Nested Constructs vs. Sub-Selects in SPARQL[⋆]

Axel Polleres[1], Juan Reutter[2], and Egor V. Kostylev[3]

[1] Vienna University of Economics and Business, AT
[2] Pontificia Universidad Católica de Chile and Center for Semantic Web research, CL
[3] University of Oxford, UK

Abstract. The issue of subqueries in SPARQL has appeared in different papers as an extension point to the original SPARQL query language. Particularly, nested CONSTRUCT in FROM clauses are a feature that has been discussed as a potential input for SPARQL 1.1 which was resolved to be left out in favour of select subqueries under the—unproven—conjecture that such subqueries can express nested construct queries. In this paper, we show that it is indeed possible to unfold nested SPARQL construct queries into subqueries in SPARQL 1.1; our transformation, however, requires an exponential blowup in the nesting depth. This suggests that nested construct queries are indeed a useful syntactic feature in SPARQL that cannot compactly be replaced by subqueries.

1 Introduction

SPARQL as a standard query language for the Semantic Web data has been the subject of many theoretical and practical works in over the last years. Upon version 1.0 of its standardisation, works have been published on its expressivity [2, 8, 10, 11] and on adding features, among others value creation and subqueries [2, 9].

These features have been taken on board in version 1.1 of the SPARQL specification. The first feature corresponds to an extension of SPARQL select queries, allowing SELECT clauses to assign values (i.e., blank nodes, literals or IRIs) to particular variables, by means of a new keyword AS. But more importantly, SPARQL 1.1 now permits the use of *subqueries*, that is, select queries can be arbitrarily nested within WHERE clauses [4, Section 12].

To showcase these operators, let us start from the following simple SPARQL 1.0 query that essentially replaces all object values in a given graph with blank nodes:

```
CONSTRUCT { ?S ?P []} WHERE { ?S ?P ?O }.
```

In SPARQL 1.1 we could assign blank nodes explicitly by means of a subquery:

```
CONSTRUCT { ?S ?P ?Y } WHERE
 { SELECT ?S ?P ( bnode() AS ?Y ) WHERE { ?S ?P ?O } },
```

which can be written more compactly[4] as follows

[⋆] This work was supported by Jubiläumsstiftung WU Wien as well as by the WWTF project "SEE", and also by Iniciativa Científica Milenio Grant 120004

[4] The BIND keyword is just syntactic sugar for writing a sub-SELECT query which just assigns a single expression to a variable more concisely.

```
CONSTRUCT { ?S ?P ?Y } WHERE { ?S ?P ?O BIND( bnode() AS ?Y ) }.
```

While nested select queries are allowed in SPARQL 1.1, other forms of subqueries such as nested CONSTRUCT queries in FROM clauses, as proposed in [2, 9], were not included to the new specification under the—unproven—conjecture that select subqueries can express nested construct queries.[5]

For instance, consider the following query Q that uses nested constructs:

```
CONSTRUCT { ?X ex:goodFriend ?Y }
FROM {
     CONSTRUCT { ?U ex:friends ?V }
     WHERE { ?U foaf:knows ?V . ?U ex:worksWith ?V }
     }
WHERE { ?X ex:friends ?Y . ?X ex:friends ?Z . ?Y ex:friends ?Z }.
```

The intention of this query is to construct first an RDF document where a person U is a friend of V if U knows V and U works together with V. This is done by means of the *inner* construct subquery of Q. The resulting graph is then queried by the WHERE clause of the *outer* query, which creates a graph of good friends: X is a good friend of Y if X and Y are friends and have another friend in common. With a little work one can see that query Q is indeed equivalent to the following SPARQL 1.1 query:

```
  CONSTRUCT { ?X ex:goodFriend ?Y } WHERE {
      { SELECT ?X ex:friends AS ?W1 ?Y
          WHERE { ?X foaf:knows ?Y . ?X ex:worksWith ?Y } } .
      { SELECT ?X ex:friends AS ?W2 ?Z
          WHERE { ?X foaf:knows ?Z . ?X ex:worksWith ?Z } } .
      { SELECT ?Y ex:friends AS ?W3 ?Z
          WHERE { ?Y foaf:knows ?Z . ?Y ex:worksWith ?Z } } }.
```

In essence, we have replaced and rewritten each of the triples in the outer WHERE clause for the conditions in the inner construct query that give rise to these triples, as it is for instance commonly done in query rewriting using views [1].

But can this idea be generalised for all nested construct queries? That is, is it true that given any SPARQL query Q that uses nested CONSTRUCT one can find a query in SPARQL 1.1 (without nested CONSTRUCT) that is equivalent to Q? This question was considered in [7], with a positive answer for a restricted case of SPARQL queries where one disallows blank nodes in templates and only uses the functionalities in SPARQL 1.0. On the other hand, it was shown that there are some queries with nested CONSTRUCT that cannot be expressed as a SPARQL 1.0 query (this follows from the close connection between construct queries and data exchange [3]). However, the results in [7] consider a much weaker language, because they do not consider any of the SPARQL 1.1 operators, and in particular the binding of variables in SELECT subqueries.

In this paper we show that by allowing full SPARQL 1.1 one can indeed express nested construct queries, that is, in other words, the language of SPARQL 1.1 construct

[5] https://www.w3.org/2009/sparql/track/issues/7

queries is *composable*. However, our result, just as [7], assumes set semantics, contrary to bag semantics in the specification. Also, our rewriting, even if polynomial when just one construct query is nested in another, is exponential in the depth of nesting. Moreover, it is quite cumbersome and unintuitive. Nevertheless, this is the first step in proving the conjecture that nested SELECT queries are indeed the right notion for subqueries in SPARQL. We do believe that looking at set semantics is a natural and important first step in the direction of this work.

The plan of this paper is as follows: after formalisation of SPARQL 1.1 in Section 2, we define the notion of composability in Section 3, present our rewriting in Section 4, and conclude in Section 5.

2 SPARQL Algebra

We next recapitulate the SPARQL algebra as well as basic notions on RDF.

RDF Graphs Let $\mathbf{I}$, $\mathbf{L}$, and $\mathbf{B}$ be countably infinite pairwise disjoint sets of *IRIs*, *literals*, and *blank nodes*, respectively, where literals include numbers, strings, and Boolean values `true` and `false`. The set of *(RDF) terms* $\mathbf{T}$ is $\mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$. An *(RDF) triple* is an element (s, p, o) of $(\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times \mathbf{T}$ where s is called the *subject*, p the *predicate*, and o the *object*. An *(RDF) graph* is a finite set of RDF triples.

SPARQL Algebra Syntax We concentrate on the core of SPARQL 1.1 and build upon the formalisation by [8]. We distinguish three types of syntactic building blocks—filter expressions, patterns, and queries, built over terms $\mathbf{T}$ and an infinite set $\mathbf{V} = \{?x, ?y, \dots\}$ of *variables*, disjoint from $\mathbf{T}$ as defined next.

(Filter) expressions are defined inductively as follows:
- all variables in $\mathbf{V}$ and all terms in $\mathbf{I} \cup \mathbf{L}$ are expressions,
- $\mathsf{bNode}()$ and $\mathsf{IRINode}(b)$, for $b \in \mathbf{B}$, are expressions,
- if $?x$ is a variable in $\mathbf{V}$ then $\mathsf{bound}(?x)$ is an expression,
- if R_1 is an expression then so are $\mathsf{isBlank}(R_1)$, $\mathsf{isIRI}(R_1)$ and $\mathsf{isLiteral}(R_1)$,
- if also R_2 is an expression then so are $(R_1 = R_2)$, $(R_1 < R_2)$, $(\neg R_1)$, $(R_1 \wedge R_2)$,
- if also R_3 is an expression, then so is $\mathsf{if}(R_1, R_2, R_3)$.

We use expressions $R_1 \vee R_2$, $R_1 \rightarrow R_2$ and $R_1 \leftrightarrow R_2$ as abbreviations for $\neg(\neg R_1 \wedge \neg R_2)$, $\neg R_1 \vee R_2$ and $(R_1 \wedge R_2) \vee (\neg R_1 \wedge \neg R_2)$, respectively.

A *triple pattern* is a triple in $(\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$. Then, *(graph) patterns* are inductively defined as follows:
- a *BGP*, that is a possibly empty set of triple patterns, is a pattern;
- if P_1 and P_2 are patterns, then so are P_1 AND P_2 and P_1 UNION P_2;
- if also R is an expression, then P_1 FILTER R and P_1 OPT_R P_2 are patterns;
- if also $?x$ is a variable not appearing in P_1, then P_1 BIND R AS $?x$ is a pattern;
- if also X is a set of variables, then SELECT X WHERE R is a pattern.

If a pattern is of the form SELECT X WHERE R then the set of its *free* variables is X; otherwise, it is the union of the free variables of all its immediate subpatterns and, in case of P_1 BIND R AS $?x$, of the singleton set $\{?x\}$.

Finally, there are several types of queries in SPARQL. Most commonly used and studied are *select queries*, which are just patterns in our formalisation. In this paper, however, we concentrate on *construct queries* of the form

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

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 *template*, and P is a pattern. For S a pattern, condition, template, etc., let $\mathit{IRI}(S)$, $blank(S)$ and $var(S)$ denote all the IRIs, blank nodes and variables, respectively, that appear in S.

The standard includes several other constructs besides the ones defined above. Some of them, such as subqueries in expressions (i.e., Exists in filters), VALUES and MINUS are easily (and polynomially) expressible, under set semantics, adopted here (see below), via the core operators [5, 6]. Others, such as the named graph operator GRAPH and property paths, are immaterial to our research and omitted for brevity. Finally, note that filter expression $\mathsf{IRINode}(b)$, whose intention is to create a fresh IRI for each blank node b, does not appear in SPARQL 1.1. However, this operator can be expressed in our formalisation, provided it additionally include the string concatenation operator as well as casting operators from strings to IRIs and back, which are present in the specification.

To distinguish SPARQL 1.1 as in the standard and our formalisation we denote $\mathcal{S}_C$ the language of construct queries as in the latter.

SPARQL Algebra Semantics The semantics of patterns in $\mathcal{S}_C$ is defined in terms of *(solution) mappings*, that is, partial functions μ from variables $\mathbf{V}$ to terms $\mathbf{T}$. The domain of μ, denoted $dom(\mu)$, is the set of variables over which μ is defined. Mappings μ_1 and μ_2 are *compatible*, written $\mu_1 \sim \mu_2$, if $\mu_1(?x) = \mu_2(?x)$ for each $?x$ in $dom(\mu_1) \cap dom(\mu_2)$. If $\mu_1 \sim \mu_2$, then $\mu_1 \cup \mu_2$ is the mapping obtained by extending μ_1 according to μ_2 on all the variables in $dom(\mu_2) \setminus dom(\mu_1)$.

The *evaluation* $[[R]]_\mu$ of an expression R in $\mathcal{S}_C$ with respect to a mapping μ is a *value* in $\mathbf{T} \cup \{\mathtt{error}\}$, where `error` is a special symbol not in $\mathbf{T}$, defined as follows:

– $[[?x]]_\mu$ is $\mu(?x)$ if $?x \in dom(\mu)$ and `error` otherwise, while $[[\ell]]_\mu$ is ℓ for $\ell \in \mathbf{I}\cup\mathbf{L}$;
– $[[\mathsf{bNode}()]]_\mu$ is a fresh blank node that does not appear in the queried graph G, while $[[\mathsf{IRINode}(b)]]_\mu = g(b)$, where g is an injective function from $\mathbf{B}$ to $\mathbf{I}\setminus\mathit{IRI}(G)$;
– $[[\mathsf{bound}(?x)]]_\mu$ is `true` if $?x \in dom(\mu)$ and `false` otherwise;
– $[[\mathsf{isBlank}(R_1)]]_\mu$, $[[\mathsf{isIRI}(R_1)]]_\mu$, and $[[\mathsf{isLiteral}(R_1)]]_\mu$ are `true` if $[[R_1]]_\mu \in \mathbf{B}$, $[[R_1]]_\mu \in \mathbf{I}$, and $[[R_1]]_\mu \in \mathbf{L}$, respectively, and `false` otherwise;
– $[[R_1 \circ R_2]]_\mu$, for a comparison operator $\circ$, is $[[R_1]]_\mu \circ [[R_2]]_\mu$ if $[[R_1]]_\mu$ and $[[R_2]]_\mu$ are both not `error` and of suitable types, or `error` otherwise;
– $[[\neg R_1]]_\mu$ is `true` if $[[R_1]]_\mu = \mathtt{false}$, it is `false` if $[[R_1]]_\mu = \mathtt{true}$, and it is `error` otherwise, while $[[R_1 \wedge R_2]]_\mu$ is `true` if $[[R_1]]_\mu = [[R_2]]_\mu = \mathtt{true}$, it is `false` if $[[R_1]]_\mu$ or $[[R_2]]_\mu$ is `false`, and it is `error` otherwise;
– $[[\mathsf{if}(R_1, R_2, R_3)]]_\mu = [[R_2]]_\mu$ if $[[R_1]]_\mu = \mathtt{true}$, it is $[[R_3]]_\mu$ if $[[R_1]]_\mu = \mathtt{false}$, and `error` otherwise.

The semantics of patterns over a graph G is defined as follows, where $\mu(P)$ is the pattern obtained from P by replacing its variables according to μ:

– $[[P]]_G = \{\mu\colon var(P) \to \mathbf{T} \mid \mu(P) \subseteq G\}$ for a BGP P;
– $[[P_1\ \mathsf{AND}\ P_2]]_G = \{\mu \mid \mu_1 \in [[P_1]]_G, \mu_2 \in [[P_2]]_G, \mu = \mu_1 \cup \mu_2\}$;
– $[[P_1\ \mathsf{UNION}\ P_2]]_G = [[P_1]]_G \cup [[P_2]]_G$;
– $[[P_1\ \mathsf{FILTER}\ R]]_G = \{\mu \mid \mu \in [[P_1]]_G, [[R]]_\mu = \mathtt{true}\}$;

- $[[P_1 \, \mathsf{OPT}_R \, P_2]]_G = [[P_1 \, \mathsf{AND} \, P_2]]_G \cup \big\{\mu \mid \mu \in [[P_1]]_G, \forall \mu_2 \in [[P_2]]_G . \big(\mu \not\sim \mu_2 \text{ or } [[R]]_{\mu \cup \mu_2} = \mathtt{false}\big)\big\}$;
- $[[P \, \mathsf{BIND} \, R \, \mathsf{AS} \, ?x]]_G = \big\{\mu' \mid \mu \in [[P]]_G, \mu' = \mu \cup \{?x \mapsto [[R]]_\mu\}, [[R]]_\mu \neq \mathtt{error}\big\} \cup \big\{\mu \mid \mu \in [[P]]_G, [[R]]_\mu = \mathtt{error}\big\}$;
- $[[\mathsf{SELECT} \, X \, \mathsf{FROM} \, P_1]]_G = \big\{\mu \mid \mu = \mu'|_X, \mu' \in [[P_1]]_G\big\}$, where $\mu'|_X$ is a restriction of μ' to X.

To define the semantics of construct queries in $\mathcal{S}_C$ fix, for every template H and graph G, a family $F(H, G)$ of *renaming functions*. This family contains, for every mapping μ from $var(H)$ to $\mathbf{T}$, an injective function $f_\mu : blank(H) \to \mathbf{B} \setminus blank(G)$. These functions must have pairwise disjoint ranges (i.e., there are no b and b' such that $f_{\mu_1}(b) = f_{\mu_2}(b')$ for different μ_1 and μ_2). Then,

$$[[\mathsf{CONSTRUCT} \, H \, \mathsf{WHERE} \, P]]_G = \\ \{\mu(f_\mu(t)) \mid \mu \in [[G]]_P, t \text{ is a triple in } H \text{ and } \mu(f_\mu(t)) \text{ is well-formed}\},$$

where f_μ is the corresponding renaming function for μ in $F(H, G)$. Here, a triple is *well-formed* if it is indeed an RDF triple, that is, does not have a blank node as predicate, literal as subject, etc.[6]

Note that we adopt set semantics, contrary to bag (multi-set) semantics in the specification. We leave the consideration of bag semantics for future work.

3 Definitions and Problem Statement

In this paper we address the question of composability of SPARQL.

Definition 1. *A query language $\mathcal{L}$ with the same input and output domains $\mathcal{D}$ is* composable *if for any queries $q_1, q_2 \in \mathcal{L}$ there is a query $q \in \mathcal{L}$ such that $q_2(q_1(D)) = q(D)$ for any $D \in \mathcal{D}$, where $q'(D')$ is the output of a query q' on an input D'.*

The language of SPARQL select queries does not satisfy the requirements in this definition, because its input domain is RDF graphs and its output domain is sets of mappings. However, the language of SPARQL construct queries does satisfy the requirements, because its inputs, the set of RDF graphs, are its outputs as well; therefore, this is the subject of inquiry of this paper.

Note, however, that whether $[[Q_2]]_{[[Q_1]]_G} = [[Q]]_G$ holds, for construct queries Q_1, Q_2 and Q, depends on the particular blank-node generating functions in the definitions of the semantics of $\mathsf{bNode}()$ and construct templates. Since the intuitive meaning of blank nodes is to represent existentially quantified named nulls whose exact names are immaterial, we silently consider RDF graphs and sets of mappings up to isomorphism, that is, up to bijective renaming of blank nodes. In this way checking whether $[[Q_2]]_{[[Q_1]]_G}$ is equivalent to $[[Q]]_G$ under such renaming does not depend on the particular choice of the generating functions.

As it is mentioned in the introduction, the question of composability of construct queries was considered in [7], and it was shown, using techniques from data exchange [3], that these queries are not composable. However, the language considered

[6] Note that this can be achieved by extending P with the respective FILTER expressions.

in this previous work is different from ours, because it includes neither the BIND operator nor the $\mathsf{bNode}()$ and $\mathsf{IRINode}(b)$ functions. The main result of this paper is the following theorem, which shows that the negative result does not extend to $\mathcal{S}_C$.

Theorem 1. *The language $\mathcal{S}_C$ of construct queries is composable.*

4 Composability of Construct Queries

Consider the following queries in language $\mathcal{S}_C$ described in Section 2.

$$Q_1 = \mathsf{CONSTRUCT}\ H_1\ \mathsf{WHERE}\ P_1,$$
$$Q_2 = \mathsf{CONSTRUCT}\ H_2\ \mathsf{WHERE}\ P_2.$$

In the rest of the paper we explain how to rewrite these queries to just one query

$$Q = \mathsf{CONSTRUCT}\ H_2\ \mathsf{WHERE}\ P,$$

also in $\mathcal{S}_C$, such that $[[Q_2]]_{[[Q_1]]_G} = [[Q]]_G$ for any graph G. Since we apply Q_2 to the result of Q_1, we call these queries the *outer* and the *inner*, respectively.

Note that the template of the rewriting Q is the template of the outer query Q_2. The pattern P of Q is defined as

$$P_{\mathit{unif}}\,(P_{\mathit{blank}}, P_{\mathit{rew}})$$

for patterns P_{unif}, P_{blank}, and P_{rew}, with the former taking the latter two as parameters (i.e., subpatterns). Next we define these patterns and study their properties. It is important to mention that rewriting Q is always of polynomial size in the size of Q_1 and Q_2 (but the rewriting is exponential in the number of queries we nest in this fashion).

Without loss of generality we assume that all local (not free) variables in different SELECT subpatterns of P_1 and P_2 have different names; and that $var(P_1) \cap var(P_2) = \emptyset$ (we can always achieve this by renaming). In what follows we denote the variables in $var(P_1)$ and in $var(P_2)$ by $?x$ and $?y$, respectively (both possibly with subscripts). We also assume that P_2 does not use any BIND sub-patterns; it is possible to cover the general case by a construction similar to the one described below, but the notation becomes more elaborated.

We start with P_{blank}. The idea of this pattern is to produce the same set of mappings as P_1 except that each of them is extended to new variables bound to fresh blank nodes, one variable for each blank node in H_1. To this end, let $?b_1, \ldots, ?b_l$ be fresh variables, for l the size of $blank(H_1)$. Then

$$P_{\mathit{blank}} = P_1\ \mathsf{BIND}\ \mathsf{bNode}()\ \mathsf{AS}\ ?b_1\ \ \ldots\ \ \mathsf{BIND}\ \mathsf{bNode}()\ \mathsf{AS}\ ?b_l.$$

The following property of P_{blank} follows from the definition of the BIND operator (recall that we consider sets of mappings up to renaming of blank nodes).

Lemma 1. *For each mapping μ, let λ_μ map each $?b_i$, $1 \leq i \leq l$, to a fresh blank node. Then, for any graph G, we have that*

$$[[P_{\mathit{blank}}]]_G = \{\mu_1 \mid \mu_1 = \mu \cup \lambda_\mu, \mu \in [[P_1]]_G\}.$$

Having P_{blank} at hand we define pattern P_{rew}. Its intention is to be a rewriting of P_2 such that it works not over the result of Q_1, but over the original input graph. For any input graph, its answer, that is, its output set of mappings, coincides with the answer of P_2, except that, first, the mappings are extended to some additional variables (e.g., all projections are discarded), and, second, instead of blank nodes constructed by Q_1 the answer to P_{rew} has the corresponding IRIs $\mathsf{IRINode}(b)$ in the ranges of its mappings. Note, however, that blank nodes b here are from template H_1 and not the blank nodes $f_\mu(b)$ in the result of Q_1, so it could be several $f_\mu(b)$ for each $\mathsf{IRINode}(b)$ (more precisely, one for each μ in the evaluation of P). Replacing $\mathsf{IRINode}(b)$ with $f_\mu(b)$ is done by the third part P_{unif} of P, which takes into account both patterns P_{blank} and P_{rew}.

Consider any occurrence p of a triple pattern (s^1, s^2, s^3) in pattern P_2. Note that we consider each occurrence, not just each triple pattern: for example the two occurrences in $(?x, ?y, ?z)$ AND $(?x, ?y, ?z)$ are considered separately. Let ρ_p be a renaming function that maps each free variable $?x$ of P_1 to a fresh variable $?x_p$ and let π_p be a renaming function that maps each $?y \in var(p)$ to a fresh variable $?y_p$. Assuming that ρ_p extends to IRIs and literals as identity, let ρ_p^{blank} further extend ρ_p to each blank node b as $\mathsf{IRINode}(b)$. For each triple $t = (r^1, r^2, r^3)$ in template H_1 let $P_{\mathit{sub}}(t, p) = P^3$, with patterns P^i, $i = 1, 2, 3$, defined as follows, taking P^0 as $\rho_p(P_1)$:

– if s^i is a variable in $\mathbf{V}$ that is different from all $s^1, \dots, s^{i-1}$ then

$$P^i = P^{i-1} \ \mathsf{BIND}\ \rho_p^{\mathit{blank}}(r^i)\ \mathsf{AS}\ \pi_p(s^i);$$

– if s^i is a variable that is equal to some s^j for $1 \leq j < i$ then

$$P^i = P^{i-1}\ \mathsf{FILTER}\ \rho_p^{\mathit{blank}}(r^i) = \pi_p(s^i);$$

– finally, if $s^i \notin \mathbf{V}$ then

$$P^i = P^{i-1}\ \mathsf{FILTER}\ \rho_p^{\mathit{blank}}(r^i) = s^i.$$

For example, if $p = (?y^1, ?y^2, ?y^3)$ and $t = (?x^1, ?x^2, b)$ then $P_{\mathit{sub}}(t, p)$ is

$$\rho_p(P_1)\ \ \mathsf{BIND}\ ?x_p^1\ \mathsf{AS}\ ?y_p^1\ \ \mathsf{BIND}\ ?x_p^2\ \mathsf{AS}\ ?y_p^2\ \ \mathsf{BIND}\ \mathsf{IRINode}(b)\ \mathsf{AS}\ ?y_p^3,$$

while if $p = (?y^1, \ell, ?y^1)$ and $t = (?x^1, ?x^2, ?x^3)$ then $P_{\mathit{sub}}(t, p)$ is

$$\rho_p(P_1)\ \ \mathsf{BIND}\ ?x_p^1\ \mathsf{AS}\ ?y_p^1\ \ \mathsf{FILTER}\ ?x_p^2 = \ell\ \ \mathsf{FILTER}\ ?x_p^3 = ?y_p^1.$$

Let $p = (s^1, s^2, s^3)$ be an occurrence of triple pattern in P_2 as above, and $H_1 = t^1, \dots, t^m$. Then let P_p be the pattern

$$(P_{\mathit{sub}}(t^1, p)\ \mathsf{UNION} \dots \mathsf{UNION}\ P_{\mathit{sub}}(t^m, p))\ \mathsf{FILTER}\ \mathsf{bound}(?y^1) \wedge \dots \wedge \mathsf{bound}(?y^k),$$

where $?y^1, \dots, ?y^k$ are all the variables among s^1, s^2, and s^3.

Patterns P_p are rewritings of all particular occurrences p of triple patterns in pattern P_2. The rewritings of different occurences, however, have no variables in common by construction. Therefore, we introduce a condition $R_{\mathit{join}}(p^1, p^2, ?y)$ for any two occurrences of triple patterns p^1 and p^2 in P_2 that share a variable $?y$, that is defined as follows, for all the blank nodes $b_1, \ldots, b_l$ in $blank(H_1)$:

$$\begin{aligned}
&\neg\mathsf{bound}(?y_{p^1}) \vee \neg\mathsf{bound}(?y_{p^2}) \vee \\
&(?y_{p^1} \neq \mathsf{IRINode}(b_1) \wedge \ldots \wedge ?y_{p^1} \neq \mathsf{IRINode}(b_l) \wedge ?y_{p^1} = ?y_{p^2}) \vee \\
&((?y_{p^1} = \mathsf{IRINode}(b_1) \vee \ldots \vee ?y_{p^1} = \mathsf{IRINode}(b_l)) \wedge ?y_{p^1} = ?y_{p^2} \wedge \\
&\qquad\qquad\qquad\qquad ?x^1_{p^1} \sim ?x^1_{p^2} \wedge \cdots \wedge ?x^n_{p^1} \sim ?x^n_{p^2}),
\end{aligned}$$

where $?x^1, \ldots, ?x^n$ are all the free variables of P_1 and $?x_1 \sim ?x_2$ is the condition

$$(\mathsf{bound}(?x_1) \leftrightarrow \mathsf{bound}(?x_2)) \wedge (\mathsf{bound}(?x_1) \rightarrow ?x_1 = ?x_2).$$

The idea of $R_{\mathit{join}}(p^1, p^2, ?y)$ is as follows: if both of $?y_{p^1}$ and $?y_{p^2}$ are defined and mapped to usual IRIs, literals or blank nodes, then they should be the same; if they are defined and mapped to some $\mathsf{IRINode}(b_i)$, then they should be not only the same, but also be associated to exactly the same mapping in the answer to the inner pattern.

We now define the rewriting P_{rew} of P_2 by structural induction, and start with conditions. To this end, the rewriting $\mathit{Rew}(R')$ of a condition R' in P_2 is obtained from R' by the following two steps, for $b_1, \ldots, b_l$ all the blank nodes of H_1 and $p^1, \ldots, p^k$ all the occurrences of triple patterns of P_2 that mention $?y$:

1. replace each $\mathsf{isBlank}(?y)$ by the expression

$$\mathsf{isBlank}(?y) \vee ?y = \mathsf{IRINode}(b_1) \vee \cdots \vee ?y = \mathsf{IRINode}(b_l);$$

2. replace each variable $?y$ by the expression

$$\mathsf{if}(\mathsf{bound}(?y_{p^1}), ?y_{p^1}, \mathsf{if}(\mathsf{bound}(?y_{p^2}), ?y_{p^2}, \ldots, \mathsf{if}(\mathsf{bound}(?y_{p^{k-1}}), ?y_{p^{k-1}}, ?y_{p^k}) \ldots)).$$

Finally, the rewriting $\mathit{Rew}(P')$ for any subpattern P' of P_2 (including occurrences of triple patterns) is defined as follows:

– if P' is the empty BGP $P_\emptyset$ then $\mathit{Rew}(P') = P_\emptyset$,
– if P' is a singleton BGP $\{p\}$ then $\mathit{Rew}(P') = P_p$,
– if P' is a BGP $\{p_1, \ldots, p_k\}$ for $k > 1$ then $\mathit{Rew}(P') = \mathit{Rew}(\{p_1\}\ \mathsf{AND} \cdots \mathsf{AND}\ \{p_k\})$ (see below),
– if $P' = P'_1\, \mathsf{AND}\, P'_2$ then $\mathit{Rew}(P') = (\mathit{Rew}(P'_1)\, \mathsf{AND}\, \mathit{Rew}(P'_2))\ \mathsf{FILTER}\ R$, where R is a conjunction of $R_{\mathit{join}}(p^1, p^2, ?y)$ for each triple pattern p^1 in P'_1 and each triple pattern p^2 in P'_2 such that p^1 and p^2 have a common variable, as well as for each their common variable $?y$,
– if $P' = P'_1\ \mathsf{OPT}_{R'}\ P'_2$ then $\mathit{Rew}(P') = \mathit{Rew}(P'_1)\ \mathsf{OPT}_{\mathit{Rew}(R') \wedge R}\ \mathit{Rew}(P'_2)$, where R is as in the case of AND,
– if $P' = P'_1\ \mathsf{UNION}\ P'_2$ then $\mathit{Rew}(P') = \mathit{Rew}(P'_1)\ \mathsf{UNION}\ \mathit{Rew}(P'_2)$,
– if $P' = P'_1\ \mathsf{FILTER}\ R'$ then $\mathit{Rew}(P') = \mathit{Rew}(P'_1)\ \mathsf{FILTER}\ \mathit{Rew}(R')$;
– if $P' = \mathsf{SELECT}\ X\ \mathsf{WHERE}\ P'_1$ then $\mathit{Rew}(P') = \mathit{Rew}(P'_1)$.

The first important property of P_{rew} is given in the following lemma. Let P^*_2 be obtained from P_2 by replacing each subpattern $\mathsf{SELECT}\ X\ \mathsf{WHERE}\ P'$ by P', that is, by disregarding all projections (recall that we assume local variables in different subpatterns to have different names).

Lemma 2. *For any graph G, mapping $\mu \in [[P_{\mathit{rew}}]]_G$, variable $?y \in \mathit{Vars}(P_2^*)$ and triple patterns p^1, p^2 of P_2 that mention $?y$, if both $?y_{p^1}$ and $?y_{p^2}$ are bound by μ then $\mu(?y_{p^1}) = \mu(?y_{p^2})$.*

Let us extend, for technical reasons, the set of terms to all pairs $[b, \mu_1]$ for all blank nodes b in H_1 and mappings μ_1. Then, having this definition and Lemma 2 at hand, we can "gather" every mapping $\mu \in [[P_{\mathit{rew}}]]_G$ to its *gathering* mapping $\bar{\mu}$ as follows, for every $?y \in \mathrm{Vars}(P_2^*)$:

– $?y$ is bound in $\bar{\mu}$ if and only if one of $?y_{p^1}, \dots, ?y_{p^k}$ is bound in μ, where $p^1, \dots, p^k$ are all triple patterns of P_2 that mention $?y$;
– if i is such that $?y_{p^i}$ is bound and $?y_{p^i} = \mathsf{IRINode}(b)$ for some $b \in \mathit{blank}(H_1)$ then $\bar{\mu}(?y) = [b, \sigma(\mu|_{X_{p^i}})]$, where $X_{p^i} = x^1_{p^i}, \dots, x^n_{p^i}$ is the copy of the free variables $x^1, \dots, x^n$ of P_1 corresponding to p^i and σ is the renaming of each $x^j_{p^i}$ to x^j;
– if i is such that $?y_{p^i}$ is bound and $?y_{p^i} \neq \mathsf{IRINode}(b)$ for any $b \in \mathit{blank}(H_1)$ then $\bar{\mu}(?y) = \mu(?y_{p^i})$.

The second property of P_{rew} is as follows.

Lemma 3. *For any graph G, a mapping $\bar{\mu}$ is a gathering of some $\mu \in [[P_{\mathit{rew}}]]_G$ if and only if there exists $\mu_2 \in [[P_2^*]]_{[[Q_1]]_G}$ such that, for every $?y \in \mathit{Vars}(P_2^*)$,*
– *if $\mu_2(?y)$ is a blank node $f_{\mu_1}(b)$ created by H_1 with respect to a mapping $\mu_1 \in [[P_1]]_G$ then $\bar{\mu}(?y) = [b, \mu_1]$,*
– *otherwise $\mu_2(?y) = \bar{\mu}(?y)$.*

Having this lemma at hand, we can define the pattern $P_{\mathit{unif}}(P_{\mathit{blank}}, P_{\mathit{rew}})$. To this end, let $Y = ?y^1, \dots, ?y^m$ be all the free variables of P_2 and let P^i, $1 \leq i \leq m$, be defined as follows, taking $P^0 = P_{\mathit{blank}}\ \mathsf{AND}\ P_{\mathit{rew}}$:

$$P^i = (P^{i-1}\ \mathsf{BIND}\ V_{?y^i}\ \mathsf{AS}\ ?y^i)\ \mathsf{FILTER}\ R_{?y^i},$$

where $V_{?y^i}$ is the following condition, for $b_1, \dots, b_l$ all the blank nodes of H_1:

$$\mathsf{if}(S_{?y^i} = \mathsf{IRINode}(b_1), ?b_1, \dots \mathsf{if}(S_{?y^i} = \mathsf{IRINode}(b_l), ?b_l, S_{?y^i}) \dots),$$

$S_{?y^i}$ is as follows, for $p^1, \dots, p^k$ all the triple patterns in P_2 that uses $?y^i$:

$$\mathsf{if}(\mathsf{bound}(?y^i_{p^1}), ?y^i_{p^1}, \dots \mathsf{if}(\mathsf{bound}(?y^i_{p^{k-1}}), ?y^i_{p^{k-1}}, ?y^i_{p^k}) \dots),$$

and $R_{?y^i}$ is as follows, for free variables $?x^1, \dots, ?x^n$ of P_1 (which appear in P_{blank}):

$$\begin{aligned}&(\mathsf{bound}(?y^i_{p^1}) \wedge (?y^i_{p^1} = \mathsf{IRINode}(b_1) \vee \dots \vee ?y^i_{p^1} = \mathsf{IRINode}(b_l))\\ &\qquad\qquad\qquad \to (?x^1 \sim ?x^1_{p^1} \wedge \dots \wedge ?x^n \sim ?x^n_{p^1})) \wedge\ \cdots\ \wedge\\ &(\mathsf{bound}(?y^i_{p^k}) \wedge (?y^i_{p^k} = \mathsf{IRINode}(b_1) \vee \dots \vee ?y^i_{p^k} = \mathsf{IRINode}(b_l))\\ &\qquad\qquad\qquad \to (?x^1 \sim ?x^1_{p^k} \wedge \dots \wedge ?x^n \sim ?x^n_{p^k})).\end{aligned}$$

Then we take the following pattern as the pattern P in resulting query Q:

$$P_{\mathit{unif}}(P_{\mathit{blank}}, P_{\mathit{rew}}) = \mathsf{SELECT}\ Y\ \mathsf{WHERE}\ P^m.$$

We have the following lemma, whose immediate corollary is Theorem 1.

Lemma 4. *For any construct queries Q_1 and Q_2 and the query Q constructed on the base of Q_1 and Q_2 it holds that $[[Q_2]]_{[[Q_1]]_G} = [[Q]]_G$ for any graph G.*

5 Conclusion

For space reasons we cannot give a complete example of our construction, but we have uploaded one at `http://web.ing.puc.cl/~jreutter/construct/`.

Our results confirm the fact that nested CONSTRUCT queries are, in theory, an unnecessary feature of SPARQL, at least concerning set semantics. However, our construction introduces a blowup when translating from nested CONSTRUCT to SPARQL 1.1, which can be exponential in the depth of nesting. We conjecture that this blowup is unavoidable in the worst case. But even if this blowup was not important, the rewriting appears to be a very deep and technical translation, and unfortunately it is not easy to discover the intentions of nested CONSTRUCT queries simply by looking at their translation. For this reason we believe that nested CONSTRUCT queries are actually a desired feature of SPARQL. As a future work we would like to investigate update queries in SPARQL. Since we conjecture that update queries can be expressed as nested construct queries, it would be conceivable that the former are also a replaceable (but welcome) addition of SPARQL 1.1.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases, vol. 8. Addison-Wesley Reading (1995)
2. Angles, R., Gutierrez, C.: Subqueries in SPARQL. In: AMW'11 (2011)
3. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.C.: Composing schema mappings: Second-order dependencies to the rescue. ACM Trans. Database Syst. 30(4), 994–1055 (2005)
4. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Rec., W3C (Mar 2013)
5. Kaminski, M., Kostylev, E.V., Cuenca Grau, B.: Semantics and Expressive Power of Subqueries and Aggregates in SPARQL 1.1. In: WWW'16 (2016)
6. Kontchakov, R., Kostylev, E.V.: On expressibility of non-monotone operators in sparql. In: KR'16 (2016)
7. Kostylev, E.V., Reutter, J.L., Ugarte, M.: CONSTRUCT Queries in SPARQL. In: ICDT'15. vol. 31 (2015)
8. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems 34(3) (2009)
9. Polleres, A., Scharffe, F., Schindlauer, R.: SPARQL++ for mapping between RDF vocabularies. In: ODBASE'07. pp. 878–896 (2007)
10. Polleres, A., Wallner, J.P.: On the relation between SPARQL1.1 and answer set programming. Journal of Applied Non-Classical Logics 23(1-2), 159–212 (2013)
11. Zhang, X., Van den Bussche, J.: On the primitivity of operators in SPARQL. Inf. Process. Lett. 114(9), 480–485 (2014)