The Expressive Power of SPARQL

Renzo Angles and Claudio Gutierrez

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 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. Sheth et al. (Eds.): ISWC 2008, LNCS 5318, pp. 114–129, 2008.

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.

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.

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

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, term(G) denotes the set of elements of T occurring in G and 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 dg(D). Each pair $\langle u_i, G_i\rangle$ is called a *named graph*; define name$(G_i)_D = u_i$ and gr$(u_i)_D = G_i$. We denote by term(D) the set of terms occurring in the graphs of D. The set of IRIs $\{u_1, \ldots, u_n\}$ is denoted 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) blank$(G_i) \cap$ 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 var(α) 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*.

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

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

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, excluding bound$(\cdot)$, and var$(C) \nsubseteq$ dom(μ), then $\mu(C) = error$; else if C is $?X = u$ and $\mu(?X) = u$, or if C is $?X = ?Y$ and $\mu(?X) = \mu(?Y)$, or if C is isIRI$(?X)$ and $\mu(?X) \in I$, if C is isLiteral$(?X)$ and $\mu(?X) \in L$, if C is isBlank$(?X)$ and $\mu(?X) \in B$, then $\mu(C) = true$; otherwise $\mu(C) = false$.
- If C is bound$(?X)$ then $\mu(C) = true$ if $?X \in$ dom(μ) else $\mu(C) = false$.[3]
- If C is $(\neg C_1)$ then $\mu(C) = true$ when $\mu(C_1) = false$; $\mu(C) = false$ when $\mu(C_1) = true$; and $\mu(C) = error$ when $\mu(C_1) = error$.
- If C is $(C_1 \vee C_2)$ then $\mu(C) = true$ if either $\mu(C_1) = true$ or $\mu(C_2) = true$; $\mu(C) = false$ if $\mu(C_1) = false$ and $\mu(C_2) = false$; otherwise $\mu(C) = error$.
- If C is $(C_1 \wedge C_2)$ then $\mu(C) = true$ if $\mu(C_1) = true$ and $\mu(C_2) = true$; $\mu(C) = false$ if either $\mu(C_1) = false$ or $\mu(C_2) = false$; otherwise $\mu(C) = error$.

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:

$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}$
$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$
$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible } \}$
$\Omega_1 \setminus_C \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible } \} \cup$
$\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ such that } \mu_1 \sim \mu_2, (\mu_1 \cup \mu_2) \not\models C\}$
$\Omega_1 \text{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$
$\Omega_1 \text{⟕}_C \Omega_2 = \{\mu \mid \mu \in (\Omega_1 \bowtie \Omega_2) \text{ and } \mu \models C\} \cup (\Omega_1 \setminus_C \Omega_2)$

Syntax and Semantics of SPARQL$_{\mathbf{C}}$.
A SPARQL$_{\text{C}}$ 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$_{\text{C}}$ graph pattern P over an RDF dataset D having active graph G, denoted $[\![P]\!]_G^D$, is defined recursively as follows:

- if P is a triple pattern t, $[\![P]\!]_G^D = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in G\}$ 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$_{\mathbf{WG}}$.
A SPARQL$_{\text{WG}}$ graph pattern `GroupGP` is defined by the following grammar[4]:

```
GroupGP        ::= "{" TB? ((GPNotTriples | Filter) "."? TB?)* "}"
GPNotTriples   ::= OptionalGP | GroupOrUnionGP | GraphGP
```

[3] Functions invoked with an argument of the wrong type are evaluated to *error*.
[4] `http://www.w3.org/TR/rdf-sparql-query/#grammar`. We use `GP` and `TB` to abbreviate `GraphPattern` and `TriplesBlock` respectively

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

```
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.

The evaluation of a SPARQL$_{\mathrm{WG}}$ 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 over an RDF dataset D having active graph G, which we will denote $\langle\!\langle E \rangle\!\rangle_G^D$ (originally denoted $\mathrm{eval}(D(G), E)$ in [9]), is defined recursively as follows:

- if E is BGP(`TB`), $\langle\!\langle E \rangle\!\rangle_G^D = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{var}(E) \text{ and } \mu(E) \subseteq G\}$ 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 $\langle\!\langle P \rangle\!\rangle_G^D$ 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$_{\mathrm{C}}$ follows a compositional semantics, whereas SPARQL$_{\mathrm{WG}}$ 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.

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

```
1: // Input: a SPARQL_WG graph pattern GroupGP
2: // Output: an algebra expression E = T(GroupGP)
3: E ← empty pattern; FS ← ∅
4: for each syntactic form f in GroupGP do
5:    if f is TB then E ← Join(E, BGP(TB))
6:    if f is OPTIONAL GroupGP_1 then
7:      if T(GroupGP_1) is Filter(F, E') then E ← LeftJoin(E, E', F)
8:      else E ← LeftJoin(E, T(GroupGP_1), true)
9:    if f is GroupGP_1 UNION ··· UNION GroupGP_n then
10:     if n > 1 then
11:       E' ← Union(···(Union(T(GroupGP_1), T(GroupGP_2))···), T(GroupGP_n))
12:     else E' ← T(GroupGP_1)
13:     E ← Join(E, E')
14:   end if
15:   if f is GRAPH VarOrIRIref GroupGP_1 then
16:     E ← (E, Graph(VarOrIRIref, T(GroupGP_1)))
17:   if f is FILTER constraint then FS ← (FS ∧ constraint)
18: end for
19: if FS ≠ ∅ then E ← Filter(FS, E)
20: 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 $\langle\langle E \rangle\rangle_G^D$
$\text{Join}(E_1, E_2)$	$\langle\langle E_1 \rangle\rangle_G^D \bowtie \langle\langle E_2 \rangle\rangle_G^D$
$\text{LeftJoin}(E_1, E_2, C)$	$\langle\langle E_1 \rangle\rangle_G^D \bowtie_C \langle\langle E_2 \rangle\rangle_G^D$
$\text{Union}(E_1, E_2)$	$\langle\langle E_1 \rangle\rangle_G^D \cup \langle\langle E_2 \rangle\rangle_G^D$
$\text{Filter}(C, E_1)$	$\{ \mu \mid \mu \in \langle\langle E_1 \rangle\rangle_G^D \text{ and } \mu \models C\}$
$\text{Graph}(u, E_1)$	$\langle\langle E_1 \rangle\rangle_{\text{gr}(u)_D}^D$
$\text{Graph}(?X, E_1)$	$\bigcup_{v \in \text{names}(D)} (\langle\langle E_1 \rangle\rangle_{\text{gr}(v)_D}^D \bowtie \{\mu_{?X \to v}\})$

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.

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 $H \leftarrow B$ where H is a positive literal called the *head* [5] of the rule and B 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 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 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.

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 $D = \text{facts}(\Pi)$, denoted $\text{ans}_d(Q, D)$ is defined as the set of substitutions $\{\theta \mid \theta(L) \in \text{facts}^*(\Pi)\}$.

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 $\text{eval} : \mathcal{Q} \times \mathcal{D} \rightarrow \mathcal{S}$ is the evaluation function. The evaluation of a query $Q \in \mathcal{Q}$ on a database $D \in \mathcal{D}$ is denoted $\text{eval}(Q, D)$. 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 $D \in \mathcal{D}$.

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 $\mathcal{T}_D : \mathcal{D}_1 \rightarrow \mathcal{D}_2$ and $\mathcal{T}_S : \mathcal{S}_1 \rightarrow \mathcal{S}_2$, and query transformation $\mathcal{T}_Q : \mathcal{Q}_1 \rightarrow \mathcal{Q}_2$, 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.)

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

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

The 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 $\text{nov}(P)$, is a subset of the variables of P defined recursively as follows: $\text{nov}(P) = \text{var}(P)$ when P is a basic graph pattern; if P is either $(P_1 \,\text{AND}\, P_2)$ or $(P_1 \,\text{UNION}\, P_2)$ then $\text{nov}(P) = \text{nov}(P_1) \cup \text{nov}(P_2)$; if P is $(P_1 \,\text{OPT}\, P_2)$ then $\text{nov}(P) = \text{nov}(P_1)$; if P is $(n \,\text{GRAPH}\, P_1)$ then either $\text{nov}(P) = \text{nov}(P_1)$ when $n \in I$ or $\text{nov}(P) = \text{nov}(P_1) \cup \{n\}$ when $n \in V$; and $\text{nov}(P_1 \,\text{FILTER}\, C) = \text{nov}(P_1)$. Intuitively $\text{nov}(P)$ 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 $\text{var}(P) \cap \text{var}(\phi(P)) = \emptyset$.

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

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

where:

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

Note 4 (Why the copy pattern $\phi(P)$ is necessary?).
Consider the naive implementation of difference of patterns, 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 such implementation would fail when $\text{var}(P_2) \setminus \text{var}(P_1) = \emptyset$, because *there exist no variables* to check unboundedness.

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 the graph pattern P_2 does not match. The copy pattern ensure that there will exist a variable to check unboundedness.

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 $?Y \in \text{var}(P_2) \setminus \text{var}(P_1)$.)

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 the 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. The problem is produced by the expression $(\neg \text{bound}(?Z))$, because the bounding state of variable $?Z$ introduces noise when testing if pattern P_2 gets matched.

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

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

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

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

One influential point in the evaluation of patterns in 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 pattern P there is a filter-safe pattern 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 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 a logical value of 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:

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

Lemma 1. *For every pattern P, the pattern* $\text{safe}(P)$ *defined by Algorithm 2 is filter-safe and it holds* $\langle\!\langle P \rangle\!\rangle = \langle\!\langle \text{safe}(P) \rangle\!\rangle$.

Thus we proved:

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

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

```
1: // Input: a SPARQL_WG graph pattern P
2: // Output: a safe graph pattern P' ← safe(P)
3: P' ← ∅
4: if P is (P1 AND P2) then P' ← (safe(P1) AND safe(P2))
5: if P is (P1 UNION P2) then P' ← (safe(P1) UNION safe(P2))
6: if P is (n GRAPH P1) then P' ← (n GRAPH safe(P1))
7: if P is (P1 OPT P2) then
8:    if P2 is (P3 FILTER C) then
9:       P' ← (safe(P1) OPT(safe((P1 AND P2) FILTER C)))
10:   else P' ← (safe(P1) OPT safe(P2))
11: end if
12: if P is (P1 FILTER C) then
13:   if var(C) ⊆ var(safe(P1)) then P' ← (safe(P1) FILTER C)
14:   else
15:     for all ?X ∈ var(C) and ?X ∉ var(safe(P1)) 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' ← (safe(P1) FILTER C)
24:   end if
25: end if
26: return P'
```

5 Expressive Power of SPARQL$_{\text{WG}}$ is Equivalent to SPARQL$_{\text{C}}$

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$_{\text{C}}$ [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.*

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 SPARQL$^{\text{Safe}}_{\text{WG}}$ and SPARQL$_{\text{C}}$ differ. Specifically, given a graph pattern $P = (P_1 \text{ OPT}(P_2 \text{ FILTER } C))$, we have that SPARQL$^{\text{Safe}}_{\text{WG}}$ evaluates the algebra expression $\text{LeftJoin}(P_1, P_2, C)$, whereas SPARQL$_{\text{C}}$ evaluates P to the

expression $[\![P_1]\!] \bowtie [\![P_2 \text{ FILTER } C]\!]$, which is the same as the $\text{SPARQL}_{\text{WG}}$ algebra expression $\text{LeftJoin}(P_1, \text{Filter}(C, P_2), true)$.

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 L_s is contained in L_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 a dataset D, 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 [6]; each triple (v_1, v_2, v_3) in the default graph of D is encoded by a fact $triple(g_0, v_1, v_2, v_3)$; each named graph $\langle u, G\rangle$ in D is encoded by a fact $graph(u)$ and each triple in G is encoded by a fact $triple(u, v_1, v_2, v_3)$.

SPARQL$_C$ mappings as Datalog substitutions. Given a graph pattern P, a 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 $\theta \in \mathcal{T}_S(\Omega)$ satisfying that, for each $x \in \text{var}(P)$ 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 $\delta(P, g)_D$. The notion of compatible mappings is implemented by the rules: $comp(X, X, X) \leftarrow term(X)$, $comp(X, null, X) \leftarrow term(X)$ and $comp(null, X, X) \leftarrow term(X)$.

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 $\neg Null(?X)$.

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

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 graph pattern P, a dataset D with default graph G, and the SPARQL$_\text{C}$ query $Q = (P, D)$. The function $\mathcal{T}_Q(Q)$ returns the Datalog query $(\Pi, p(\overline{\text{var}}(P)))$ where Π is the Datalog program $\mathcal{T}_D(D) \cup \delta(P, g_0)_D$, g_0 identifies the default graph G, 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.*

6.2 From Datalog to SPARQL$_\text{C}$

To prove that L_d is contained in L_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$_\text{C}$ 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$_\text{C}$ mappings.

Datalog facts as an RDF Dataset. Given a Datalog fact $f = p(c_1, ..., c_n)$, consider that desc(f) = { (_: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 $\{\text{desc}(f) \mid f \in F\}$, 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 $L \leftarrow p_1 \wedge \cdots \wedge p_s \wedge \neg q_1 \wedge \cdots \wedge \neg q_t \wedge L_1^{eq} \wedge \cdots \wedge L_u^{eq}$, where L_k^{eq} are literals of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$, we have have that $\text{gp}(L)_\Pi$ returns a graph pattern with the structure

$$\begin{aligned}(((\cdots((&\text{gp}(p_1)_\Pi\,\text{AND} \cdots \text{AND}\,\text{gp}(p_s)_\Pi)\\ &\text{MINUS}\,\text{gp}(q_1))\cdots)\,\text{MINUS}\,\text{gp}(q_t))\\ &\text{FILTER}(L_1^{eq} \wedge \cdots \wedge L_u^{eq})).\end{aligned} \quad (2)$$

Algorithm 3. Transformation of Datalog rules into SPARQL$_{\mathrm{C}}$ graph patterns

1: //Input: a literal $L = p(x_1, \ldots, x_n)$ and a Datalog program Π
2: //Output: a SPARQL$_{\mathrm{C}}$ graph pattern $P = \mathrm{gp}(L)_\Pi$
3: $P \leftarrow \emptyset$
4: **if** predicate p is extensional in Π **then**
5: Let $?Y$ be a fresh variable
6: $P \leftarrow ((?Y, \mathrm{predicate}, p)\,\mathrm{AND}(?Y, \mathrm{rdf{:}\_1}, x_1)\,\mathrm{AND} \cdots \mathrm{AND}(?Y, \mathrm{rdf\_n}, x_n))$
7: **else if** predicate p is intensional in Π **then**
8: **for** each rule $r \in \Pi$ with head $p(x'_1, \ldots, x'_n)$ **do**
9: $P' \leftarrow \emptyset$
10: $C \leftarrow \emptyset$
11: Let $r' = \nu(r)$ where ν is a substitution such that $\nu(x'_i) = x_i$
12: **for** each positive literal $q(y_1, \ldots, y_m)$ in the body of r' **do**
13: **if** $P' = \emptyset$ **then** $P' \leftarrow \mathrm{gp}(q)_\Pi$
14: **else** $P' \leftarrow (P'\,\mathrm{AND}\,\mathrm{gp}(q)_\Pi)$
15: **end for**
16: **for** each negative literal $\neg q(y_1, \ldots, y_m)$ in the body of r' **do**
17: $P' \leftarrow (P'\,\mathrm{MINUS}\,\mathrm{gp}(q))$
18: **end for**
19: **for** each equality formula $t_1 = t_2$ in r' **do**
20: **if** $C = \emptyset$ **then** $C \leftarrow (t_1 = t_2)$
21: **else** $C \leftarrow C \wedge (t_1 = t_2)$
22: **end for**
23: **for** each negative literal $\neg(t_1 = t_2)$ in r' **do**
24: **if** $C = \emptyset$ **then** $C \leftarrow \neg(t_1 = t_2)$
25: **else** $C \leftarrow C \wedge \neg(t_1 = t_2)$
26: **end for**
27: **if** $C \neq \emptyset$ **then** $P' \leftarrow (P'\,\mathrm{FILTER}\,C)$
28: **if** $P = \emptyset$ **then** $P \leftarrow P'$
29: **else** $P \leftarrow (P\,\mathrm{UNION}\,P')$
30: **end for**
31: **end if**
32: **return** P

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

Datalog queries as SPARQL$_C$ queries. Given a Datalog program Π, a literal $L = p(x_1, \ldots, x_n)$, and the Datalog query $Q = (\Pi, L)$. The function $\mathcal{T}'_Q(Q)$ returns the SPARQL$_{\mathrm{C}}$ query (P, D) where P is the graph pattern $\mathrm{gp}(L)_\Pi$ and D is an RDF dataset with default graph $\mathcal{T}'_D(\mathrm{facts}(\Pi))$.

The following theorem states that the above transformations work well.

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

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 SPARQL$_{\mathrm{WG}}$ and SPARQL$_{\mathrm{C}}$.

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. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Cyganiak, R.: A relational algebra for sparql. Technical Report HPL-2005-170, HP Labs (2005)
3. Furche, T., Linse, B., Bry, F., Plexousakis, D., Gottlob, G.: RDF Querying: Language Constructs and Evaluation Methods Compared. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds.) Reasoning Web 2006. LNCS, vol. 4126, pp. 1–52. Springer, Heidelberg (2006)
4. Klyne, G., Carroll, J.: Resource Description Framework (RDF) Concepts and Abstract Syntax (February 2004), `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`
5. Levene, M., Loizou, G.: A Guided Tour of Relational Databases and Beyond. Springer, Heidelberg (1999)
6. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
7. Pérez, J., Arenas, M., Gutierrez, C.: Semantics of SPARQL. Technical Report TR/DCC-2006-17, Department of Computer Science, Universidad de Chile (2006)
8. Polleres, A.: From SPARQL to rules (and back). In: Proceedings of the 16th International World Wide Web Conference (WWW), pp. 787–796. ACM, New York (2007)
9. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (January 2008), `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`
10. Schenk, S.: A sparql semantics based on datalog. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNCS (LNAI), vol. 4667, pp. 160–174. Springer, Heidelberg (2007)