neg \exists e \text{ Default}(p, \text{mbox}, e),
\]

\[
\varphi([p,?,n,?w,?e]) (p, n, w, e) = \text{Default}(p, \text{name}, n) \land \text{Default}(p, \text{works\_at}, w) \land \\
\text{Default}(p, \text{mbox}, e),
\]

where ‘?’ is omitted before variables to resemble the conventional FO notation. Having these, we need to create the formula \( \varphi_{q_{\text{cons}}} \) that always outputs exactly the same graph as \( q_{\text{cons}} \). 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

\[
\varphi_{q_{\text{cons}}}(x, y, z) = \neg \text{IsBlank}(y) \land \\
\exists p, n, w \left[ \varphi([p,?,n,?w,?e]) (p, n, w, e) \land (x = n \land y = \text{works\_at} \land z = w) \right] \lor \\
\exists p, n, w, e \left[ \varphi([p,?,n,?w,?e]) (p, n, w, e, ?) \land (x = n \land y = \text{mbox} \land z = e) \right].
\]

Our next result is the inclusion in the other direction.

**Lemma 6.** The language \( \text{FO}_{\text{ref}} \) is contained in \( \text{c-SPARQL}_{\text{bf}} \).

**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 [27]. Let

\[
P_1 \text{ MINUS } P_2 = (P_1 \text{ OPT } (P_2 \text{ AND } (?x_1, ?x_2, ?x_3))) \text{ FILTER } \neg \text{bound}(?x_1),
\]

where \( ?x_1, ?x_2 \) and \( ?x_3 \) are mentioned neither in \( P_1 \) nor in \( P_2 \). It is readily verified that \([P_1 \text{ MINUS } P_2]_D^G = [P_1]_D^G \setminus [P_2]_D^G\) 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 \( \text{c-SPARQL}_{\text{bf},gf} \) and \( \text{FO}_{\text{ref}} \) are equivalent in expressive power.

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

**Corollary 8.** The languages \( \text{c-SPARQL}_{\text{bf},gf} \) and \( \text{FO}_{\text{ref}}^{\text{ter}} \) 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 \( Q \) with the same input and answer sets \( I \), and evaluation function \( \text{eval} \), is composable if and only if for every query \( q \) there is another query \( q' \in Q \) such that \( \text{eval}(q, \text{eval}(q', I)) = \text{eval}(q, I) \) for any input database.
I ∈ I. According to this definition, it makes no sense to talk about compositability of the language c-SPARQLbf of blank-free c-queries, because it does not satisfies the condition that the sets of inputs and answers coincide. But queries in c-SPARQLbf, gf enjoy such a property, and we can obtain compositability of c-SPARQLbf, gf from compositability of FOternary.

**Corollary 9.** The language c-SPARQLbf, 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 [27]. 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-SPARQLbf 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-SPARQLbf, 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 [27] 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 Σp2-complete if projection is allowed. Also, several optimization techniques have been developed for the evaluation of well-designed queries (see [22,27]). 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-SPARQLbf, 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 \((P \text{ FILTER } R)\) such that all variables in \(R\) are mentioned in \(P\),
3. only OPT-subpatterns \((P_1 \text{ OPT } P_2)\) 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 \(\text{uwd}\) and of to specify sublanguages satisfying these restrictions, such as c-SPARQLbf, gf. Note that c-SPARQLbf, gf contains c-SPARQLbf, 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.

---

4 The latter setting is known as data complexity of the problem (see [38]).
Lemma 11. The language \(c\text{-SPARQL}_{bf, gf}^{\text{uwd}}\) is contained in \(c\text{-SPARQL}_{bf, gf}^{\text{of}}\).

This result relies on the following fact: Consider a c-query \(\text{CONSTRUCT} \ H \ \text{WHERE} \ P\) in \(c\text{-SPARQL}_{bf, gf}^{\text{uwd}}\). 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 \text{SELECT} queries, which explains the importance of well-designed \(\text{OPT}\) in those classes of queries.

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

Corollary 12. Every c-query from \(c\text{-SPARQL}_{bf, gf}^{\text{uwd}}\) can be transformed to an equivalent c-query from \(c\text{-SPARQL}_{bf, gf}^{\text{of}}\) in \(\text{LOGSPACE}\).

We also relate the described languages to a fragment of first order logic, defined next. A formula \(\varphi \in \text{FO}_{\text{ternary}}^{\text{rdf}}\) is \(\exists\)-positive if it is in the \(\{\exists, \neg, \land, \lor\}\) 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 \(\exists\text{pos-FO}_{\text{ternary}}^{\text{rdf}}\).

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

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

Quite similarly, an inspection of the proof of Lemma 6 shows that a transformation of an existential formula in which the predicate \text{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 \(\exists\text{pos-FO}_{\text{ternary}}^{\text{rdf}}\) is contained in \(c\text{-SPARQL}_{bf, gf}^{\text{uwd}}\).

We can now state the main theorem of this section.

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

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

Corollary 16. The language \(c\text{-SPARQL}_{bf, gf}^{\text{uwd}}\) is composable.

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

Proposition 17. The problem of checking whether a triple is in the answer to a c-query from \(c\text{-SPARQL}_{bf, gf}^{\text{uwd}}\) over a dataset is \(\text{NP}\)-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 \(c\text{-SPARQL}_{bf, gf}^{\text{uwd}}\) queries, that is, c-queries that do not use \text{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 \( \mu \) from \( \text{var}(H) \) to the domain of \( D \), an injective function \( f_\mu : \text{blank}(H) \to \mathcal{B} \setminus \text{blank}(D) \). These functions must have pairwise disjoint ranges.

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

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

where \( f_\mu \) is the corresponding renaming function for \( \mu \) in \( F(H, D) \).

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

\[
\text{CONSTRUCT} \ \{ (_:\text{b}, \text{manages}, \text{?n}), (_\text{?n}, \text{mbox}, \text{?e}) \} \\
\text{WHERE} \ \\
( (\text{?p, name, ?n}) \land (\text{?p, works_at, ?w}) ) \\
\text{OPT} \ (\text{?p, mbox, ?e}),
\]

where \( _:\text{b} \) 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 \( f_{\mu_1}(_\text{b}) = _\text{b1} \), \( f_{\mu_2}(_\text{b}) = _\text{b2} \) and \( f_{\mu_3}(_\text{b}) = _\text{b3} \) then the result of evaluating the query above over \( G_{ex} \) contains the triples \( (_\text{b1,manages,Fran}), (_\text{b2,manages,Cristian}), (_\text{b3,manages,Cristian}) \) and \( (_\text{Cristian,mbox,cris@puc.cl}) \).

In order to understand the properties of c-SPARQL, 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 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 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, 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, 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

\[
\forall \bar{x} \forall \bar{y} \ (\varphi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \ \psi(\bar{x}, \bar{z})),
\]

where \( \bar{x}, \bar{y}, \) and \( \bar{z} \) are disjoint tuples of variables and \( \varphi \) and \( \psi \) are first-order formulas. We concentrate on a restricted class of dependencies, that we call source-to-target dependencies

\(^5\) Our definition differs slightly from the chase for RDF used in [10], but it follows the same spirit.
(or st-dependencies), which are those in which \( \varphi \) belongs to \( \text{FO}_{\text{ternary}} \) (i.e., formulas over Default and IsBlank relations), and \( \psi \) is a conjunction of atoms over another ternary relation \( \text{OTriple} \). 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 \( \text{DE}_{\text{rdf}} \) the language of de-mappings.

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

In data exchange one is usually interested in computing universal solutions for a source instance and a de-mapping \( \Sigma \). 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 \( \Sigma \) as a target instance constructed by sequentially adding triples to \( \text{OTriple} \). To compute this instance, for every st-dependency of the form (1) in \( \Sigma \) proceed as follows. Take every assignment \( \pi : \vec{x} \cup \vec{y} \rightarrow T \) such that \( S \models_{\text{adom}} \varphi(\pi(\vec{x}), \pi(\vec{y})) \) and extend \( \pi \) by assigning a distinct fresh blank node from \( B \) to each variable in \( \vec{z} \). Then add to the target instance the fact \( \text{OTriple}(\pi(v_1), \pi(v_2), \pi(v_3)) \) for each conjunct \( \text{OTriple}(v_1, v_2, v_3) \) in \( \psi \), 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 \( \text{DE}_{\text{rdf}} \) 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.

\[ \text{Theorem 19.} \quad \text{The languages c-SPARQL}_{\text{gf}} \text{ and } \text{DE}_{\text{rdf}} \text{ are equivalent in expressive power.} \]

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.

\[ \text{Proposition 20.} \quad \text{The language c-SPARQL}_{\text{gf}} \text{ is not composable.} \]

Next we refine the results above for the language c-SPARQL_{\text{gf}}^{\text{uwd}}. Since we have shown that such queries are equivalent to positive FO, it would be reasonable to guess that c-SPARQL_{\text{gf}}^{\text{uwd}} is equivalent to the query language given by de-mappings where every dependency (1) is such that the formula \( \varphi \) 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 GLAV_{\text{rdf}} in this paper.

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

\[ \text{Example 21.} \quad \text{Consider the c-query} \]

\[
\text{CONSTRUCT \{_:b, p, ?x\}, (_:b, p, ?y)\}}
\]
\[
\text{WHERE ((?x, p, a) OPT (?x, p, ?y))}.
\]

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
\[
\forall x \forall y (\text{Default}(x, p, a) \land \text{Default}(x, p, y) \rightarrow \exists z (\text{OTriple}(z, p, x) \land \text{OTriple}(z, p, y))),
\]
\[
\forall x (\text{Default}(x, p, a) \rightarrow \exists z \text{OTriple}(z, p, x)),
\]
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 GLAV_{\text{rdf}}.

In the above example both the chase of the de-mapping and the answer to the \textit{CONSTRUCT} query are \textit{homomorphically equivalent}, in the sense of [19]. This correspondence is not accidental, and an equivalence between GLAV_{\text{rdf}} and c-SPARQL^{\text{uwd}} 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.

\begin{itemize}
\item \textbf{Proposition 22.} The language GLAV_{\text{rdf}} is contained in c-SPARQL^{\text{uwd}}.
\end{itemize}

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.

\begin{itemize}
\item \textbf{Corollary 23.} The language c-SPARQL^{\text{uwd}} is not composable.
\end{itemize}

We conclude this section with the following observation. Example 21 is problematic as we use the same blank nodes in two triples in the \textit{CONSTRUCT} template. If we disallow blank nodes we regain equivalence between c-queries and data exchange settings, since c-SPARQL^{\text{uwd}} 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 [34]. The recent version 1.1 of SPARQL also allows for some form of recursion, namely property paths [39], 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. [23,28], 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 \textit{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.
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 \((A_3, \text{same:user}, A_1)\), among others. In SPARQL we propose to obtain all such triples by means of the following query:

```sparql
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\textsubscript{bf} or an expression of the form

\[
\text{WITH RECURSIVE } t \text{ AS } \{ q_1 \} q_2,
\]

where \( t \) is an IRI from \( I \), \( q_1 \) is a c-query from c-SPARQL\textsubscript{bf}, and \( q_2 \) is a recursive c-query. The set of all recursive c-queries is denoted rec-c-SPARQL\textsubscript{bf}.

We reinforce the idea that in this definition \( q_1 \) is non-recursive, but \( q_2 \) 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 \( \text{gr}_{D \cup D'}(u) = \text{gr}_D(u) \cup \text{gr}_{D'}(u) \) for any URI \( u \).

Definition 26. If a recursive query \( q \) from rec-c-SPARQL\textsubscript{bf} is a simple c-query, then its answer \( \text{ans}(q, D) \) over a dataset \( D \) is defined according to the semantics of c-queries. Otherwise, that is, if \( q \) is of the form (2), then the answer \( \text{ans}(q, D) \) is equal to \( \text{ans}(q_2, D_{\Pi_0}) \), where \( D_{\Pi_0} \) is the least fixed point of the sequence \( D_0, D_1, \ldots \) with \( D_0 = D \) and

\[
D_{i+1} = D \cup \{(t, \text{ans}(q_1, D_i))\}, \quad \text{for } i \geq 0.
\]

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 \( \text{ans}(q, D) \) is well-defined. By our results on expressive power, one way to guarantee this is to require graph pattern of the c-query \( q_1 \) 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\textsubscript{bf}. A recursive c-query \( q \) is semi-positive if it is either a simple c-query, or it is of the form (2) with \( q_2 \) semi-positive and every subpattern \( P \) in \( q_1 \) satisfying the following conditions:

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

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

Proposition 27. For every recursive c-query \( q \) in rec-c-SPARQL\textsubscript{bf}\textsubscript{semi} and dataset \( D \) the answer \( \text{ans}(q, D) \) 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

\[
Pr(\overline{x}) \leftarrow \varphi(\overline{x}, \overline{y}),
\]

where \( \overline{x} \) and \( \overline{y} \) are tuples of variables, \( Pr \) is a predicate from \( V \cup \{\text{OTriple}\} \), and \( \varphi \) is a conjunction of positive and negated atoms (including equalities) over \( I \cup V \cup \{\text{Default, Named}\} \), such that every variable from \( \overline{x} \) appears in \( \varphi \). In such a rule, \( Pr(\overline{x}) \) is the head and \( \varphi(\overline{x}, \overline{y}) \) 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 $\Pi_i$ is the predicate $Pr_i$;
2. each $\Pi_i$ does not mention any $Pr_j$ with $j > i$;
3. each $\Pi_i$ 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 $Datalog_{rbr}$. 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 $rec$-c-SPARQL_{semi} and $Datalog_{rbr}$ 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 $rec$-c-SPARQL_{semi} contains first order logic with transitive closure. Second, it is a technicality to check that this query formalism contains SPARQL 1.1 property paths [39], c-query answering over OWL 2 RL entailment regime [16], the algebra defined in [35], navigational SPARQL [28], as well as GraphLog [11] and TriAL [23] query languages. Also, it is possible to show that none of these formalisms can express all $rec$-c-SPARQL_{semi} queries, that is, the containment is strict in all the cases. Hence, we may conclude that $rec$-c-SPARQL_{semi} 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

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