The Expressive Power of SPARQL

Renzo Angles and Claudio Gutierrez

Technical Report TR/DCC-2008-5
Department of Computer Science, Universidad de Chile
{rangles,cgutierr}@dcc.uchile.cl

Abstract. This paper studies the expressive power of SPARQL. The main result is that SPARQL and non-recursive safe Datalog with negation have equivalent expressive power, and hence, by classical results, SPARQL is equivalent from an expressiveness point of view to Relational Algebra. We present explicit generic rules of the transformations in both directions. Among other findings of the paper are the proof that negation can be simulated in SPARQL, that non-safe filters are superfluous, and that current SPARQL W3C semantics can be simplified to a standard compositional one.

1 Introduction

Determining the expressive power of a query language is crucial for understanding its capabilities and complexity, that is, what queries a user is able to pose, and how complex the evaluation of queries is, issues that are central considerations to take into account when designing a query language.

SPARQL, the query language for RDF, has recently become a W3C recommendation [9]. In the RDF Data Access Working Group (WG) were it was designed, expressiveness concerns generated ample debate. Many of them remained open due to lack of understanding of the theoretical expressive power of the language.

This paper studies in depth the expressive power of SPARQL. A first issue addressed is the incorporation of negation. The W3C specification of SPARQL provides explicit operators for join and union of graph patterns, even for specifying optional graph patterns, but it does not define explicitly the difference of graph patterns. Although intuitively it can be emulated via a combination of optional patterns and filter conditions (like negation as failure in logic programming), we show that there are several non-trivial issues to be addressed if one likes to define the difference of patterns inside the language.

A second expressiveness issue refers to graph patterns with non-safe filter, i.e., graph patterns $(P\,\text{FILTER}\,C)$ for which there are variables in C not present in P. It turns out that these type of patterns, which have non-desirable properties, can be simulated by safe ones (i.e., patterns where every variable occurring in C also occurs in P). This simple result has important consequences for defining a clean semantics, in particular a compositional and context-free one.

A third topic of concern was the presence of non desirable features in the W3C semantics like its operational character. We show that the W3C specification of the semantics of SPARQL is equivalent to a well behaved and studied compositional semantics for SPARQL, which we will denote in this paper SPARQL_{C} [6].

Using the above results, we are able to determine the expressive power of SPARQL. We prove that SPARQL_{C} and non-recursive safe Datalog with negation (nr-Datalog$^{\neg}$) are equivalent in their expressive power. For this, first we show that SPARQL_{C} is contained in nr-Datalog$^{\neg}$ by defining transformations (for databases, queries, and solutions) from SPARQL_{C} to nr-Datalog$^{\neg}$, and we prove that the result of evaluating a SPARQL_{C} query is equivalent, via the transformations, to the result of evaluating (in nr-Datalog$^{\neg}$) the transformed query. Second, we show that nr-Datalog$^{\neg}$ is contained in SPARQL_{C} using a similar approach. It is important to remark that the transformations used are explicit and simple, and in all steps bag semantics is considered.

Finally, and by far, the most important result of the paper is the proof that SPARQL has the same expressive power of Relational Algebra under bag semantics (which is the one of SPARQL). This follows from the well known fact that Relational Algebra has the same expressive power as nr-Datalog$^{\neg}$ [1].

The paper is organized as follows. In Section 2 we present preliminary material. Section 3 presents the study of negation. Section 4 studies non-safe filter patterns. Section 5 proves that the W3C specification of SPARQL and SPARQL_{C} are equivalent. Section 6 proves that SPARQL_{C} and nr-Datalog$^{\neg}$ have the same expressive power. Section 7 presents the conclusions.

Related Work. The W3C recommendation SPARQL is from January 2008. Hence, it is no surprise that little work has been done in the formal study of its expressive power. Several conjectures were raised during the WG sessions [1]. Furche et al. [3] surveyed expressive features of query languages for RDF (including old versions of SPARQL) in order to compare them systematically. But there is no particular analysis of the expressive power of SPARQL.

Cyganiak [2] presented a translation of SPARQL into Relational Algebra considering only a core fragment of SPARQL. His work is extremely useful to implement and optimize SPARQL in SQL engines. At the level of analysis of expressive issues it presented a list of problems that should be solved (many of which still persist), like the filter scope problem and the nested optional problem.

Polleres [8] proved the inclusion of the fragment of SPARQL patterns with safe filters into Datalog by giving a precise and correct set of rules. Schenk [10] proposed a formal semantics for SPARQL based on Datalog, but concentrated on complexity more than expressiveness issues. Both works do not consider bag semantics of SPARQL in their translations.

[1] See `http://lists.w3.org/Archives/Public/public-rdf-dawg-comments/`, especially the years 2006 and 2007.

The work of Perez et al. [6] and the technical report [7], that gave the formal basis for SPARQL_{C} compositional semantics, addressed several expressiveness issues, but no systematic study of the expressive power of SPARQL was done.

2 Preliminaries

2.1 RDF and Datasets

Assume there are pairwise disjoint infinite sets I, B, L (IRIs, Blank nodes, and RDF literals respectively). We denote by T the union $I \cup B \cup L$ (RDF *terms*). A tuple $(v_1, v_2, v_3) \in (I \cup B) \times I \times T$ is called an *RDF triple*, where v_1 is the *subject*, v_2 the *predicate*, and v_3 the *object*. An *RDF Graph* [4] (just graph from now on) is a set of RDF triples. Given a graph G, $\text{term}(G)$ denotes the set of elements of T occurring in G and $\text{blank}(G)$ denotes the set of blank nodes in G. The *union* of graphs, $G_1 \cup G_2$, is the set theoretical union of their sets of triples.

An *RDF dataset* D is a set $\{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ where each G_i is a graph and each u_j is an IRI. G_0 is called the *default graph* of D and it is denoted $\text{dg}(D)$. Each pair $\langle u_i, G_i\rangle$ is called a *named graph*; define $\text{name}(G_i)_D = u_i$ and $\text{gr}(u_i)_D = G_i$. We denote by $\text{term}(D)$ the set of terms occurring in the graphs of D. The set of IRIs $\{u_1, \ldots, u_n\}$ is denoted $\text{names}(D)$. Every dataset satisfies that: (i) it always contains one default graph (which could be empty); (ii) there may be no named graphs; (iii) each u_j is distinct; and (iv) $\text{blank}(G_i) \cap \text{blank}(G_j) = \emptyset$ for $i \neq j$. Finally, the *active graph* of D is the graph G_i used for querying D.

2.2 SPARQL

A SPARQL query is syntactically represented by a block consisting of a *query form* (SELECT, CONSTRUCT or DESCRIBE), zero o more *dataset clauses* (FROM and FROM NAMED), a WHERE *clause*, and possibly *solution modifiers* (e.g. DISTINCT). The WHERE clause provides a *graph pattern* to match against the RDF dataset constructed from the dataset clauses.

There are two formalizations of SPARQL which will be used throughout this study: $\text{SPARQL}_{\text{WG}}$, the W3C recommendation language SPARQL [9] and SPARQL_{C}, the formalization of SPARQL given in [6]. We will need some general definitions before describe briefly both languages.

Assume the existence of an infinite set V of variables disjoint from T. We denote by $\text{var}(\alpha)$ the set of variables occurring in the structure α. A tuple from $(I \cup L \cup V) \times (I \cup L \cup V) \times (I \cup V)$ is called a *triple pattern*. A *basic graph pattern* is a finite set of triple patterns.

A *filter constraint* is defined recursively as follows: (i) if $?X, ?Y \in V$ and $u \in I \cup L$ then $?X = u$, $?X = ?Y$, bound($?X$), isIRI($?X$), isLiteral($?X$), and isBlank($?X$) are *atomic filter constraints*[2]; (ii) if C_1 and C_2 are filter constraints then $(\neg C_1)$, $(C_1 \wedge C_2)$, and $(C_1 \vee C_2)$ are *complex filter constraints*.

[2] For a complete list of atomic filter constraints see [9].

A *mapping* μ is a partial function $\mu : V \to T$. The domain of μ, $\mathrm{dom}(\mu)$, is the subset of V where μ is defined. The *empty mapping* μ_0 is a mapping such that $\mathrm{dom}(\mu_0) = \emptyset$. Two mappings μ_1, μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, when for all $?X \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ it satisfies that $\mu_1(?X) = \mu_2(?X)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping. The expression $\mu_{?X \to v}$ denote a mapping such that $\mathrm{dom}(\mu) = \{?X\}$ and $\mu(?X) = v$

Let C_1 and C_2 be filter constrains. The evaluation of a filter constraint C against a mapping μ, denoted $\mu(C)$, is defined in a three value logic with values $\{$*true*, *false*, *error*$\}$ as follows:

- if C is an atomic filter constraint, then
  - if $\mathrm{var}(C) \subseteq \mathrm{dom}(\mu)$ then
    $\mu(C) =$ *true* when
    - C is $?X = u$ and $\mu(?X) = u$; or
    - C is $?X = ?Y$ and $\mu(?X) = \mu(?Y)$; or
    - C is $\mathrm{isIRI}(?X)$ and $\mu(?X) \in I$; or
    - C is $\mathrm{isLiteral}(?X)$ and $\mu(?X) \in L$; or
    - C is $\mathrm{isBlank}(?X)$ and $\mu(?X) \in B$; or
    - C is $\mathrm{bound}(?X)$;

    and $\mu(C) =$ *false* otherwise.
  - if $\mathrm{var}(C) \nsubseteq \mathrm{dom}(\mu)$ then
    - if C is $\mathrm{bound}(?X)$ then $\mu(C) =$ *false* else $\mu(C) =$ *error*.[3]
- if C is a complex filter constraint, then $\mu(C)$ is defined as follows:

$\mu(C_1)$	$\mu(C_2)$	$\mu(C_1) \wedge \mu(C_2)$	$\mu(C_1) \vee \mu(C_2)$
true	*true*	*true*	*true*
true	*false*	*false*	*true*
true	*error*	*error*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*
false	*error*	*false*	*error*
error	*true*	*error*	*true*
error	*false*	*false*	*error*
error	*error*	*error*	*error*

$\mu(C_1)$	$\neg\mu(C_1)$
true	*false*
false	*true*
error	*error*

A mapping μ satisfies a filter constraint C, denoted $\mu \models C$, iff $\mu(C) =$ *true*. Consider the following operations between two sets of mappings Ω_1, Ω_2:

$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}$

$\Omega_1 \bowtie_C \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \text{ and } (\mu_1 \cup \mu_2) \models C\}$

$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$

$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible } \}$

$\Omega_1 \setminus_C \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2,\ \mu_1 \text{ and } \mu_2 \text{ are not compatible } \} \cup$
$\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1,\ (\mu_1 \cup \mu_2) \not\models C\}$

$\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

$\Omega_1 ⟕_C \Omega_2 = (\Omega_1 \bowtie_C \Omega_2) \cup (\Omega_1 \setminus_C \Omega_2)$

[3] Functions invoked with an argument of the wrong type are evaluated to *error*.

Table 1. Semantics of SPARQL$_{\mathrm{C}}$ graph patterns. P_1, P_2 are SPARQL$_{\mathrm{C}}$ graph patterns, C is a filter constraint, $u \in I$ and $?X \in V$.

Graph pattern P	Evaluation $[\![P]\!]_G^D$
$(P_1 \text{ AND } P_2)$	$[\![P_1]\!]_G^D \bowtie [\![P_2]\!]_G^D$
$(P_1 \text{ OPT } P_2)$	$[\![P_1]\!]_G^D ⟕ [\![P_2]\!]_G^D$
$(P_1 \text{ UNION } P_2)$	$[\![P_1]\!]_G^D \cup [\![P_2]\!]_G^D$
$(P_1 \text{ FILTER } C)$	$\{\mu \mid \mu \in [\![P_1]\!]_G^D \text{ and } \mu \models C\}$
$(u \text{ GRAPH } P_1)$	$[\![P_1]\!]_{\mathrm{gr}(u)_D}^D$
$(?X \text{ GRAPH } P_1)$	$\bigcup_{v \in \mathrm{names}(D)} ([\![P_1]\!]_{\mathrm{gr}(v)_D}^D \bowtie \{\mu_{?X \to v}\})$

Syntax and Semantics of SPARQL$_{\mathrm{C}}$.
A SPARQL$_{\mathrm{C}}$ graph pattern `P` is defined recursively by the following grammar:

```
P  ::= t | "(" GP ")"
GP ::= P "AND" P | P "UNION" P | P "OPT" P | P "FILTER" C | n "GRAPH" P
```

where `t` denotes a triple pattern, `C` denotes a filter constraint, and `n` $\in I \cup V$.

The evaluation of a SPARQL$_{\mathrm{C}}$ graph pattern P over an RDF dataset D having active graph G, denoted $[\![P]\!]_G^D$ (or $[\![P]\!]$ where D and G are clear from the context), is defined recursively as follows:

- if P is a triple pattern t, $[\![P]\!]_G^D = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{var}(t) \text{ and } \mu(t) \in G\}$ where $\mu(t)$ is the triple obtained by replacing the variables in t according to mapping μ.
- if P is a complex graph pattern then $[\![P]\!]_G^D$ is defined as given in Table 1.

Syntax and Semantics of SPARQL$_{\mathrm{WG}}$.
A SPARQL$_{\mathrm{WG}}$ graph pattern `GroupGP` is defined by the following grammar[4]:

```
GroupGP         ::= "{" TB? ((GPNotTriples | Filter) "."? TB?)* "}"
GPNotTriples    ::= OptionalGP | GroupOrUnionGP | GraphGP
OptionalGP      ::= "OPTIONAL" GroupGP
GraphGP         ::= "GRAPH" VarOrIRIref GroupGP
GroupOrUnionGP ::= GroupGP ( "UNION" GroupGP )*
Filter          ::= "FILTER" Constraint
```

where `TB` denotes a basic graph pattern (a set of triple patterns), `VarOrIRIref` denotes a term in the set $I \cup V$ and `Constraint` denotes a filter constraint. Note that the operator {`A . B`} represents the AND but it has not fixed arity.

[4] `http://www.w3.org/TR/rdf-sparql-query/#grammar`. We use `GP` and `TB` to abbreviate `GraphPattern` and `TriplesBlock` respectively

The evaluation of a $\text{SPARQL}_{\text{WG}}$ graph pattern `GroupGP` is defined by a series of steps, starting by transforming `GroupGP`, via a function T, into an intermediate algebra expression E (with operators BGP, Join, Union, LeftJoin, Graph and Filter), and finally evaluating E on an RDF dataset D.

The transformation T(`GroupGP`) is given by Algorithm 1. The evaluation of $E = T(\texttt{GroupGP})$ over an RDF dataset D having active graph G, which we will denote $\langle\langle E \rangle\rangle_G^D$ (or $\langle\langle E \rangle\rangle$ where D and G are clear from the context)[5], is defined recursively as follows:

- if E is BGP(`TB`), $\langle\langle E \rangle\rangle_G^D = \{\mu \mid \text{dom}(\mu) = \text{var}(E) \text{ and } \mu(E) \subseteq G\}$ where $\mu(E)$ is the set of triples obtained by replacing the variables in the triple patterns of `TB` according to mapping μ.
- if E is a complex expression then $\langle\langle E \rangle\rangle_G^D$ is defined as given in Table 2.

Note 1. In the definition of graph patterns, we avoided blank nodes, because this restriction does not diminish the generality of our study. In fact, each SPARQL query Q can be simulated by a SPARQL query Q' without blank nodes in its pattern. It follows from the definitions of RDF instance mapping, solution mapping, and the order of evaluation of solution modifiers (see [9]), that if Q is a query with graph pattern P, and Q' is the same query where each blank node b in P has been replaced by a fresh variable $?X_b$ then Q and Q' give the same results. (Note that, if Q has the query form `SELECT` or `DESCRIBE`, the "$*$" parameter is –according to the specification of SPARQL– an abbreviation for all variables occurring in the pattern. In this case the query Q' should explicit in the SELECT clause all variables of the original pattern P.)

Note 2. SPARQL_{C} follows a compositional semantics, whereas $\text{SPARQL}_{\text{WG}}$ follows a mixture of compositional and operational semantics where the meaning of certain patterns depends on their context, e.g., lines 7 and 8 in algorithm 1.

Note 3. In this paper we will follow the simpler syntax of SPARQL_{C}, better suited to do formal analysis and processing than the syntax presented by $\text{SPARQL}_{\text{WG}}$. There is an easy and intuitive way of translating back and forth between both syntax formalisms, which we will not detail here.

[5] The evaluation function in $\text{SPARQL}_{\text{WG}}$ is originally denoted eval$(D(G), E)$ in [9].

Algorithm 1 Transformation of $\text{SPARQL}_{\text{WG}}$ patterns into algebra expressions.

// Input: a $\text{SPARQL}_{\text{WG}}$ graph pattern `GroupGP`
// Output: an algebra expression $E = T(\texttt{GroupGP})$
$E \leftarrow$ empty pattern; $FS \leftarrow \emptyset$
for each syntactic form f in `GroupGP` **do**
  if f is `TB` **then** $E \leftarrow \text{Join}(E, \text{BGP}(\texttt{TB}))$
  if f is `OPTIONAL GroupGP`$_1$ **then**
    if $T(\texttt{GroupGP}_1)$ is $\text{Filter}(F, E')$ **then** $E \leftarrow \text{LeftJoin}(E, E', F)$
    else $E \leftarrow \text{LeftJoin}(E, T(\texttt{GroupGP}_1), true)$
  if f is `GroupGP`$_1$ `UNION` $\cdots$ `UNION GroupGP`$_n$ **then**
    if $n > 1$ **then**
      $E' \leftarrow \text{Union}(\cdots(\text{Union}(T(\texttt{GroupGP}_1), T(\texttt{GroupGP}_2))\cdots), T(\texttt{GroupGP}_n))$
    else $E' \leftarrow T(\texttt{GroupGP}_1)$
    $E \leftarrow \text{Join}(E, E')$
  end if
  if f is `GRAPH VarOrIRIref GroupGP`$_1$ **then**
    $E \leftarrow \text{Join}(E, \text{Graph}(\texttt{VarOrIRIref}, T(\texttt{GroupGP}_1)))$
  if f is `FILTER constraint` **then** $FS \leftarrow (FS \wedge \texttt{constraint})$
end for
if $FS \neq \emptyset$ **then** $E \leftarrow \text{Filter}(FS, E)$
return E

Table 2. Semantics of $\text{SPARQL}_{\text{WG}}$ graph patterns. A pattern `GroupGP` is transformed into an algebra expression E using algorithm 1. Then E is evaluated as the table shows. E_1 and E_2 are algebra expressions, C is a filter constraint, $u \in I$ and $?X \in V$.

Algebra Expression E	Evaluation $\langle\langle E \rangle\rangle_G^D$
$\text{Join}(E_1, E_2)$	$\langle\langle E_1 \rangle\rangle_G^D \bowtie \langle\langle E_2 \rangle\rangle_G^D$
$\text{LeftJoin}(E_1, E_2, C)$	$\langle\langle E_1 \rangle\rangle_G^D ⟕_C \langle\langle E_2 \rangle\rangle_G^D$
$\text{Union}(E_1, E_2)$	$\langle\langle E_1 \rangle\rangle_G^D \cup \langle\langle E_2 \rangle\rangle_G^D$
$\text{Filter}(C, E_1)$	$\{\ \mu \mid \mu \in \langle\langle E_1 \rangle\rangle_G^D \text{ and } \mu \models C\}$
$\text{Graph}(u, E_1)$	$\langle\langle E_1 \rangle\rangle_{\text{gr}(u)_D}^D$
$\text{Graph}(?X, E_1)$	$\bigcup_{v\ \in\ \text{names}(D)}(\langle\langle E_1 \rangle\rangle_{\text{gr}(v)_D}^D \bowtie \{\mu_{?X \to v}\})$

2.3 Datalog

We will briefly review notions of Datalog (For further details and proofs see [1,5]).

A *term* is either a variable or a constant. An *atom* is either a *predicate formula* $p(x_1, ..., x_n)$ where p is a predicate name and each x_i is a term, or an *equality formula* $t_1 = t_2$ where t_1 and t_2 are terms. A *literal* is either an atom (a *positive literal* L) or the negation of an atom (a *negative literal* $\neg L$).

A Datalog *rule* is an expression of the form $L \leftarrow L_1, \ldots, L_n$ where L is a positive literal called the *head*[6] of the rule and $L_1, \ldots, L_n$ is a set of literals called the *body*. A rule is *ground* if it does not have any variables. A ground rule with empty body is called a *fact*.

A *Datalog program* Π is a finite set of Datalog rules. The set of facts occurring in Π, denoted facts(Π), is called the *initial database* of Π. A predicate is *extensional* in Π if it occurs only in facts(Π), otherwise it is called *intensional*.

A variable x occurs positively in a rule r if and only if x occurs in a positive literal L in the body of r such that: (1) L is a predicate formula; (2) if L is $x = c$ then c is a constant; (3) if L is $x = y$ or $y = x$ then y is a variable occurring positively in r. A Datalog rule r is said to be *safe* if all the variables occurring in the literals of r (including the head of r) occur positively in r. A Datalog program Π is *safe* if all the rules of Π are safe. The safety restriction provides a syntactic restriction of programs which enforces the finiteness of derived predicates.

The *dependency graph* of a Datalog program Π is a digraph (N, E) where the set of nodes N is the set of predicates that occur in the literals of Π, and there is an arc (p_1, p_2) in E if there is a rule in Π whose body contains predicate p_1 and whose head contains predicate p_2. A Datalog program is said to be *recursive* if its dependency graph is cyclic, otherwise it is said to be *non-recursive*.

Hence, a Datalog program is *non-recursive* and *safe* if it does not contain any predicate that is recursive in the program and it can only generate a finite number of answers. In what follows, we only consider non-recursive and safe Datalog programs.

A *substitution* θ is a set of assignments $\{x_1/t_1, \ldots, x_n/t_n\}$ where each x_i is a variable and each t_i is a term. Given a rule r, we denote by $\theta(r)$ the rule resulting of substituting the variable x_i for the term t_i in each literal of r. A substitution is *ground* if every term t_i is a constant.

A rule r in a Datalog program Π is true with respect to a ground substitution θ, if for each literal L in the body of r one of the following conditions is satisfied:
(i) $\theta(L) \in$ facts(Π); (ii) $\theta(L)$ is an equality $t = t$ where t is a constant;
(iii) $\theta(L)$ is a literal of the form $\neg p(c_1, ..., c_n)$ and $p(c_1, ..., c_n) \notin$ facts(Π);
(iv) $\theta(L)$ is a literal of the form $\neg(c_1 = c_2)$ and c_1 and c_2 are distinct constants.

The *meaning* of a Datalog program Π, denoted facts$^*(\Pi)$, is the database resulting from adding to the initial database of Π as many new facts of the form $\theta(L)$ as possible, where θ is a substitution that makes a rule r in Π true and L is the head of r. Then the rules are applied repeatedly and new facts are added to the database until this iteration stabilizes, i.e., until a *fixpoint* is reached.

A *Datalog query* Q is a pair (Π, L) where Π is a Datalog program and L is a positive (goal) literal. The *answer* to Q over database $D =$ facts(Π), denoted ans$_d(Q, D)$ is defined as the set of substitutions $\{\theta \mid \theta(L) \in \text{facts}^*(\Pi)\}$.

[6] We may assume that all heads of rules have only variables by adding the corresponding equality formula to its body.

2.4 Comparing Expressive Power of Languages

By the *expressive power* of a query language, we understand the set of all queries expressible in that language [1,5]. In order to determine the expressive power of a query language L, usually one chooses a well-studied query language L' and compares L and L' in their expressive power. Two query languages have the same expressive power if they express exactly the same set of queries.

A given query language is defined as a quadruple $(\mathcal{Q}, \mathcal{D}, \mathcal{S}, \text{eval})$, where $\mathcal{Q}$ is a set of queries, $\mathcal{D}$ is a set of databases, $\mathcal{S}$ is a set of solutions, and $\text{eval} : \mathcal{Q} \times \mathcal{D} \to \mathcal{S}$ is the evaluation function. The evaluation of a query $Q \in \mathcal{Q}$ on a database $D \in \mathcal{D}$ is denoted $\text{eval}(Q, D)$ (usually $\text{eval}(Q, D)$ is simply denoted $Q(D)$ if no confusion arises). Two queries $Q_1, Q_2 \in \mathcal{Q}$ are *equivalent*, denoted $Q_1 \equiv Q_2$, if $\text{eval}(Q_1, D) = \text{eval}(Q_2, D)$ for every $D \in \mathcal{D}$, i.e., they return the same answer for all input databases.

Let $L_1 = (\mathcal{Q}_1, \mathcal{D}_1, \mathcal{S}_1, \text{eval}_1)$ and $L_2 = (\mathcal{Q}_2, \mathcal{D}_2, \mathcal{S}_2, \text{eval}_2)$ be two query languages. We say that L_1 is *contained* in L_2 if and only if there are bijective data transformations $\mathcal{T}_D : \mathcal{D}_1 \to \mathcal{D}_2$ and $\mathcal{T}_S : \mathcal{S}_1 \to \mathcal{S}_2$, and query transformation $\mathcal{T}_Q : \mathcal{Q}_1 \to \mathcal{Q}_2$, such that for all $Q \in \mathcal{Q}_1$ and $D \in \mathcal{D}_1$ it satisfies that $\mathcal{T}_S(\text{eval}_1(Q, D)) = \text{eval}_2(\mathcal{T}_Q(Q), \mathcal{T}_D(D))$. We say that L_1 and L_2 are *equivalent* if and only if L_1 is contained in L_2 and L_2 is contained in L_1. (Note that if L_1 and L_2 are subsets of a language L, then $\mathcal{T}_D$, $\mathcal{T}_S$ and $\mathcal{T}_Q$ are the identity.)

3 Expressing Difference of Patterns in $\text{SPARQL}_{\text{WG}}$

The $\text{SPARQL}_{\text{WG}}$ specification indicates that it is possible to test if a graph pattern does not match a dataset, via a combination of optional patterns and filter conditions (like negation as failure in logic programming)([9] Sec. 11.4.1). In this section we analyze in depth the scope and limitations of this approach.

We will introduce a syntax for the "difference" of two graph patterns P_1 and P_2, denoted $(P_1 \,\text{MINUS}\, P_2)$, with the intended informal meaning: "the set of mappings that match P_1 and does not match P_2". Formally:

Definition 1. *Let P_1, P_2 be graph patterns and D be a dataset with active graph G. Then*

$$\langle\!\langle (P_1 \,\text{MINUS}\, P_2) \rangle\!\rangle_G^D = \langle\!\langle P_1 \rangle\!\rangle_G^D \setminus \langle\!\langle P_2 \rangle\!\rangle_G^D.$$

A *naive implementation* of the MINUS operator in terms of the other operators would be the graph pattern $((P_1 \,\text{OPT}\, P_2) \,\text{FILTER}\, C)$ where C is the filter constraint $(\neg\, \text{bound}(?X))$ for some variable $?X \in \text{var}(P_2) \setminus \text{var}(P_1)$. This means that for each mapping $\mu \in \langle\!\langle (P_1 \,\text{OPT}\, P_2) \rangle\!\rangle_G^D$ at least one variable $?X$ occurring in P_2, but not occurring in P_1, does not match (i.e., $?X$ is unbounded). There are two problems with this solution:

- Variable $?X$ cannot be an arbitrary variable. For example, P_2 could be in turn an optional pattern $(P_3 \,\text{OPT}\, P_4)$ where only variables in P_3 are relevant.
- If $\text{var}(P_2) \setminus \text{var}(P_1) = \emptyset$ there is no variable $?X$ to check unboundedness.

The above two problems motivate the introduction of the notions of non-optional variables and copy patterns.

The set of *non-optional variables* of a graph pattern P, denoted $\mathrm{nov}(P)$, is a subset of the variables of P defined recursively as follows: $\mathrm{nov}(P) = \mathrm{var}(P)$ when P is a basic graph pattern; if P is either $(P_1 \,\mathrm{AND}\, P_2)$ or $(P_1 \,\mathrm{UNION}\, P_2)$ then $\mathrm{nov}(P) = \mathrm{nov}(P_1) \cup \mathrm{nov}(P_2)$; if P is $(P_1 \,\mathrm{OPT}\, P_2)$ then $\mathrm{nov}(P) = \mathrm{nov}(P_1)$; if P is $(n \,\mathrm{GRAPH}\, P_1)$ then either $\mathrm{nov}(P) = \mathrm{nov}(P_1)$ when $n \in I$ or $\mathrm{nov}(P) = \mathrm{nov}(P_1) \cup \{n\}$ when $n \in V$; and $\mathrm{nov}(P_1 \,\mathrm{FILTER}\, C) = \mathrm{nov}(P_1)$. Intuitively $\mathrm{nov}(P)$ contains the variables that necessarily must be bounded in any mapping of P.

Let $\phi : V \to V$ be a variable-renaming function. Given a graph pattern P, a *copy pattern* $\phi(P)$ is an isomorphic copy of P whose variables have been renamed according to ϕ and satisfying that $\mathrm{var}(P) \cap \mathrm{var}(\phi(P)) = \emptyset$.

Theorem 1. *Let P_1 and P_2 be graph patterns. Then:*

$$(P_1 \,\mathrm{MINUS}\, P_2) \equiv ((P_1 \,\mathrm{OPT}((P_2 \,\mathrm{AND}\, \phi(P_2)) \,\mathrm{FILTER}\, C_1)) \,\mathrm{FILTER}\, C_2) \qquad (1)$$

where:

- C_1 *is the filter constraint* $(?X_1 = ?X'_1 \wedge \cdots \wedge ?X_n = ?X'_n)$ *where* $?X_i \in \mathrm{var}(P_2)$ *and* $?X'_i = \phi(?X_i)$ *for* $1 \leq i \leq n$.
- C_2 *is the filter constraint* $(\neg\, \mathrm{bound}(?X'))$ *for some* $?X' \in \mathrm{nov}(\phi(P_2))$.

Proof. Let P be the graph pattern $(P_1 \,\mathrm{MINUS}\, P_2)$ and P' be the right hand side of (1). We will prove that for every dataset D with active graph G, it satisfies that $\langle\langle P \rangle\rangle^D_G = \langle\langle P' \rangle\rangle^D_G$.

(a) *Evaluation* $\langle\langle P \rangle\rangle$*:* By definition, $\langle\langle P \rangle\rangle = \langle\langle P_1 \rangle\rangle \setminus \langle\langle P_2 \rangle\rangle$. Then, a mapping μ is in $\langle\langle P \rangle\rangle$ if and only if $\mu \in \langle\langle P_1 \rangle\rangle$ and for every mapping $\mu' \in \langle\langle P_2 \rangle\rangle$, μ and μ' are not compatible.

(b) *Evaluation* $\langle\langle P' \rangle\rangle$*:* To simplify the idea of the proof, we reduce P' to the graph pattern $((P_1 \,\mathrm{OPT}\, P_2) \,\mathrm{FILTER}\, C_2)$ where C_2 is $(\neg\, \mathrm{bound}(?X))$ for some $?X \in \mathrm{nov}(P_2) \setminus \mathrm{var}(P_1)$. Note that this reduction does not diminish the generality of the proof because $\phi(P_2)$ and C_1 were added into P' to solve the case when $\mathrm{nov}(P_2) \setminus \mathrm{var}(P_1) = \emptyset$ (See Note 4 later).
Here, a mapping μ is in $\langle\langle P' \rangle\rangle$ if and only if $\mu \in \langle\langle (P_1 \,\mathrm{OPT}\, P_2) \rangle\rangle$ and $\mu \models C_2$. Given $\mu_1 \in \langle\langle P_1 \rangle\rangle$, it holds that $\mu \in \langle\langle (P_1 \,\mathrm{OPT}\, P_2) \rangle\rangle$ iff either (i) $\mu = \mu_1 \cup \mu_2$ for some $\mu_2 \in \langle\langle P_2 \rangle\rangle$ compatible with μ_1; or (ii) $\mu = \mu_1$ and for every $\mu_2 \in \langle\langle P_2 \rangle\rangle$, μ_1 and μ_2 are not compatible. Note that, in case (i), $\mu(?X)$ is bounded for every variable $?X \in \mathrm{nov}(P_2)$ and, in case (ii), $\mu(?X)$ is unbounded for every variable $?X \in \mathrm{nov}(P_2) \setminus \mathrm{var}(P_1)$. Given that C_2 contains the filter constraint $(\neg\, \mathrm{bound}(?X))$ for some variable $?X \in \mathrm{nov}(P_2) \setminus \mathrm{var}(P_1)$, only case (ii) satisfies the condition $\mu \models C_2$ (Note that here is critical the fact that $?X$ is a safe variable occurring in P_2 but not in P_1).
Then, $\langle\langle P' \rangle\rangle$ will only contain mappings from case (ii), that is each mapping $\mu \in \langle\langle P' \rangle\rangle$ satisfies that $\mu = \mu_1 \in \langle\langle P_1 \rangle\rangle$ and for every mapping $\mu_2 \in \langle\langle P_2 \rangle\rangle$, μ_1 and μ_2 are not compatible.
Therefore, $\langle\langle P' \rangle\rangle$ has exactly the same mappings as the evaluation of $\langle\langle P \rangle\rangle$ in (a), and we conclude the proof.

Note 4 (Why the copy pattern $\phi(P)$ is necessary?).
Consider the naive implementation for $(P_1 \,\text{MINUS}\, P_2)$, that is the graph pattern $((P_1 \,\text{OPT}\, P_2) \,\text{FILTER}\, C)$ where C is the filter constraint $(\neg\, \text{bound}(?X))$ for some $?X \in \text{var}(P_2) \setminus \text{var}(P_1)$.

Note that the above implementation would fail when $\text{var}(P_2) \setminus \text{var}(P_1) = \emptyset$, because *there exist no variables* to check unboundedness. For example, consider the graph patterns $P_1 = (?X, \text{name}, ?N)$ and $P_2 = (?X, \text{lastname}, \text{"Perez"})$. The naive implementation of $(P_1 \,\text{MINUS}\, P_2)$ will give a pattern with filter condition $C = \emptyset$ because there are no variables in $\text{var}(P_2) \setminus \text{var}(P_1)$ (Note that it is not possible to use variable $?X$ to check unboundedness when evaluating P_2 because –to satisfy the entire pattern– variable $?X$ must have already been bound in the evaluation of pattern P_1).

To solve this problem, P_2 is replaced by $((P_2 \,\text{AND}\, \phi(P_2)) \,\text{FILTER}\, C_1)$ where $\phi(P_2)$ is a copy of P_2 whose variables have been renamed and whose relations of equality with the original ones are in condition C_1. Then we can use some variable from $\phi(P_2)$ to check if graph pattern P_2 does not match. The copy pattern ensures that there will exist a variable to check unboundedness.

Then, the implementation of $(P_1 \,\text{MINUS}\, P_2)$ in the example will be

$$
\begin{aligned}
&(((?X, \text{name}, ?N) \,\text{OPT} \\
&\qquad (((?X, \text{lastname}, \text{"Perez"}) \,\text{AND} (?X', \text{lastname}, \text{"Perez"})) \\
&\qquad\qquad \text{FILTER}(?X = ?X'))) \,\text{FILTER}(\neg\, \text{bound}(?X'))),
\end{aligned}
$$

where variable $?X' \in \phi(P_2)$ has been selected to check unboundedness.

Note that the inclusion of copy patterns could introduce an exponential blow-up in the size of the pattern. A possible optimization (still inside the syntax of SPARQL) is to select a *safe triple pattern* t of P_2, i.e., a triple pattern having only safe variables (at least one), and using the copy pattern $\phi(t)$ instead of the entire copy pattern $\phi(P_2)$.

Note 5 (Why non-optional variables?).
Consider the graph pattern

$$P = ((?X, \text{name}, ?N) \,\text{MINUS}((?X, \text{knows}, ?Y) \,\text{OPT}(?Y, \text{mail}, ?Z))).$$

The naive implementation of P would be the graph pattern

$$P' = ((P_1 \,\text{OPT}\, P_2) \,\text{FILTER}(\neg\, \text{bound}(?Z))),$$

where $P_1 = (?X,\text{name}, ?N)$, $P_2 = ((?X,\text{knows}, ?Y) \,\text{OPT}(?Y,\text{mail}, ?Z))$ and $?Z$ is the variable selected to check unboundedness. (Note that variable $?Y$ could also have been selected because $?Y \in \text{var}(P_2) \setminus \text{var}(P_1)$.)

Additionally, consider the RDF graph

$$
\begin{aligned}
G = \{\ &(\text{a},\text{name},n_a),\ (\text{b},\text{name},n_b),\ (\text{b},\text{knows},\text{c}),\ (\text{b},\text{mail},m_b), \\
&(\text{c},\text{name},n_c),\ (\text{c},\text{knows},\text{d}),\ (\text{d},\text{name},n_d),\ (\text{d},\text{mail},m_d)\ \}.
\end{aligned}
$$

Let $P_2 = (P_3 \,\mathrm{OPT}\, P_4)$ where $P_3 = (?X,\text{knows}, ?Y)$ and $P_4 = (?Y,\text{mail}, ?Z)$. Consider the following evaluations over graph G:

$[\![P_1]\!]_G =$

?X	?N
a	n_a
b	n_b
c	n_c
d	n_d

$[\![P_3]\!]_G =$

?X	?Y
b	c
c	d

$[\![P_4]\!]_G =$

?Y	?Z
b	m_b
d	m_d

$[\![P_2]\!]_G =$

?X	?Y	?Z
b	c	
c	d	m_d

$[\![(P_1 \,\mathrm{OPT}\, P_2)]\!]_G =$

?X	?N	?Y	?Z
a	n_a		
b	n_b	c	
c	n_c	d	m_d
d	n_d		

Then $P = (P_1 \,\mathrm{MINUS}\, P_2)$ and $P' = ((P_1 \,\mathrm{OPT}\, P_2)\,\mathrm{FILTER}(\neg\,\mathrm{bound}(?Z)))$ are evaluated as follows:

$[\![P]\!]_G =$

?X	?N
a	n_a
d	n_d

$[\![P']\!]_G =$

?X	?N	?Y	?Z
a	n_a		
b	n_b	c	
d	n_d		

Note that the evaluation of graph pattern P' differs from that of pattern P. To see the problem recall the informal semantics: a mapping μ matches pattern P if and only if μ matches P_1 and μ does not match P_2. This latter condition means: it is false that every variable in P_2 (but not in P_1) is bounded. But to say "every variable" is not correct in this context because P_2 contains the optional pattern $(?Y,\text{mail}, ?Z)$, and its variables could be unbounded for some valid solutions of P_2. Hence the problem is produced by the expression $(\neg\,\mathrm{bound}(?Z))$, because the bounding state of variable $?Z$ introduces noise when testing if pattern P_2 gets matched.

In fact, consider the mapping μ such that $\mu(?X) = b$, $\mu(?N) = n_b$ and $\mu(?Y) = c$. This mapping is not a solution for P because it matches P_2 since it matches $(?X,\text{knows}, ?Y)$ although it does not match the optional pattern $(?Y,\text{mail}, ?Z)$. On the other hand, we have that μ matches P' because it matches $(P_1 \,\mathrm{OPT}\, P_2)$ and μ satisfies the filter constraint $(\neg\,\mathrm{bound}(?Z))$.

Now, if we ensure the selection of a "non-optional variable" to check unboundedness when transforming P, we have that $?Y$ is the unique non-optional variable occurring in P_2 but not occurring in P_1, i.e., variable $?Y$ works exactly as the test to check if a mapping matching P_1 matches P_2 as well. Hence, instead of P', the graph pattern

$$P'' = ((P_1 \,\mathrm{OPT}\, P_2)\,\mathrm{FILTER}(\neg\,\mathrm{bound}(?Y)))$$

is the one that expresses faithfully the graph pattern $(P_1 \,\mathrm{MINUS}\, P_2)$, and in fact, the evaluation of P'' gives exactly the same set of mappings as P.

4 Avoiding Unsafe Patterns in $\text{SPARQL}_{\text{WG}}$

One influential point in the evaluation of patterns in $\text{SPARQL}_{\text{WG}}$ is the behavior of *filters*. What is the scope of a filter? What is the meaning of a filter having variables that do not occur in the graph pattern to be filtered?

It was proposed in [6] that for reasons of simplicity for the user and cleanness of the semantics, the scope of filters should be the expression which they filter, and free variables should be disallowed in the filter condition. Formally, a graph pattern of the form $(P\,\text{FILTER}\,C)$ is said to be *safe* if $\text{var}(C) \subseteq \text{var}(P)$. In [6] only safe filter patterns were allowed in the syntax, and hence the scope of the filter C is the pattern P which defines the filter condition. This approach is further supported by the fact that non-safe filters are rare in practice.

The WG decided to follow a different approach, and defined the scope of a filter condition C to be a case-by-case and context-dependent feature:

1. The scope of a filter is defined as follows: a filter "is a restriction on solutions over the whole group in which the filter appears".
2. There is one exception, though, when filters combine with optionals. If a filter expression C belongs to the group graph pattern of an optional, the scope of C is local to the group where the optional belongs to. This is reflected in lines 7 and 8 of Algorithm 1.

The complexities that this approach brings were recognized in the discussion of the WG, and can be witnessed by the reader by following the evaluation of patterns in $\text{SPARQL}_{\text{WG}}$.

Let $\text{SPARQL}_{\text{WG}}^{\text{Safe}}$ be the subset of queries of $\text{SPARQL}_{\text{WG}}$ having only filter-safe patterns. In what follows, we will show that, in $\text{SPARQL}_{\text{WG}}$, non-safe filters are superfluous, and hence its non-standard and case-by-case semantics can be avoided. In fact, we will prove that non-safe filters do not add expressive power to the language, or in other words, that $\text{SPARQL}_{\text{WG}}$ and $\text{SPARQL}_{\text{WG}}^{\text{Safe}}$ have the same expressive power, that is, for each graph pattern P there is a filter-safe graph pattern $P' = \text{safe}(P)$ which computes exactly the same mappings as P.

The transformation $\text{safe}(P)$ is given by Algorithm 2. This algorithm works as the identity for most patterns. The key part is the treatment of patterns which combine filters and optionals. Line 9 is exactly the codification of the $\text{SPARQL}_{\text{WG}}$ evaluation of filters inside optionals. For non-safe filters (see lines 15-20), it replaces each atomic filter condition C', where a free variable occurs, by either an expression *false* when C' is $\text{bound}(\cdot)$; or an expression $\text{bound}(a)$ otherwise. (Note that $\text{bound}(a)$ is evaluated to *error* because a is a constant.)

Note 6 (On Algorithm 2). The expression in line 9 must be refined for bag semantics to the expression:

$$\begin{aligned}
P' \leftarrow (\ & \text{safe}((P_1\,\text{AND}\,P_3)\,\text{FILTER}\,C)\ \text{UNION}\\
& (\text{safe}(P_1)\,\text{MINUS}\,\text{safe}(P_3))\ \text{UNION}\\
& ((\text{safe}(P_1)\,\text{MINUS}(\text{safe}(P_1)\,\text{MINUS}\,\text{safe}(P_3)))\\
& \qquad \text{MINUS}\,\text{safe}((P_1\,\text{AND}\,P_3)\,\text{FILTER}\,C))\ \ )
\end{aligned}$$

Algorithm 2 Transformation of a general graph pattern into a safe pattern.

1: // Input: a SPARQL$_{\text{WG}}$ graph pattern P
2: // Output: a safe graph pattern $P' \leftarrow \text{safe}(P)$
3: $P' \leftarrow \emptyset$
4: **if** P is $(P_1 \,\text{AND}\, P_2)$ **then** $P' \leftarrow (\text{safe}(P_1) \text{ AND } \text{safe}(P_2))$
5: **if** P is $(P_1 \,\text{UNION}\, P_2)$ **then** $P' \leftarrow (\text{safe}(P_1) \,\text{UNION}\, \text{safe}(P_2))$
6: **if** P is $(n \,\text{GRAPH}\, P_1)$ **then** $P' \leftarrow (n \,\text{GRAPH}\; \text{safe}(P_1))$
7: **if** P is $(P_1 \,\text{OPT}\, P_2)$ **then**
8:   **if** P_2 is $(P_3 \,\text{FILTER}\, C)$ **then**
9:     $P' \leftarrow (\text{safe}(P_1) \,\text{OPT}(\text{safe}((P_1 \,\text{AND}\, P_3) \,\text{FILTER}\, C)))$
10:   **else** $P' \leftarrow (\text{safe}(P_1) \,\text{OPT}\, \text{safe}(P_2))$
11: **end if**
12: **if** P is $(P_1 \,\text{FILTER}\, C)$ **then**
13:   **if** $\text{var}(C) \subseteq \text{var}(\text{safe}(P_1))$ **then** $P' \leftarrow (\text{safe}(P_1) \,\text{FILTER}\, C)$
14:   **else**
15:     **for all** $?X \in \text{var}(C)$ and $?X \notin \text{var}(\text{safe}(P_1))$ **do**
16:       **for all** atomic filter constraint C' in C
17:        **if** C' is $(?X = u)$ or $(?X = ?Y)$ or isIRI$(?X)$ or isBlank$(?X)$ or isLiteral$(?X)$
18:          Replace in C the constraint C' by bound(a) //where a is a constant
19:        **else if** C' is bound$(?X)$ **then**
20:          Replace in C the constraint C' by *false*
21:       **end for**
22:     **end for**
23:     $P' \leftarrow (\text{safe}(P_1) \,\text{FILTER}\, C)$
24:   **end if**
25: **end if**
26: **return** P'

Lemma 1. *For every SPARQL$_{_{WG}}$ graph pattern P, the pattern* safe(P) *is filter-safe and it holds* $\langle\!\langle P \rangle\!\rangle = \langle\!\langle \text{safe}(P) \rangle\!\rangle$.

Proof. We present the proof for the most relevant cases presented in Algorithm 2, that is, (a) transformation in line 9 and (b) rewriting of filters in lines 17-20.

(a) Let $P = (P_1 \,\text{OPT}(P_2 \,\text{FILTER}\, C))$. Here $T(P) = \text{LeftJoin}(T(P_1), T(P_2), C)$ and $\langle\!\langle T(P) \rangle\!\rangle = \langle\!\langle T(P_1) \rangle\!\rangle ⟕_C \langle\!\langle T(P_2) \rangle\!\rangle$.
Suppose that $\Omega_1 = \langle\!\langle T(P_1) \rangle\!\rangle$ and $\Omega_2 = \langle\!\langle T(P_2) \rangle\!\rangle$. Then $\langle\!\langle T(P) \rangle\!\rangle$ is given by the expression $(\Omega_1 \bowtie_C \Omega_2) \cup (\Omega_1 \setminus_C \Omega_2)$ where:

$$^{(\star)}(\Omega_1 \bowtie_C \Omega_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2, \text{ and } (\mu_1 \cup \mu_2) \models C\}$$

$$^{(\star\star)}(\Omega_1 \setminus_C \Omega_2) = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\} \cup \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}$$

(i) Let $P' = (P_1 \,\text{OPT}((P_1 \,\text{AND}\, P_2) \,\text{FILTER}\, C))$. We will prove that, *under set semantics*, $\langle\!\langle P \rangle\!\rangle_G^D = \langle\!\langle P' \rangle\!\rangle_G^D$ for every dataset D with active graph G.

Consider that $P_3 = (P_1 \,\text{AND}\, P_2)$. Then $T(P')$ returns the algebra expression $\text{LeftJoin}(T(P_1), T(P_3), C)$ and $\langle\langle T(P')\rangle\rangle = \langle\langle T(P_1)\rangle\rangle ⟕_C \langle\langle T(P_3)\rangle\rangle$. Suppose that $\Omega_1 = \langle\langle T(P_1)\rangle\rangle$, $\Omega_2 = \langle\langle T(P_2)\rangle\rangle$ and $\Omega_3 = \langle\langle T(P_3)\rangle\rangle$. Then $\langle\langle T(P')\rangle\rangle$ is given by the expression $(\Omega_1 \bowtie_C \Omega_3) \cup (\Omega_1 \setminus_C \Omega_3)$ where:

$^{(1)}(\Omega_1 \bowtie_C \Omega_3) =$
$\{\mu_1 \cup \mu_3 \mid \mu_1 \in \Omega_1, \mu_3 \in \Omega_3, \mu_1 \sim \mu_3 \text{ and } (\mu_1 \cup \mu_3) \models C\}$

and

$^{(2)}(\Omega_1 \setminus_C \Omega_3) =$
$\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_3 \in \Omega_3, \mu_1 \text{ and } \mu_3 \text{ are not compatible}\} \cup$
$\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_3 \in \Omega_3 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_3) \not\models C\}$.

Assume $\Omega_3 = \Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}$. If we rewrite (1) by solving μ_3, we will have the set

$^{(1.1)}\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \text{ and } (\mu_1 \cup \mu_2) \models C\}$.

In the former set of (2): by definition of Ω_3, it applies that μ_1 is not compatible with every mapping $(\mu'_1 \cup \mu_2) \in \Omega_3$ such that $\mu'_1 \in \Omega_1$, $\mu_2 \in \Omega_2$ and $\mu'_1 \sim \mu_2$. This condition is true if and only if $\mu'_1 \neq \mu_1$. Consequently μ_1 is not compatible with every $\mu_2 \in \Omega_2$. Then, we can simplify the former set in (2) as:

$^{(2.1)}\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\}$

In the latter set of (2): by definition of Ω_3, we have that each mapping $(\mu'_1 \cup \mu_2) \in \Omega_3$ such that $\mu'_1 \in \Omega_1$, $\mu_2 \in \Omega_2$, $\mu'_1 \sim \mu_2$ and $\mu_1 \sim (\mu'_1 \cup \mu_2)$, it satisfies that $(\mu_1 \cup (\mu'_1 \cup \mu_2)) \not\models C$. The condition $\mu_1 \sim (\mu'_1 \cup \mu_2)$ is true if and only if $\mu'_1 = \mu_1$. Consequently μ_1 is compatible with some $\mu_2 \in \Omega_2$ and $(\mu_1 \cup \mu_2) \not\models C$. Then, we can simplify the latter set in (2) to the set:

$^{(2.2)}\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}$

Finally, we have that (1.1) corresponds to $(\star)$, (2.1) is the former set in $(\star\star)$ and (2.2) is the latter set in $(\star\star)$.
Then, we have proved that $\langle\langle P\rangle\rangle = \langle\langle P'\rangle\rangle$.

(ii) Let P' be the graph pattern

$($ $((P_1 \,\text{AND}\, P_2)\,\text{FILTER}\, C)\,\text{UNION}$
$(P_1 \,\text{MINUS}\, P_2)\,\text{UNION}$
$((P_1 \,\text{MINUS}(P_1 \,\text{MINUS}\, P_2))$
$\text{MINUS}((P_1 \,\text{AND}\, P_2)\,\text{FILTER}\, C))$ $)$

We will prove that, *under bag semantics*, $\langle\langle P\rangle\rangle^D_G = \langle\langle P'\rangle\rangle^D_G$ for every dataset D with active graph G.

Consider that $P_3 = ((P_1 \,\text{AND}\, P_2)\,\text{FILTER}\, C)$, $P_4 = (P_1 \,\text{MINUS}\, P_2)$ and $P_5 = ((P_1 \,\text{MINUS}(P_1 \,\text{MINUS}\, P_2))\,\text{MINUS}((P_1 \,\text{AND}\, P_2)\,\text{FILTER}\, C))$. We have that $T(P') = \text{Union}(\text{Union}(T(P_3), T(P_4)), T(P_5))$ where

$T(P_3) = \text{Filter}(C, \text{Join}(T(P_1), T(P_2)))$,
$T(P_4) = \text{Diff}(T(P_1), T(P_2), \mathit{true})$, and
$T(P_5) = \text{Diff}(\text{Diff}(T(P_1), T(P_4), \mathit{true}), T(P_3), \mathit{true})$

Suppose that $\Omega_1 = \langle\langle T(P_1)\rangle\rangle$ and $\Omega_2 = \langle\langle T(P_2)\rangle\rangle$. Then $\langle\langle T(P')\rangle\rangle$ is given by the expression $\langle\langle T(P_3)\rangle\rangle \cup \langle\langle T(P_4)\rangle\rangle \cup \langle\langle T(P_5)\rangle\rangle$ where

$$\langle\langle T(P_3)\rangle\rangle = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2, \text{ and } (\mu_1 \cup \mu_2) \models C\}$$

$$\langle\langle T(P_4)\rangle\rangle = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\} \cup \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models \mathit{true}\}$$

$$\langle\langle T(P_5)\rangle\rangle = (\langle\langle P_1\rangle\rangle \setminus_{true} \langle\langle T(P_4)\rangle\rangle) \setminus_{true} \langle\langle T(P_3)\rangle\rangle = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}$$

From the above sets we can state that:

- $\langle\langle T(P_3)\rangle\rangle$ correspond to the set in $(\star)$;
- $\langle\langle T(P_4)\rangle\rangle$ correspond to the former set in $(\star\star)$. Note that the second set will always be empty because condition $(\mu_1 \cup \mu_2) \not\models \mathit{true}$ is false in any case.
- The expression $(\langle\langle P_1\rangle\rangle \setminus_{true} \langle\langle T(P_4)\rangle\rangle)$ returns the subset of mappings in $\langle\langle P_1\rangle\rangle$ which are compatible with some mapping in $\langle\langle P_2\rangle\rangle$; from this set we subtract mappings from $\langle\langle P_3\rangle\rangle$ (i.e. such mappings that satisfies condition C); Then $\langle\langle T(P_5)\rangle\rangle$ returns mappings in $\langle\langle P_1\rangle\rangle$ that are compatible with some mapping in $\langle\langle P_2\rangle\rangle$ but not satisfying condition C, that is $\langle\langle T(P_5)\rangle\rangle$ corresponds to the latter set in $(\star\star)$.

Then we have proved that $\langle\langle P\rangle\rangle = \langle\langle P'\rangle\rangle$.

(b) Consider the following semantics defined in the $\text{SPARQL}_{\text{WG}}$ specification [9]:

- Apart from bound($\cdot$), all functions and operators operate on RDF Terms and will produce a type *error* if any arguments are unbound (Sec. 11.2).
- Function bound(var) returns true if var is bound to a value, and returns false otherwise (Sec. 11.4.1).

Let P be the non-safe graph pattern $(P_1 \,\text{FILTER}\, C)$, $?X$ be a variable in $\text{var}(C) \setminus \text{var}(P_1)$ and μ be a mapping in $\langle\langle P_1\rangle\rangle$. The evaluation $\mu(C')$ of an atomic filter constraint C' in C which contains variable $?X$, will be given (according to the above semantics) as follows:

(i) if C' is $(?X = u)$ or $(?X = ?Y)$ or isIRI($?X$) or isBlank($?X$) or isLiteral($?X$) then $\mu(C') = \mathit{error}$;

(ii) else if C' is bound($?X$) then $\mu(C') = \mathit{false}$.

To attain the same results, we can replace C' in C by either

- the filter expression bound(a) with $a \in I \cup L$ in case (i); or
- the filter expression *false* in case (ii).

Applying the above procedure to each atomic filter condition in C having a variable in $\text{var}(C) \setminus \text{var}(P_1)$, we will transform P in a safe filter pattern.

Thus we proved:

Theorem 2. *$SPARQL_{WG}$ and $SPARQL_{WG}^{Safe}$ have the same expressive power.*

5 Expressive power of $\text{SPARQL}_{\text{WG}}$ is equivalent to SPARQL_{C}

As we have been showing, the semantics that the WG gave to SPARQL departed in some aspects from a compositional semantics. We also indicated that there is an alternative formalization, with a standard compositional semantics, which was called SPARQL_{C} [6].

The good news is that, albeit apparent differences, these languages are equivalent in expressive power, that is, they compute the same class of queries.

Theorem 3. *$SPARQL^{Safe}_{WG}$ is equivalent to $SPARQL_{C}$ under bag semantics.*

Proof. The proof of this theorem is an induction on the structure of patterns. The only non-evident case is the particular evaluation of filters inside optionals where the semantics of $\text{SPARQL}^{\text{Safe}}_{\text{WG}}$ and SPARQL_{C} differ. Specifically, given a graph pattern $P = (P_1 \, \text{OPT}(P_2 \, \text{FILTER} \, C))$, we have that $\text{SPARQL}^{\text{Safe}}_{\text{WG}}$ evaluates the algebra expression $T(P) = \text{LeftJoin}(T(P_1), T(P_2), C)$, whereas SPARQL_{C} evaluates P to the expression $[\![P_1]\!] ⟕ [\![(P_2 \, \text{FILTER} \, C)]\!]$, which is the same as the $\text{SPARQL}_{\text{WG}}$ algebra expression $\text{LeftJoin}(T(P_1), \text{Filter}(C, T(P_2)), \mathit{true})$. Note that the scope of filter condition C in $\text{SPARQL}_{\text{WG}}$ is the entire pattern P, whereas in SPARQL_{C} the scope of C is the graph pattern P_2.

Let P be the graph pattern $(P_1 \, \text{OPT}(P_2 \, \text{FILTER} \, C))$ where $\text{var}(C) \subseteq \text{var}(P_2)$ (i.e., P is filter safe). We will show that for every dataset D with active graph G, it satisfies that $\langle\!\langle P \rangle\!\rangle^D_G = [\![P]\!]^D_G$.

- *Evaluation* $\langle\!\langle P \rangle\!\rangle^D_G$: Following the steps of evaluation in $\text{SPARQL}_{\text{WG}}$, we have that $T(P) = \text{LeftJoin}(T(P_1), T(P_2), C)$ and $\langle\!\langle T(P) \rangle\!\rangle = \langle\!\langle T(P_1) \rangle\!\rangle ⟕_C \langle\!\langle T(P_2) \rangle\!\rangle$. Suppose that $\Omega_1 = \langle\!\langle T(P_1) \rangle\!\rangle$ and $\Omega_2 = \langle\!\langle T(P_2) \rangle\!\rangle$. Then $\langle\!\langle T(P) \rangle\!\rangle$ is given by the expression $(\Omega_1 \bowtie_C \Omega_2) \cup (\Omega_1 \setminus_C \Omega_2)$ where:

  $^{(\star)}(\Omega_1 \bowtie_C \Omega_2) =$
  $$\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \text{ and } (\mu_1 \cup \mu_2) \models C\}$$
  and

  $^{(\star\star)}(\Omega_1 \setminus_C \Omega_2) =$
  $$\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\} \cup$$
  $$\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}.$$
- *Evaluation* $[\![P]\!]^D_G$: We have that $[\![P]\!] = [\![P_1]\!] ⟕ [\![(P_2 \, \text{FILTER} \, C)]\!]$. Suppose that $\Omega_1 = [\![P_1]\!]$, $\Omega_2 = [\![P_2]\!]$ and $\Omega_3 = [\![(P_2 \, \text{FILTER} \, C)]\!]$. Then $[\![P]\!]$ is given by the expression $(\Omega_1 \bowtie \Omega_3) \cup (\Omega_1 \setminus \Omega_3)$ where

  $^{(1)}(\Omega_1 \bowtie \Omega_3) = \{\mu_1 \cup \mu_3 \mid \mu_1 \in \Omega_1, \mu_3 \in \Omega_3 \text{ and } \mu_1 \sim \mu_3\}$

  and

  $^{(2)}(\Omega_1 \setminus \Omega_3) = \{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_3 \in \Omega_3, \mu_1 \text{ and } \mu_3 \text{ are not compatible}\}$.

  Considering that $\Omega_3 = \{\mu_2 \in \Omega_2 \mid \mu_2 \models C\}$. If we redefine (1) by solving $\mu_3 \in \Omega_3$, we will have the set

  $^{(1.1)}\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \text{ and } (\mu_1 \cup \mu_2) \models C\}$.

Additionally, consider to change the universal quantifier in (2) by an existential one, That is $(\Omega_1 \setminus \Omega_3) = \{\mu_1 \in \Omega_1 \mid \nexists \mu_3 \in \Omega_3$ such that $\mu_1 \sim \mu_3\}$. Here we have two cases:

- When $\Omega_3 = \emptyset$. In this case, there exists no mapping $\mu_2 \in \Omega_2$ satisfying that $\mu_2 \models C$. Then this case encodes the set

$$^{(2.1)}\{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}.$$

- When $\Omega_3 \neq \emptyset$. In this case, for each mapping $\mu_2 \in \Omega_2$ satisfying that $\mu_2 \models C$, it applies that μ_1 and μ_2 are not compatible. Then this case encodes the set

$$^{(2.2)}\{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2 \text{ such that } \mu_2 \models C, \; \mu_1 \text{ and } \mu_2 \text{ are not compatible}\}$$

Note that (1.1) corresponds to ($\star$), (2.1) corresponds to the latter set in ($\star\star$), and (2.2) corresponds to the former set in ($\star\star$). Then we have proved that $\langle\langle P \rangle\rangle^D_G = [\![P]\!]^D_G$.

6 Expressive Power of SPARQL_C

In this section we study the expressive power of SPARQL_C by comparing it against non recursive safe Datalog with negation (just Datalog from now on).

Note that because SPARQL_C and Datalog programs have different type of input and output formats, we have to normalize them to be able to do the comparison. Following definitions in section 2.4, let $L_s = (\mathcal{Q}_s, \mathcal{D}_s, \mathcal{S}_s, \text{ans}_s)$ be the SPARQL_C language, and $L_d = (\mathcal{Q}_d, \mathcal{D}_d, \mathcal{S}_d, \text{ans}_d)$ be the Datalog language.

In this comparison we restrict the notion of *SPARQL_C Query* to a pair (P, D) where P is a graph pattern and D is an RDF dataset.

6.1 From SPARQL_C to Datalog

To prove that the SPARQL_C language $L_s = (\mathcal{Q}_s, \mathcal{D}_s, \mathcal{S}_s, \text{ans}_s)$ is contained in the Datalog language $L_d = (\mathcal{Q}_d, \mathcal{D}_d, \mathcal{S}_d, \text{ans}_d)$, we define transformations $\mathcal{T}_Q : \mathcal{Q}_s \to \mathcal{Q}_d$, $\mathcal{T}_D : \mathcal{D}_s \to \mathcal{D}_d$, and $\mathcal{T}_S : \mathcal{S}_s \to \mathcal{S}_d$. That is, $\mathcal{T}_Q$ transforms a SPARQL_C query into a Datalog query, $\mathcal{T}_D$ transforms an RDF dataset into a set of Datalog facts, and $\mathcal{T}_S$ transforms a set of SPARQL_C mappings into a set of Datalog substitutions.

RDF datasets as Datalog facts.
Given an RDF dataset $D = \{G_0, \langle u_1, G_1 \rangle, \dots, \langle u_n, G_n \rangle\}$, the transformation $\mathcal{T}_D(D)$ works as follows: each term t in D is encoded by a fact $iri(t)$, $blank(t)$ or $literal(t)$ when t is an IRI, a blank node or a literal respectively; the set of terms in D is defined by the set of rules $term(X) \leftarrow iri(X)$, $term(X) \leftarrow blank(X)$, and $term(X) \leftarrow literal(X)$; the fact $Null(null)$ encodes the *null* value [7]; each triple (v_1, v_2, v_3) in the default graph G_0 is encoded by a fact $triple(g_0, v_1, v_2, v_3)$; each named graph $\langle u_i, G_i \rangle$ is encoded by a fact $graph(u)$ and each triple (v_1, v_2, v_3) in G_i is encoded by a fact $triple(u_i, v_1, v_2, v_3)$.

[7] We use the term null to represent an unbounded value.

Table 3. Transforming SPARQL_C graph patterns into Datalog Rules. D is a dataset having active graph identified by g. $\overline{\text{var}}(P)$ denotes the tuple of variables obtained from a lexicographical ordering of the variables in the graph pattern P. Each p_i is a predicate identifying the graph pattern P_i. If L is a literal, then $\nu_j(L)$ denotes a copy of L with its variables renamed according to a variable renaming function $\nu_j : V \to V$. *cond* is a literal encoding the filter condition C. Each P_{1i} is a copy of P_1 and $u_i \in \text{names}(D)$. $P_3 = (P_1 \text{ AND } P_2)$, $P_4 = (P_1 \text{ FILTER } C_1)$ and $P_5 = (P_1 \text{ FILTER } C_2)$.

Pattern P	$\delta(P, g)_D$
(x_1, x_2, x_3)	$p(\overline{\text{var}}(P)) \leftarrow triple(g, x_1, x_2, x_3)$
$(P_1 \text{ AND } P_2)$	$p(\overline{\text{var}}(P)) \leftarrow \nu_1(p_1(\overline{\text{var}}(P_1))) \ \wedge \ \nu_2(p_2(\overline{\text{var}}(P_2))) \ \bigwedge_{x \in \text{var}(P_1) \cap \text{var}(P_2)} comp(\nu_1(x), \nu_2(x), x)$, $\delta(P_1, g)_D$, $\delta(P_2, g)_D$
	$\text{dom}(\nu_1) = \text{dom}(\nu_2) = \text{var}(P_1) \cap \text{var}(P_2)$, $\text{range}(\nu_1) \cap \text{range}(\nu_2) = \emptyset$.
$(P_1 \text{ UNION } P_2)$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x)$, $p(\overline{\text{var}}(P)) \leftarrow p_2(\overline{\text{var}}(P_2)) \bigwedge_{x \in \text{var}(P_1) \wedge x \notin \text{var}(P_2)} Null(x)$, $\delta(P_1, g)_D$, $\delta(P_2, g)_D$
$(P_1 \text{ OPT } P_2)$	$p(\overline{\text{var}}(P)) \leftarrow p_3(\overline{\text{var}}(P_3))$, $p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p_1'(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x)$, $p_1'(\overline{\text{var}}(P_1)) \leftarrow p_3(\overline{\text{var}}(P_3))$, $\delta(P_1, g)_D$, $\delta(P_2, g)_D$, $\delta(P_3, g)_D$
$(u \text{ GRAPH } P_1)$ and $u \in I$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1))$, $\delta(P_1, u)_D$
$(?X \text{ GRAPH } P_1)$ and $?X \in V$	$p(\overline{\text{var}}(P)) \leftarrow p_{11}(\overline{\text{var}}(P_{11})) \wedge graph(?X) \wedge ?X = u_1$, $\delta(P_{11}, u_1)_D$, $\ldots$ $p(\overline{\text{var}}(P)) \leftarrow p_{1n}(\overline{\text{var}}(P_{1n})) \wedge graph(?X) \wedge ?X = u_n$, $\delta(P_{1n}, u_n)_D$
$(P_1 \text{ FILTER } C)$ C is atomic	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge cond$ $\delta(P_1, g)_D$
$(P_1 \text{ FILTER } C)$ C is $(\neg(C_1))$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p_4(\overline{\text{var}}(P_1))$, $\delta(P_1, g)_D$, $\delta(P_4, g)_D$
$(P_1 \text{ FILTER } C)$ C is $(C_1 \wedge C_2)$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1))$, $p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1))$, $p''(\overline{\text{var}}(P_1)) \leftarrow p_4(\overline{\text{var}}(P_1)) \wedge p_5(\overline{\text{var}}(P_1))$, $\delta(P_1, g)_D$, $\delta(P_4, g)_D$, $\delta(P_5, g)_D$
$(P_1 \text{ FILTER } C)$ C is $(C_1 \vee C_2)$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1))$, $p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1))$ $p''(\overline{\text{var}}(P_1)) \leftarrow p_4(\overline{\text{var}}(P_1))$, $p''(\overline{\text{var}}(P_1)) \leftarrow p_5(\overline{\text{var}}(P_1))$, $\delta(P_1, g)_D$, $\delta(P_4, g)_D$, $\delta(P_5, g)_D$

$SPARQL_C$ mappings as Datalog substitutions.
Given a graph pattern P, an RDF dataset D with default graph G, and the set of mappings $\Omega = [\![P]\!]_G^D$. The transformation $\mathcal{T}_S(\Omega)$ returns a set of substitutions defined as follows: for each mapping $\mu \in \Omega$ there exists a substitution θ in $\mathcal{T}_S(\Omega)$ satisfying that, for each $x \in \mathrm{var}(P)$ there exists $x/t \in \theta$ such that $t = \mu(x)$ when $\mu(x)$ is bounded and $t = null$ otherwise.

Graph patterns as Datalog rules.
Let P be a graph pattern to be evaluated against an RDF graph identified by g which occurs in dataset D. We denote by $\delta(P, g)_D$ the function which transforms P into a set of Datalog rules. Table 3 shows the transformation rules defined by the function $\delta(P, g)_D$, where:

- The notion of compatible mappings is implemented by the rules:
  $comp(X, X, X) \leftarrow term(X)$,
  $comp(X, null, X) \leftarrow term(X)$
  $comp(null, X, X) \leftarrow term(X)$ and
  $comp(X, X, X) \leftarrow Null(X)$.
- Let $?X, ?Y \in V$ and $u \in I \cup L$. An atomic filter condition C is encoded by a literal L as follows:
  - if C is either $(?X = u)$ or $(?X = ?Y)$ then L is C;
  - if C is isIRI($?X$) then L is $iri(?X)$;
  - if C is isLiteral($?X$) then L is $literal(?X)$;
  - if C is isBlank($?X$) then L is $blank(?X)$;
  - if C is bound($?X$) then L is $\neg Null(?X)$.

The transformation follows essentially the intuitive transformation presented by Polleres [8] with the improvement of the necessary code to support faithful translation of bag semantics. Specifically, we changed the transformations for complex filter expressions by simulating them with double negation.

$SPARQL_C$ queries as Datalog queries.
Given a $\mathrm{SPARQL_C}$ query $Q = (P, D)$ where P is a graph pattern and D is an RDF dataset. The function $\mathcal{T}_Q(Q)$ returns the Datalog query $(\Pi, p(\overline{\mathrm{var}}(P)))$ where Π is the Datalog program $\mathcal{T}_D(D) \cup \delta(P, g_0)_D$, the identifier g_0 references the default graph of D, and p is the goal literal related to P.

The following theorem states that the above transformations work well.

Theorem 4. *$SPARQL_C$ is contained in non-recursive safe Datalog with negation.*

Proof. We need to prove that for every $\mathrm{SPARQL_C}$ query $Q = (P, D)$ it satisfies that $\mathcal{T}_S(\mathrm{ans}_s(Q, D)) = \mathrm{ans}_d(\mathcal{T}_Q(Q), \mathcal{T}_D(D))$ where $\mathrm{ans}_s(Q, D)$ denotes the evaluation function $[\![P]\!]^D_{\mathrm{dg}(D)}$. Considering that $\mathcal{T}_Q(Q)$ is the Datalog query $(\Pi, p(\overline{\mathrm{var}}(P)))$ where Π is the Datalog program $\mathcal{T}_D(D) \cup \delta(P, g_0)_D$. We need to show that for each mapping $\mu \in [\![P]\!]^D_{\mathrm{dg}(G)}$ there exists substitution θ such that $\theta(p(\overline{\mathrm{var}}(P))) \in \mathrm{facts}^*(\Pi)$ and $\theta = \mathcal{T}_S(\mu)$. The proof is by induction on the structure of P.

(1) Base case: P is a triple pattern (x_1, x_2, x_3).
I this case $\delta(P, g)$ returns the rule $p(\overline{\text{var}}(P)) \leftarrow triple(g, x_1, x_2, x_3)$.
Given a substitution θ, it satisfies that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\Pi)$ iff there is a substitution $\theta = \{x_i/v_i \mid x_i \in \text{var}(P)\}$ such that $\theta(triple(g, x_1, x_2, x_3)) \in \mathcal{T}_D(D)$.
On the other hand, a mapping μ is in $[\![P]\!]^D_G$ if and only if $\text{dom}(\mu) = \text{var}(P)$ and $\mu((x_1, x_2, x_3)) = (v_1, v_2, v_3) \in G$. Then $\mu(x_i) = v_i$ when $x_i \in \text{var}(P)$. If we transform μ into a substitution, that is $\mathcal{T}_S(\mu) = \{x_i/v_i \mid x_i \in \text{var}(P)\}$. Then $\theta = \mathcal{T}_S(\mu)$ and we are done.

Inductive case: Let P_1 and P_2 be graph patterns. We consider several cases:

(2) P is $(P_1 \text{ AND } P_2)$.
In this case $\delta(P, g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow \nu_1(p_1(\overline{\text{var}}(P_1))) \wedge \nu_2(p_2(\overline{\text{var}}(P_2))) \bigwedge_{x \in \text{var}(P_1) \cap \text{var}(P_2)} comp(\nu_1(x), \nu_2(x), x),$$
$$\delta(P_1, g),\ \delta(P_2, g)\ \}$$

where $\text{dom}(\nu_1) = \text{dom}(\nu_2) = \text{var}(P_1) \cap \text{var}(P_2)$ and $\text{range}(\nu_1) \cap \text{range}(\nu_2) = \emptyset$. Note that we use functions ν_1 and ν_2 to rename common variables between patterns P_1 and P_2, and we use the renamed variables to simulate the notion of compatible mappings through the predicate *comp*.
Given a substitution θ, it satisfies that a fact $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\Pi)$ iff $\theta(\nu_1(p_1(\overline{\text{var}}(P_1)))) \in \text{facts}^*(\Pi)$, $\theta(\nu_2(p_2(\overline{\text{var}}(P_2)))) \in \text{facts}^*(\Pi)$, and for each variable $x_i \in \text{var}(P_1) \cap \text{var}(P_2)$, $\theta(comp(\nu_1(x_i), \nu_2(x_i), x_i)) \in \text{facts}^*(\Pi)$ i.e., $\theta(x_i) = \theta(\nu_1(x_i)) = \theta(\nu_2(x_i))$, or $\theta(\nu_1(x_i)) = null$ and $\theta(x_i) = \theta(\nu_2(x_i))$, or $\theta(\nu_2(x_i)) = null$ and $\theta(x_i) = \theta(\nu_1(x_i))$.
On the other hand, a mapping μ is in $[\![(P_1 \text{ AND } P_2)]\!]^D_G$ iff $\mu = \mu_1 \cup \mu_2$ such that $\mu_1 \in [\![P_1]\!]^D_G$, $\mu_2 \in [\![P_2]\!]^D_G$, and μ_1 is compatible with μ_2 i.e, for each $x \in \text{var}(P_1) \cap \text{var}(P_2)$ it applies that $\mu_1(x) = \mu_2(x)$ or $\mu_1(x)$ is unbounded or $\mu_2(x)$ is unbounded.
For induction hypothesis, we have substitutions $\theta_1 = \mathcal{T}_S(\mu_1)$, $\theta_2 = \mathcal{T}_S(\mu_2)$ such that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$, $\theta_2(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi))$, and for each $x \in \text{var}(P_1) \cap \text{var}(P_2)$ we have that $\theta_1(x) = \theta_2(x)$, or $\theta_1(x)$ is null, or $\theta_2(x)$ is null. Considering that $\mathcal{T}_S(\mu) = \theta_1 \cup \theta_2$ we have that $\theta = \mathcal{T}_S(\mu)$ and we are done.

(3) If P is $(P_1 \text{ UNION } P_2)$.
In this case $\delta(P, g)$ returns the set of rules

$$\{p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x),$$
$$p(\overline{\text{var}}(P)) \leftarrow p_2(\overline{\text{var}}(P_2)) \bigwedge_{x \in \text{var}(P_1) \wedge x \notin \text{var}(P_2)} Null(x),$$
$$\delta(P_1, g),\ \delta(P_2, g)\ \}$$

Given a substitution θ, it satisfies that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\Pi)$ iff either

(a) $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and x is null for each $x \in \text{var}(P) \setminus \text{var}(P_1)$, i.e, $\theta = \{x/v \mid x \in \text{var}(P_1)\} \cup \{x/null \mid x \in \text{var}(P) \setminus \text{var}(P_1)\}$; or
(b) $\theta(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi)$ and x is null for each $x \in \text{var}(P) \setminus \text{var}(P_2)$, i.e, $\theta = \{x/v \mid x \in \text{var}(P_2)\} \cup \{x/null \mid x \in \text{var}(P) \setminus \text{var}(P_2)\}$.

On the other hand, a mapping μ is in $[\![(P_1 \,\text{UNION}\, P_2)]\!]^D_G$ if and only if either (a) $\mu = \mu_1 \in [\![P_1]\!]^D_G$ or (b) $\mu = \mu_2 \in [\![P_2]\!]^D_G$. For induction hypothesis, we have that there exist substitutions $\theta_1 = \mathcal{T}_S(\mu_1)$, $\theta_2 = \mathcal{T}_S(\mu_2)$ satisfying that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and $\theta_2(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi)$. Assuming that $\theta_1 = \{x/v \mid x \in \text{var}(P_1)\}$ and $\theta_2 = \{x/v \mid x \in \text{var}(P_2)\}$. It applies that in case (a), $\mathcal{T}_S(\mu) = \theta_1 \cup \{x/null \mid x \in \text{var}(P) \setminus \text{var}(P_1)\}$; and in case (b), $\mathcal{T}_S(\mu) = \theta_2 \cup \{x/null \mid x \in \text{var}(P) \setminus \text{var}(P_2)\}$. Then, we have that $\theta = \mathcal{T}_S(\mu)$ and we are done.

(4) P is $(P_1 \,\text{OPT}\, P_2)$.
In this case $\delta(P, g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_3(\overline{\text{var}}(P_3)),$$
$$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p_1'(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x),$$
$$p_1'(\overline{\text{var}}(P_1)) \leftarrow p_3(\overline{\text{var}}(P_3)),$$
$$\delta(P_1, g),\ \delta(P_2, g),\ \delta(P_3, g)\ \},$$

where $P_3 = (P_1 \,\text{AND}\, P_2)$.
Given a substitution θ, we have that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\Pi)$ iff either

(i) $\theta(p_3(\overline{\text{var}}(P_3))) \in \text{facts}^*(\Pi)$; or

(ii) $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and is false that $\theta(p_1'(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$; that is, if $\theta = \theta_1$ such that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$, then for all $\theta_2(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi)$ it is false that $comp(\theta_1(x), \theta_2(x), \theta(x))$, i.e., it applies that $\theta_1(x) \neq \theta_2(x)$ for each variable $x \in \text{var}(P_1) \cap \text{var}(P_2)$. In this case, $\theta(x)$ is null for each variable $x \in \text{var}(P) \setminus \text{var}(P_1)$.

On the other hand, a mapping μ is in $[\![(P_1 \,\text{OPT}\, P_2)]\!]^D_G$ iff either:

(a) $\mu \in [\![P_3]\!]^D_G$ where $P_3 = (P_1 \,\text{AND}\, P_2)$; or

(b) $\mu = \mu_1 \in [\![P_1]\!]^D_G$ such that for all $\mu_2 \in [\![P_2]\!]^D_G$ it satisfies that μ_1 and μ_2 are not compatible. Here $\mu(x)$ is unbounded for each $x \in \text{var}(P) \setminus \text{var}(P_1)$.

For induction hypothesis, we have substitutions $\theta_1 = \mathcal{T}_S(\mu_1)$ and $\theta_2 = \mathcal{T}_S(\mu_2)$ satisfying that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and $\theta_2(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi)$. Suppose that $\theta' = \mathcal{T}_S(\mu)$. Following definition of μ, we have that:

- In case (a), $\theta'(p_3(\overline{\text{var}}(P_3))) \in \text{facts}^*(\Pi)$ (as was showed in (2)).
- In case (b), $\theta' = \theta_1$ and θ_1 is not compatible with every θ_2, that is $\theta_1(x) \neq \theta_2(x)$ for each variable $x \in \text{var}(P_1) \cap \text{var}(P_2)$. Additionally, $x/null \in \theta'$ for each $x \in \text{var}(P) \setminus \text{var}(P_1)$.

Considering that (a) corresponds to (i), and (b) corresponds to (ii), then $\theta = \theta' = \mathcal{T}_S(\mu)$ and we are done.

(5) P is $(u \,\text{GRAPH}\, P_1)$ where $u \in I$.
In this case $\delta(P, g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)),$$
$$\delta(P_1, u)\ \}$$

Given a substitution θ, we have that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\mathcal{T}_D(D) \cup \delta(P, g))$ if and only if $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\mathcal{T}_D(D) \cup \delta(P_1, u))$. On the other hand, a mapping μ is in $[\![P]\!]^D_G$ if and only if $\mu \in [\![P_1]\!]^D_{G'}$ such that $G' = \text{gr}(u)_D$. In both cases, the active graph identified g has been changed by the graph identified u. Then by induction hypothesis we have that $\theta = \mathcal{T}_S(\mu)$.

(6) P is $(?X \, \mathrm{GRAPH} \, P_1)$ where $?X \in V$.
In this case, for each named graph identified u_i in dataset D, we have that $\delta(P, g)$ contains the following two rules:
$p(\overline{\mathrm{var}}(P)) \leftarrow p_{1i}(\overline{\mathrm{var}}(P_{1i})) \ \wedge \ \mathit{graph}(?X) \wedge ?X = u_i$, and
$\delta(P_{1i}, u_i)_D$.
Considering that P_{1i} is a copy of P_1 and using result (5), we can prove that $p(\overline{\mathrm{var}}(P)) \leftarrow p_{1i}(\overline{\mathrm{var}}(P_{1i}))$ is correct for each named graph u_i in dataset D. Additionally, given that $\mathrm{var}(P)$ is $?X \cup \mathrm{var}(P_{1i})$, we use the literals $\mathit{graph}(?X)$ and $?X = u_i$ to assign the respective IRI u_i to variable $?X$, then we are changing the active graph to the graph identified by u_i. As result, a substitution θ is in $\delta(P, g)$ iff θ is a substitution for a some $\delta(P_{1i}, u_i)$ where u_i identifies a graph in D. Then we have proved the case.

(7) If P is $(P_1 \, \mathrm{FILTER} \, C)$ and C is an atomic filter constraint.
In this case $\delta(P, g)$ returns the set of rules
$\{ \ p(\overline{\mathrm{var}}(P)) \leftarrow p_1(\overline{\mathrm{var}}(P_1)) \wedge \mathit{cond}$,
$\delta(P_1, g) \ \}$,
where cond is a Datalog literal encoding the filter condition C.
Given a substitution θ, we have that $\theta(p(\overline{\mathrm{var}}(P))) \in \mathrm{facts}^*(\mathit{\Pi})$ if and only if $\theta(p_1(\overline{\mathrm{var}}(P_1))) \in \mathrm{facts}^*(\mathit{\Pi})$ and $\theta(\mathit{cond})$ is true.
On the other hand, a mapping μ is in $[\![P]\!]_G^D$ iff $\mu \in [\![P_1]\!]_G^D$ and μ satisfies C. By induction hypothesis and considering that cond is a Datalog literal equivalent to filter constraint C, it applies that there exists substitution $\theta = \mathcal{T}_S(\mu)$ satisfying that $\theta(p_1(\overline{\mathrm{var}}(P_1))) \in \mathrm{facts}^*(\mathit{\Pi})$ and $\theta(\mathit{cond})$ is true.

(8) If P is $(P_1 \, \mathrm{FILTER} \, C)$ and C is $(\neg(C_1))$.
In this case $\delta(P, g)$ returns the set of rules
$\{ \ p(\overline{\mathrm{var}}(P)) \leftarrow p_1(\overline{\mathrm{var}}(P_1)) \wedge \neg p_4(\overline{\mathrm{var}}(P_1))$,
$\delta(P_1, g)$, $\delta(P_4, g) \ \}$,
where $P_4 = (P_1 \, \mathrm{FILTER} \, C_1)$.
Given a substitution θ, it satisfies that $\theta(p(\overline{\mathrm{var}}(P))) \in \mathrm{facts}^*(\mathit{\Pi})$ if and only if $\theta(p_1(\overline{\mathrm{var}}(P_1))) \in \mathrm{facts}^*(\mathit{\Pi})$ and is false that $\theta(p_4(\overline{\mathrm{var}}(P_1))) \in \mathrm{facts}^*(\mathit{\Pi})$. The last condition implies that, if cond_1 is the Datalog literal encoding C_1 then, $\theta(\mathit{cond}_1)$ is not true.
On the other hand, we have that a mapping μ is in $[\![P]\!]_G^D$ if and only if $\mu \in [\![P_1]\!]_G^D$ and it is not true that $\mu \models C_1$.
By induction hypothesis and considering that cond_1 is the Datalog literal equivalent to C_1, we have that there exists substitution $\theta = \mathcal{T}_S(\mu)$ satisfying that $\theta(p_1(\overline{\mathrm{var}}(P_1))) \in \mathrm{facts}^*(\mathit{\Pi})$ and $\theta(\mathit{cond}_1)$ is not true.

(9) If P is $(P_1 \, \mathrm{FILTER} \, C)$ and C is $(C_1 \wedge C_2)$.
In this case $\delta(P, g)$ returns the set of rules
$\{ \ p(\overline{\mathrm{var}}(P)) \leftarrow p_1(\overline{\mathrm{var}}(P_1)) \wedge \neg p'(\overline{\mathrm{var}}(P_1))$,
$p'(\overline{\mathrm{var}}(P_1)) \leftarrow p_1(\overline{\mathrm{var}}(P_1)) \wedge \neg p''(\overline{\mathrm{var}}(P_1))$,
$p''(\overline{\mathrm{var}}(P_1)) \leftarrow p_4(\overline{\mathrm{var}}(P_1)) \wedge p_5(\overline{\mathrm{var}}(P_1))$,
$\delta(P_1, g)$, $\delta(P_4, g)$, $\delta(P_5, g) \ \}$
where $P_4 = (P_1 \, \mathrm{FILTER} \, C_1)$ and $P_5 = (P_1 \, \mathrm{FILTER} \, C_2)$.
Note that the graph pattern $(P_1 \, \mathrm{FILTER}(C_1 \wedge C_2))$ can be rewritten as

$((P_1\, \text{FILTER}\, C_1)\, \text{AND} (P_1\, \text{FILTER}\, C_2))$ (it is showed in the rule for predicate p'' and by the patterns P_4 and P_5). This transformation is true under set-semantics, but it fails when we consider bag-semantics because it duplicates the bag of solutions. To solve this problem, we consider a double negation of the filter condition, that is we rewrite C to $(\neg(\neg C))$ (as is showed by the rules for predicates p and p'). Given that negated literals does not increase solutions, we will have only solutions from predicate p_1. Then we have proved the case.

(10) If P is $(P_1\, \text{FILTER}\, C)$ and C is $(C_1 \vee C_2)$.
In this case $\delta(P, g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1)),$$
$$p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1)),$$
$$p''(\overline{\text{var}}(P_1)) \leftarrow p_4(\overline{\text{var}}(P_1)),$$
$$p''(\overline{\text{var}}(P_1)) \leftarrow p_5(\overline{\text{var}}(P_1)),$$
$$\delta(P_1, g),\ \delta(P_4, g),\ \delta(P_5, g)\ \}$$

where $P_4 = (P_1\, \text{FILTER}\, C_1)$ and $P_5 = (P_1\, \text{FILTER}\, C_2)$.
Note that the graph pattern $(P_1\, \text{FILTER}(C_1 \vee C_2))$ can be rewritten as $((P_1\, \text{FILTER}\, C_1)\, \text{UNION} (P_1\, \text{FILTER}\, C_2))$ (it is showed by the rules for predicate p'' and by the patterns P_4 and P_5). Similar to (9), we apply a double negation of the filter condition C (as is showed by the rules for predicates p and p') to solve the problem for bag-semantics. This proved the case.

Note 7. Given a graph pattern P, the transformation $\delta(P, g)$ preserves the bag semantics of the SPARQL WG specification. Consider the cardinality m of a solution s for P (and the equivalent solution for $\delta(P, g)$). It can be checked that: in case (1), the value of m is 1 because each triple occurs once in the active graph; in case (2), m is the product of the cardinalities for s in the bags of solutions for $\langle\langle P_1 \rangle\rangle$ and $\langle\langle P_2 \rangle\rangle$; in case (3), m is the sum of the cardinalities for s in the bags of solutions for $\langle\langle P_1 \rangle\rangle$ and $\langle\langle P_2 \rangle\rangle$; in case (4), m is given by either the product of cardinalities for s in the bags of solutions for $\langle\langle P_1 \rangle\rangle$ and $\langle\langle P_2 \rangle\rangle$, or the cardinalities for s in the bag of solutions for $\langle\langle P_1 \rangle\rangle$; in case (5), m is given by the cardinality of s in the bag of solutions for named graph u; in case (6), m is given by the sum of cardinalities for s in the bag of solutions for each named graph in the dataset; in cases (7),(8),(9), and (10), m is given by the cardinality of s in the bag of solutions for P_1.

6.2 From Datalog to SPARQL_C

To prove that the Datalog language $L_d = (\mathcal{Q}_d, \mathcal{D}_d, \mathcal{S}_d, \text{ans}_d)$ is contained in the SPARQL_C language $L_s = (\mathcal{Q}_s, \mathcal{D}_s, \mathcal{S}_s, \text{ans}_s)$, we define transformations $\mathcal{T}'_Q : \mathcal{Q}_d \to \mathcal{Q}_s$, $\mathcal{T}'_D : \mathcal{D}_d \to \mathcal{D}_s$, and $\mathcal{T}'_S : \mathcal{S}_d \to \mathcal{S}_s$. That is, $\mathcal{T}'_Q$ transforms a Datalog query into an SPARQL_C query, $\mathcal{T}'_D$ transforms a set of Datalog facts into an RDF dataset, and $\mathcal{T}'_S$ transforms a set of Datalog substitutions into a set of SPARQL_C mappings.

Datalog facts as an RDF Dataset
Given a Datalog fact $f = p(c_1, ..., c_n)$, consider function desc(f) which returns the set of triples { (_:b,predicate,p), (_:b,rdf:_1,c_1),...,(_:b,rdf:_n,c_n) }, where _:b is a fresh blank node. Given a set of Datalog facts F, we have that $\mathcal{T}'_D(F)$ returns an RDF dataset with default graph $G_0 = \{\text{desc}(f) \mid f \in F\}$, where $\text{blank}(\text{desc}(f_i)) \cap \text{blank}(\text{desc}(f_j)) = \emptyset$ for each $f_i, f_j \in F$ with $i \neq j$.

Datalog substitutions as SPARQL$_C$ mappings.
Given a set of substitutions Θ, the transformation $\mathcal{T}'_S(\Theta)$ returns a set of mappings defined as follows: for each substitution $\theta \in \Theta$ there exists a mapping $\mu \in \mathcal{T}'_S(\Theta)$ satisfying that, if $x/t \in \theta$ then $x \in \text{dom}(\mu)$ and $\mu(x) = t$.

Datalog rules as SPARQL$_C$ graph patterns
Let Π be a Datalog program, and L be a literal $p(x_1, \ldots, x_n)$ where p is a predicate in Π and each x_i is a variable. We define the function $\text{gp}(L)_\Pi$ which returns a graph pattern encoding the program (Π, L), that is, the fragment of the program Π used for evaluating literal L.

The translation works intuitively as follows:

(a) If predicate p is extensional, then $\text{gp}(L)_\Pi$ returns the graph pattern $((?Y, \text{predicate}, p) \,\text{AND}(?Y, \text{rdf:\_1}, x_1) \,\text{AND} \cdots \text{AND}(?Y, \text{rdf\_n}, x_n))$, where $?Y$ is a fresh variable.
(b) If predicate p is intensional, then for each rule in Π of the form

$$L \leftarrow L_1 \wedge \cdots \wedge L_s \wedge \neg L_{s+1} \wedge \cdots \wedge \neg L_t \wedge L_1^{eq} \wedge \cdots \wedge L_u^{eq},$$

where each L_i is a predicate formula and each L_k^{eq} is a literal either of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$, it applies that $\text{gp}(L)_\Pi$ returns a graph pattern with the structure

$$\begin{aligned}(((\cdots((\text{gp}(L_1)_\Pi \,\text{AND} \cdots \text{AND}\, \text{gp}(L_s)_\Pi)\\ \text{MINUS}\, \text{gp}(L_{s+1})_\Pi) \cdots)\, \text{MINUS}\, \text{gp}(L_t)_\Pi)\\ \text{FILTER}(L_1^{eq} \wedge \cdots \wedge L_u^{eq})). \quad (2)\end{aligned}$$

The formal definition of $\text{gp}(L)_\Pi$ is Algorithm 3.

Datalog queries as SPARQL$_C$ queries.
Given a Datalog query $Q = (\Pi, L)$ where Π is a Datalog program and L is the goal literal. The function $\mathcal{T}'_Q(Q)$ returns the SPARQL$_C$ query (P, D) where P is the graph pattern $\text{gp}(L)_\Pi$ and D is an RDF dataset with default graph $G_0 = \mathcal{T}'_D(\text{facts}(\Pi))$.

Algorithm 3 Transformation of Datalog rules into SPARQL_C graph patterns

//Input: a literal $L = p(x_1, \ldots, x_n)$ and a Datalog program Π
//Output: a SPARQL_C graph pattern $P = \text{gp}(L)_\Pi$
$P \leftarrow \emptyset$
if predicate p is extensional in Π **then**
  Let $?Y$ be a fresh variable
  $P \leftarrow ((?Y, \text{predicate}, p) \,\text{AND} (?Y, \text{rdf:\_1}, x_1) \,\text{AND} \cdots \text{AND} (?Y, \text{rdf\_n}, x_n))$
else if predicate p is intensional in Π **then**
  for each rule $r \in \Pi$ with head $p(x'_1, \ldots, x'_n)$ **do**
    $P' \leftarrow \emptyset$
    $C \leftarrow \emptyset$
    Let $r' = \nu(r)$ where ν is a substitution such that $\nu(x'_i) = x_i$
    for each positive literal $q(y_1, \ldots, y_m)$ in the body of r' **do**
      if $P' = \emptyset$ **then** $P' \leftarrow \text{gp}(q)_\Pi$
      else $P' \leftarrow (P' \,\text{AND}\, \text{gp}(q)_\Pi)$
    end for
    for each negative literal $\neg q(y_1, \ldots, y_m)$ in the body of r' **do**
      $P' \leftarrow (P' \,\text{MINUS}\, \text{gp}(q))$
    end for
    for each equality formula $t_1 = t_2$ in r' **do**
      if $C = \emptyset$ **then** $C \leftarrow (t_1 = t_2)$
      else $C \leftarrow C \wedge (t_1 = t_2)$
    end for
    for each negative literal $\neg(t_1 = t_2)$ in r' **do**
      if $C = \emptyset$ **then** $C \leftarrow \neg(t_1 = t_2)$
      else $C \leftarrow C \wedge \neg(t_1 = t_2)$
    end for
    if $C \neq \emptyset$ **then** $P' \leftarrow (P' \,\text{FILTER}\, C)$
    if $P = \emptyset$ **then** $P \leftarrow P'$
    else $P \leftarrow (P \,\text{UNION}\, P')$
  end for
end if
return P

The following theorem states that the above transformations work well.

Theorem 5. *nr-Datalog*$^\neg$ *is contained in* SPARQL_C.

Proof. We need to prove that for every Datalog query $Q = (\Pi, L)$ it satisfies that $\mathcal{T}'_S(\text{ans}_d(Q, \text{facts}(\Pi)) = \text{ans}_s(\mathcal{T}'_Q(Q), \mathcal{T}'_D(\text{facts}(\Pi)))$. Considering that $\text{ans}_s(\cdots)$ denotes function $[\![\cdot]\!]$, we will show that $\mathcal{T}'_S(\text{ans}_d(Q, \text{facts}(\Pi)) = [\![\text{gp}(L)_\Pi]\!]^D_{\text{dg}(D)}$ where $\text{dg}(D) = \mathcal{T}'_D(\text{facts}(\Pi))$.

The proof is by induction on the level l of the program (Π, L). The level of a program (Π, L) is the number $l(L)$ where: $l(\neg L) = l(L)$; $l(L) = 0$ if L contains an extensional predicate; $l(L) = 1 + \max_i(l(L_i))$ if L contains an intensional predicate and L_i are all literals which occur in the body of any rule with head L. (Note that the function is well defined because the Datalog programs considered are not recursive.)

Base case: $l(\Pi, L) = 0$.
Let $L = p(x_1, \ldots, x_n)$. In this case p is extensional and L matches line 4 of Algorithm 3. Hence $\text{gp}(L)_\Pi$ returns the graph pattern

$$P = ((?Y, \text{predicate}, p) \,\text{AND}(?Y, \text{rdf:\_1}, x_1) \,\text{AND} \cdots \text{AND}(?Y, \text{rdf:\_n}, x_n)).$$

Now, a mapping μ is in $[\![P]\!]^D_{\text{dg}(D)}$ if and only if for every triple pattern t in P it satisfies that $\mu(t) \in \text{dg}(D)$.

On the other hand, a substitution θ is in $\text{ans}_d((\Pi, L), \text{facts}(\Pi))$ if and only if $\theta(L) \in \text{facts}(\Pi)$ (Note that we only consider the initial database $\text{facts}(\Pi)$ because predicate p is extensional).

Note that $\mathcal{T}'_S$ maps bijectively substitutions from $\text{ans}_d((\Pi, L), \text{facts}(\Pi))$ to mappings in $[\![\text{gp}(L)_\Pi]\!]^D_{\text{dg}(D)}$. Specifically, for each variable $v \in L$ it satisfies that $\theta(v) = \mu(v)$. This proves the basic case.

Inductive step: $l(\Pi, L) = n > 0$.
Recall that $L = p(x_1, \ldots, x_n)$ and assume that Π_p denotes the set of rules of Π having predicate p in the head. In this case, L matches line 7 of Algorithm 3 and $\text{gp}(L)_\Pi$ returns the graph pattern

$$(\ \text{gp}(L^{r_1})_\Pi \,\text{UNION} \cdots \text{UNION}\, \text{gp}(L^{r_m})_\Pi\ ), \tag{3}$$

where $\text{gp}(L^{r_i})_\Pi$ returns the graph pattern corresponding to rule $r_i \in \Pi_p$. In this case it clearly holds that $[\![\text{gp}(L)_\Pi]\!]^D_{\text{dg}(D)} = \bigcup_i [\![\text{gp}(L^{r_i})_\Pi]\!]^D_{\text{dg}(D)}$.

On the other hand, a substitution θ is in $\text{ans}_d((\Pi, L), \text{facts}(\Pi))$ iff there is a rule $r_i \in \Pi_p$ such that $\theta'(r_i)$ is true in Π. Considering (3), it is enough to prove that for each particular rule $r_i \in \Pi_p$ it satisfies that:

$$\mathcal{T}'_S(\text{ans}((\Pi, L^{r_i}), \text{facts}(\Pi))) = [\![\text{gp}(L^{r_i})]\!]^D_{\text{dg}(D)}. \tag{4}$$

To prove this, assume that the rule r_i has the following general structure:

$$L \ \leftarrow\ L_1 \wedge \cdots \wedge L_s \wedge \neg L_{s+1} \wedge \cdots \wedge \neg L_t \wedge L^{eq}_1 \wedge \cdots \wedge L^{eq}_u, \tag{5}$$

where each L_j is a predicate formula (positive or negative) and each L^{eq}_k is a literal of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$.

Let us compute the SPARQL_C evaluation first. We have that $\text{gp}(L)_\Pi$ returns a graph pattern with the structure

$$\begin{aligned}(((\cdots((\text{gp}(L_1)_\Pi \,\text{AND} \cdots \text{AND}\, \text{gp}(L_s)_\Pi)&\\ \text{MINUS}\, \text{gp}(L_{s+1})_\Pi) \cdots)\, \text{MINUS}\, \text{gp}(L_t)_\Pi)&\\ \text{FILTER}(L^{eq}_1 \wedge \cdots \wedge L^{eq}_u)),&\end{aligned} \tag{6}$$

Observe that a mapping μ is in $[\![\text{gp}(L)_\Pi]\!]^D_{\text{dg}(D)}$ if and only if:

(i) for each L_i with $1 \le i \le s$, there exists a mapping $\mu'_i \in [\![\text{gp}(L_i)_\Pi]\!]^D_{\text{dg}(D)}$ satisfying that μ and μ'_i are compatible;

(ii) for each L_j with $s < j \leq t$, there exists no mapping $\mu''_j \in [\![\mathrm{gp}(L_j)_\Pi]\!]^D_{\mathrm{dg}(D)}$ satisfying that μ and μ''_j are compatible; and
(iii) for each literal L^{eq}_k, it satisfies that $\mu(t_1) = \mu(t_2)$ when L^{eq}_k is $t_1 = t_2$, and $\mu(t_1) \neq \mu(t_2)$ when L^{eq}_k is $\neg(t_1 = t_2)$ (suppose that $\mu(t_i) = t_i$ where t_i is a constant).

Now, let us compute the Datalog evaluation. A substitution θ is in the result of $\mathrm{ans}_d((\Pi, L), \mathrm{facts}(\Pi))$ if and only if $\theta(L) \in \mathrm{facts}^*(\Pi)$. This means that:

(a) for each L_i with $1 \leq i \leq s$, there exists a substitution θ'_i in the result of $\mathrm{ans}_d((\Pi, L_i), \mathrm{facts}(\Pi))$ satisfying that $\theta(x) = \theta'(x)$ for each variable $x \in \mathrm{var}(\theta') \cap \mathrm{var}(\theta'_i)$, .
(b) for each L_j with $s < j \leq t$, there exists no substitution θ''_j in the result of $\mathrm{ans}_d((\Pi, L_j), \mathrm{facts}(\Pi))$ satisfying that $\theta(x) = \theta''(x)$ for each variable $x \in \mathrm{var}(\theta) \cap \mathrm{var}(\theta''_j)$.
(c) for each literal L^{eq}_k, it satisfies that $\theta'(t_1) = \theta'(t_2)$ when L^{eq}_k is $t_1 = t_2$, and $\theta'(t_1) \neq \theta'(t_2)$ when L^{eq}_k is $\neg(t_1 = t_2)$ (assume that $\theta'(t_i) = t_i$ where t_i is a constant).

Note that (because Π is not recursive), for each pair of literal L_i, L_j in rule r_i, it holds that $l(\Pi, L_i) < l(\Pi, L)$ and $l(\Pi, L_j) < l(\Pi, L)$. Hence, by induction hypothesis we have that $\mathcal{T}'_S(\mathrm{ans}_d((\Pi, L_i), \mathrm{facts}(\Pi))) = [\![\mathrm{gp}(L_i)_\Pi]\!]^D_{\mathrm{dg}(D)}$ and $\mathcal{T}'_S(\mathrm{ans}_d((\Pi, L_j), \mathrm{facts}(\Pi))) = [\![\mathrm{gp}(L_j)_\Pi]\!]^D_{\mathrm{dg}(D)}$. These identities plus the conditions (i), (ii), (iii) and (a), (b), (c) above, show the bijections between maps $\mu \in [\![\mathrm{gp}(L)_\Pi]\!]^D_{\mathrm{dg}(D)}$ and substitutions $\theta \in \mathrm{ans}((\Pi, L), D_d)$, that is:

$$\mathcal{T}'_S(\mathrm{ans}((\Pi, L), \mathrm{facts}(\Pi))) = [\![\mathrm{gp}(L)_\Pi]\!]^D_{\mathrm{dg}(D)}.$$

This concludes the proof.

7 Conclusions

We have studied the expressive power of SPARQL. Among the most important findings are the definition of negation, the proof that non-safe filter patterns are superfluous, the proof of the equivalence between $\mathrm{SPARQL_{WG}}$ and $\mathrm{SPARQL_C}$.

From these results we can state the most relevant result of the paper:

Theorem 6 (main). *$SPARQL_{WG}$ has the same expressive power as Relational Algebra under bag semantics.*

This result follows from the well known fact (for example, see [1] and [5]) that relational algebra and non-recursive safe Datalog with negation have the same expressive power, and from theorems 2, 3, 4 and 5.

Relational Algebra is probably one of the most studied query languages, and has become a favorite by theoreticians because of a proper balance between expressiveness and complexity. The result that SPARQL is equivalent in its expressive power to Relational Algebra, has important implications which are not discussed in this paper. Some examples are the translation of some results from Relational Algebra into SPARQL, and the settlement of several open questions about expressiveness of SPARQL, e.g., the expressive power added by the operator *bound* in combination with optional patterns. Future work includes the development of the manifold consequences implied by the Main Theorem.

Acknowledgments. R. Angles was supported by Mecesup project No. UCH0109. R. Angles and C. Gutierrez were supported by FONDECYT project No. 1070348. The authors wish to thank the reviewers for their comments.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. R. Cyganiak. A relational algebra for sparql. Technical Report HPL-2005-170, HP Labs, 2005.
3. T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In *Reasoning Web*, number 4126 in LNCS, pages 1–52, 2006.
4. G. Klyne and J. Carroll. Resource Description Framework (RDF) Concepts and Abstract Syntax. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`, February 2004.
5. M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999.
6. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, number 4273 in LNCS, pages 30–43. Springer-Verlag, 2006.
7. J. Pérez, M. Arenas, and C. Gutierrez. Semantics of SPARQL. Technical Report TR/DCC-2006-17, Department of Computer Science, Universidad de Chile, 2006.
8. A. Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th International World Wide Web Conference (WWW)*, pages 787–796. ACM, 2007.
9. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`, January 2008.
10. S. Schenk. A sparql semantics based on datalog. In *30th Annual German Conference on Advances in Artificial Intelligence (KI)*, volume 4667 of *LNCS*, pages 160–174. Springer, 2007.