Semantics and Complexity of SPARQL

Jorge Pérez[1], Marcelo Arenas[2], and Claudio Gutierrez[3]

[1] Universidad de Talca, Chile
[2] Pontificia Universidad Católica de Chile
[3] Universidad de Chile

Abstract. SPARQL is the W3C candidate recommendation query language for RDF. In this paper we address systematically the formal study of SPARQL, concentrating in its graph pattern facility. We consider for this study simple RDF graphs without special semantics for literals and a simplified version of filters which encompasses all the main issues. We provide a compositional semantics, prove there are normal forms, prove complexity bounds, among others that the evaluation of SPARQL patterns is PSPACE-complete, compare our semantics to an alternative operational semantics, give simple and natural conditions when both semantics coincide and discuss optimization procedures.

1 Introduction

The Resource Description Framework (RDF) [12] is a data model for representing information about World Wide Web resources. Jointly with its release in 1998 as Recommendation of the W3C, the natural problem of querying RDF data was raised. Since then, several designs and implementations of RDF query languages have been proposed (see [9] for a recent survey). In 2004 the RDF Data Access Working Group (part of the Semantic Web Activity) released a first public working draft of a query language for RDF, called SPARQL [15]. Currently (August 2006) SPARQL is a W3C Candidate Recommendation.

Essentially, SPARQL is a graph-matching query language. Given a data source D, a query consists of a pattern which is matched against D, and the values obtained from this matching are processed to give the answer. The data source D to be queried can be composed of multiple sources. A SPARQL query consists of three parts. The *pattern matching part*, which includes several interesting features of pattern matching of graphs, like optional parts, union of patterns, nesting, filtering (or restricting) values of possible matchings, and the possibility of choosing the data source to be matched by a pattern. The *solution modifiers*, which once the output of the pattern has been computed (in the form of a table of values of variables), allows to modify these values applying classical operators like projection, distinct, order, limit, and offset. Finally, the *output* of a SPARQL query can be of different types: yes/no queries, selections of values of the variables which match the patterns, construction of new triples from these values, and descriptions of resources.

Although taken one by one the features of SPARQL are simple to describe and understand, it turns out that the combination of them makes SPARQL into

I. Cruz et al. (Eds.): ISWC 2006, LNCS 4273, pp. 30–43, 2006.

a complex language, whose semantics is far from being understood. In fact, the semantics of SPARQL currently given in the document [15], as we show in this paper, does not cover all the complexities brought by the constructs involved in SPARQL, and includes ambiguities, gaps and features difficult to understand. The interpretations of the examples and the semantics of cases not covered in [15] are currently matter of long discussions in the W3C mailing lists.

The natural conclusion is that work on formalization of the semantics of SPARQL is needed. A formal approach to this subject is beneficial for several reasons, including to serve as a tool to identify and derive relations among the constructors, identify redundant and contradicting notions, and to study the complexity, expressiveness, and further natural database questions like rewriting and optimization. To the best of our knowledge, there is no work today addressing this formalization systematically. There are proposals addressing partial aspects of the semantics of some fragments of SPARQL. There is also some work addressing formal issues of the semantics of query languages for RDF which can be of use for SPARQL. In fact, SPARQL shares several constructs with other proposals of query languages for RDF. In the related work section, we discuss these developments in more detail. None of these works, nevertheless, covers the problems posed by the core constructors of SPARQL from the syntactic, semantic, algorithmic and computational complexity point of view, which is the subject of this paper.

Contributions. An in depth analysis of the semantics benefits from abstracting some features, which although relevant, in a first stage tend to obscure the interplay of the basic constructors used in the language. One of our main goals was to isolate a core fragment of SPARQL simple enough to be the subject matter of a formal analysis, but which is expressive enough to capture the core complexities of the language. In this direction, we chose the graph pattern matching facility, which is additionally one of the most complex parts of the language. The fragment isolated consists of the grammar of patterns restricted to queries on one dataset (i.e. not considering the dataset graph pattern) over simple RDF graphs, not considering RDF/S vocabulary and without special semantics for literals. There are other two sources of abstractions which do not alter in essential ways SPARQL: we use set semantics as opposed to the bag semantics implied in the document of the W3C, and we avoid blanks in the syntax of patterns, because in our fragment can be replaced by variables [8,4].

The contributions of this paper are:

- A streamlined version of the core fragment of SPARQL with precise syntax and semantics. A formal version of SPARQL helps clarifying cases where the current English-wording semantics gives little information, identify areas of problems and permits to propose solutions.
- We present a compositional semantics for patterns in SPARQL, prove that there is a notion of normal form for graph patterns, and indicate optimization procedures and rules for the operators based on them.
- We give thorough analysis of the computational complexity of the fragment. Among other bounds, we prove that the complexity of evaluation of SPARQL general graph patterns is PSPACE-complete even without filter conditions.

– We formalize a natural procedural semantics which is implicitly used by developers. We compare these two semantics, the operational and the compositional mentioned above. We show that putting some slight and reasonable syntactic restrictions on the scope of variables, they coincide, thus isolating a natural fragment having a clear semantics and an efficient evaluation procedure.

Organization of the paper. Section 2 presents a formalized algebraic syntax and a compositional semantics for SPARQL. Section 3 presents the complexity study of the fragment considered. Section 4 presents and in depth discussion of graph patterns not including the UNION operator. Finally, Section 5 discusses related work and gives some concluding remarks.

2 Syntax and Semantics of SPARQL

In this section, we give an algebraic formalization of the core fragment of SPARQL over simple RDF, that is, RDF without RDFS vocabulary and literal rules. This allows us to take a close look at the core components of the language and identify some of its fundamental properties (for details on RDF formalization see [8], or [13] for a complete reference including RDFS vocabulary).

Assume there are pairwise disjoint infinite sets I, B, and L (IRIs, Blank nodes, and literals). A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an *RDF triple.* In this tuple, s is the *subject*, p the *predicate* and o the *object.* We denote by IL the union $I \cup L$, and by T the union $I \cup B \cup L$. Assume additionally the existence of an infinite set V of variables disjoint from the above sets.

Definition 1. *An* RDF graph *[11] is a set of RDF triples. In our context, we refer to an RDF graph as an* RDF dataset*, or simply a* dataset*.*

2.1 Syntax of SPARQL Graph Pattern Expressions

In order to avoid ambiguities in the parsing, we present the syntax of SPARQL graph patterns in a more traditional algebraic way, using the binary operators UNION, AND and OPT, and FILTER. We fully parenthesize expressions and make explicit the left associativity of OPT (`OPTIONAL`) and the precedence of AND over OPT implicit in [15].

A SPARQL graph pattern expression is defined recursively as follows:

(1) A tuple from $(IL \cup V) \times (I \cup V) \times (IL \cup V)$ is a graph pattern (a *triple pattern*).
(2) If P_1 and P_2 are graph patterns, then expressions (P_1 AND P_2), (P_1 OPT P_2), and (P_1 UNION P_2) are graph patterns.
(3) If P is a graph pattern and R is a SPARQL *built-in* condition, then the expression (P FILTER R) is a graph pattern.

A SPARQL *built-in* condition is constructed using elements of the set $V \cup IL$ and constants, logical connectives ($\neg$, $\wedge$, $\vee$), inequality symbols ($<$, $\leq$, $\geq$, $>$), the equality symbol ($=$), unary predicates like bound, isBlank, and isIRI, plus other features (see [15] for a complete list).

In this paper, we restrict to the fragment of filters where the built-in condition is a Boolean combination of terms constructed by using = and bound, that is:

(1) If $?X, ?Y \in V$ and $c \in I \cup L$, then bound($?X$), $?X = c$ and $?X = ?Y$ are built-in conditions.
(2) If R_1 and R_2 are built-in conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in conditions.

Additionally, we assume that for (P FILTER R) the condition $\text{var}(R) \subseteq \text{var}(P)$ holds, where var(R) and var(P) are the sets of variables occurring in R and P, respectively. Variables in R not occurring in P bring issues that are not computationally desirable. Consider the example of a built in condition R defined as $?X = ?Y$ for two variables not occurring in P. What should be the result of evaluating (P FILTER R)? We decide not to address this discussion here.

2.2 Semantics of SPARQL Graph Pattern Expressions

To define the semantics of SPARQL graph pattern expressions, we need to introduce some terminology. A *mapping* μ from V to T is a partial function $\mu : V \to T$. Abusing notation, for a triple pattern t we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, dom(μ), is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* when for all $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. Note that two mappings with disjoint domains are always compatible, and that the empty mapping (i.e. the mapping with empty domain) $\mu_\emptyset$ is compatible with any other mapping. Let Ω_1 and Ω_2 be sets of mappings. We define the join of, the union of and the difference between Ω_1 and Ω_2 as:

$$\begin{aligned}\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\},\\ \Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\ \Omega_1 \smallsetminus \Omega_2 &= \{\mu \in \Omega_1 \mid \text{ for all } \mu' \in \Omega_2,\ \mu \text{ and } \mu' \text{ are not compatible}\}.\end{aligned}$$

Based on the previous operators, we define the left outer-join as:

$$\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2).$$

We are ready to define the semantics of graph pattern expressions as a function $[\![\cdot]\!]_D$ which takes a pattern expression and returns a set of mappings. We follow the approach in [8] defining the semantics as the set of mappings that matches the dataset D.

Definition 2. *Let D be an RDF dataset over T, t a triple pattern and P_1, P_2 graph patterns. Then the* evaluation *of a graph pattern over D, denoted by $[\![\cdot]\!]_D$, is defined recursively as follows:*

(1) $[\![t]\!]_D = \{\mu \mid \text{dom}(\mu) = \text{var}(t)$ and $\mu(t) \in D\}$, where var(t) is the set of variables occurring in t.

(2) $[\![(P_1 \text{ AND } P_2)]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$.
(3) $[\![(P_1 \text{ OPT } P_2)]\!]_D = [\![P_1]\!]_D \mathbin{⟕} [\![P_2]\!]_D$.
(4) $[\![(P_1 \text{ UNION } P_2)]\!]_D = [\![P_1]\!]_D \cup [\![P_2]\!]_D$.

Consider pattern expression $(P_1 \text{ OPT } P_2)$ and let μ_1 be a mapping in $[\![P_1]\!]_D$. If there exists a mapping $\mu_2 \in [\![P_2]\!]_D$ such that μ_1 and μ_2 are compatible, then $\mu_1 \cup \mu_2$ belongs to $[\![(P_1 \text{ OPT } P_2)]\!]_D$. But if no such a mapping μ_2 exists, then μ_1 belongs to $[\![(P_1 \text{ OPT } P_2)]\!]_D$. Thus, operator OPT (optional) allows information to be added to a mapping μ if the information is available, instead of just rejecting μ whenever some part of the pattern does not match.

The semantics of FILTER expressions goes as follows. Given a mapping μ and a built-in condition R, we say that μ satisfies R, denoted by $\mu \models R$, if:

(1) R is bound$(?X)$ and $?X \in \text{dom}(\mu)$;
(2) R is $?X = c$, $?X \in \text{dom}(\mu)$ and $\mu(?X) = c$;
(3) R is $?X =?Y$, $?X \in \text{dom}(\mu)$, $?Y \in \text{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$;
(4) R is $(\neg R_1)$, R_1 is a built-in condition, and it is not the case that $\mu \models R_1$;
(5) R is $(R_1 \vee R_2)$, R_1 and R_2 are built-in conditions, and $\mu \models R_1$ or $\mu \models R_2$;
(6) R is $(R_1 \wedge R_2)$, R_1 and R_2 are built-in conditions, $\mu \models R_1$ and $\mu \models R_2$.

Definition 3. *Given an RDF dataset D and a FILTER expression* $(P \text{ FILTER } R)$*,*

$$[\![(P \text{ FILTER } R)]\!]_D = \{\mu \in [\![P]\!]_D \mid \mu \models R\}.$$

Example 1. Consider the RDF dataset D:

$$\begin{array}{llll}
D = \{ & (B_1, \text{name}, \text{paul}), & (B_1, \text{phone}, \text{777-3426}), & \\
& (B_2, \text{name}, \text{john}), & (B_2, \text{email}, \text{john@acd.edu}), & \\
& (B_3, \text{name}, \text{george}), & (B_3, \text{webPage}, \text{www.george.edu}), & \\
& (B_4, \text{name}, \text{ringo}), & (B_4, \text{email}, \text{ringo@acd.edu}), & \\
& (B_4, \text{webPage}, \text{www.starr.edu}), & (B_4, \text{phone}, \text{888-4537}), & \}
\end{array}$$

The following are graph pattern expressions and their evaluations over D according to the above semantics:

(1) $P_1 = ((?A, \text{email}, ?E) \text{ OPT } (?A, \text{webPage}, ?W))$. Then

$[\![P_1]\!]_D =$

	$?A$	$?E$	$?W$
μ_1 :	B_2	john@acd.edu	
μ_2 :	B_4	ringo@acd.edu	www.starr.edu

(2) $P_2 = (((?A, \text{name}, ?N) \text{ OPT } (?A, \text{email}, ?E)) \text{ OPT } (?A, \text{webPage}, ?W))$. Then

$[\![P_2]\!]_D =$

	$?A$	$?N$	$?E$	$?W$
μ_1 :	B_1	paul		
μ_2 :	B_2	john	john@acd.edu	
μ_3 :	B_3	george		www.george.edu
μ_4 :	B_4	ringo	ringo@acd.edu	www.starr.edu

(3) $P_3 = ((?A, \text{name}, ?N) \text{ OPT } ((?A, \text{email}, ?E) \text{ OPT } (?A, \text{webPage}, ?W)))$. Then

$[\![P_3]\!]_D =$

	$?A$	$?N$	$?E$	$?W$
μ_1 :	B_1	paul		
μ_2 :	B_2	john	john@acd.edu	
μ_3 :	B_3	george		
μ_4 :	B_4	ringo	ringo@acd.edu	www.starr.edu

Note the difference between $[\![P_2]\!]_D$ and $[\![P_3]\!]_D$. These two examples show that $[\![((A \text{ OPT } B) \text{ OPT } C)]\!]_D \neq [\![(A \text{ OPT } (B \text{ OPT } C))]\!]_D$ in general.

(4) $P_4 = ((?A, \text{name}, ?N) \text{ AND } ((?A, \text{email}, ?E) \text{ UNION } (?A, \text{webPage}, ?W)))$. Then

$[\![P_4]\!]_D =$

	$?A$	$?N$	$?E$	$?W$
μ_1 :	B_2	john	john@acd.edu	
μ_2 :	B_3	george		www.george.edu
μ_3 :	B_4	ringo	ringo@acd.edu	
μ_4 :	B_4	ringo		www.starr.edu

(5) $P_5 = (((?A, \text{name}, ?N) \text{ OPT } (?A, \text{phone}, ?P)) \text{ FILTER } \neg\,\text{bound}(?P))$. Then

$[\![P_5]\!]_D =$

	$?A$	$?N$	$?P$
μ_1 :	B_2	john	
μ_2 :	B_3	george	

2.3 A Simple Normal Form for Graph Patterns

We say that two graph pattern expressions P_1 and P_2 are *equivalent*, denoted by $P_1 \equiv P_2$, if $[\![P_1]\!]_D = [\![P_2]\!]_D$ for every RDF dataset D.

Proposition 1. *Let P_1, P_2 and P_3 be graph pattern expressions and R a built-in condition. Then:*

(1) AND *and* UNION *are associative and commutative.*
(2) $(P_1 \text{ AND } (P_2 \text{ UNION } P_3)) \equiv ((P_1 \text{ AND } P_2) \text{ UNION } (P_1 \text{ AND } P_3))$.
(3) $(P_1 \text{ OPT } (P_2 \text{ UNION } P_3)) \equiv ((P_1 \text{ OPT } P_2) \text{ UNION } (P_1 \text{ OPT } P_3))$.
(4) $((P_1 \text{ UNION } P_2) \text{ OPT } P_3) \equiv ((P_1 \text{ OPT } P_3) \text{ UNION } (P_2 \text{ OPT } P_3))$.
(5) $((P_1 \text{ UNION } P_2) \text{ FILTER } R) \equiv ((P_1 \text{ FILTER } R) \text{ UNION } (P_2 \text{ FILTER } R))$.

The application of the above equivalences permits to translate any graph pattern into an equivalent one of the form:

$$P_1 \quad \text{UNION} \quad P_2 \quad \text{UNION} \quad P_3 \quad \text{UNION} \quad \cdots \quad \text{UNION} \quad P_n, \tag{1}$$

where each P_i $(1 \leq i \leq n)$ is a UNION-free expression. In Section 4, we study UNION-free graph pattern expressions.

3 Complexity of Evaluating Graph Pattern Expressions

A fundamental issue in every query language is the complexity of query evaluation and, in particular, what is the influence of each component of the language in this complexity. In this section, we address these issues for graph pattern expressions.

As it is customary when studying the complexity of the evaluation problem for a query language, we consider its associated decision problem. We denote this problem by EVALUATION and we define it as follows:

INPUT : An RDF dataset D, a graph pattern P and a mapping μ.
QUESTION : Is $\mu \in [\![P]\!]_D$?

We start this study by considering the fragment consisting of graph pattern expressions constructed by using only AND and FILTER operators. This simple fragment is interesting as it does not use the two most complicated operators in SPARQL, namely UNION and OPT. Given an RDF dataset D, a graph pattern P in this fragment and a mapping μ, it is possible to efficiently check whether $\mu \in [\![P]\!]_D$ by using the following algorithm. First, for each triple t in P, verify whether $\mu(t) \in D$. If this is not the case, then return *false*. Otherwise, by using a bottom-up approach, verify whether the expression generated by instantiating the variables in P according to μ satisfies the FILTER conditions in P. If this is the case, then return *true*, else return *false*. Thus, we conclude that:

Theorem 1. Evaluation *can be solved in time* $O(|P| \cdot |D|)$ *for graph pattern expressions constructed by using only* AND *and* FILTER *operators.*

We continue this study by adding to the above fragment the UNION operator. It is important to notice that the inclusion of UNION in SPARQL is one of the most controversial issues in the definition of this language. In fact, in the W3C candidate recommendation for SPARQL [15], one can read the following: *"The working group decided on this design and closed the disjunction issue without reaching consensus. The objection was that adding UNION would complicate implementation and discourage adoption"*. In the following theorem, we show that indeed the inclusion of UNION operator makes the evaluation problem for SPARQL considerably harder:

Theorem 2. Evaluation *is NP-complete for graph pattern expressions constructed by using only* AND*,* FILTER *and* UNION *operators.*

We conclude this study by adding to the above fragments the OPT operator. This operator is probably the most complicated in graph pattern expressions and, definitively, the most difficult to define. The following theorem shows that the evaluation problem becomes even harder if we include the OPT operator:

Theorem 3. Evaluation *is PSPACE-complete for graph pattern expressions.*

It is worth mentioning that in the proof of Theorem 3, we actually show that Evaluation remains PSPACE-complete if we consider expressions without FILTER conditions, showing that the main source of complexity in SPARQL comes from the combination of UNION and OPT operators.

When verifying whether $\mu \in [\![P]\!]_D$, it is natural to assume that the size of P is considerably smaller that the size of D. This assumption is very common when studying the complexity of a query language. In fact, it is named data-complexity in the database literature [19] and it is defined as the complexity of the evaluation problem for a fixed query. More precisely, for the case of SPARQL, given a graph pattern expression P, the evaluation problem for P, denoted by Evaluation(P), has as input an RDF dataset D and a mapping μ, and the problem is to verify whether $\mu \in [\![P]\!]_D$. From known results for the data-complexity of first-order logic [19], it is easy to deduce that:

Theorem 4. EVALUATION(P) *is in LOGSPACE for every graph pattern expression P.*

4 On the Semantics of UNION-Free Pattern Expressions

The exact semantics of graph pattern expressions has been largely discussed on the mailing list of the W3C. There seems to be two main approaches proposed to compute answers to a graph pattern expression P. The first uses an operational semantics and consists essentially in the execution of a depth-first traversal of the parse tree of P and the use of the intermediate results to avoid some computations. This approach is the one followed by ARQ [1] (a language developed by HPLabs) in the cases we test, and by the W3C when evaluating graph pattern expressions containing nested optionals [17]. For instance, the computation of the mappings satisfying (A OPT (B OPT C)) is done by first computing the mappings that match A, then checking which of these mappings match B, and for those who match B checking whether they also match C [17]. The second approach, compositional in spirit and the one we advocate here, extends classical conjunctive query evaluation [8] and is based on a bottom up evaluation of the parse tree, borrowing notions of relational algebra evaluation [3,10] plus some additional features.

As expected, there are queries for which both approaches do not coincide (see Section 4.1 for examples). However, both semantics coincide in most of the "real-life" examples. For instance, for all the queries in the W3C candidate recommendation for SPARQL, both semantics coincide [15]. Thus, a natural question is what is the exact relationship between the two approaches mentioned above and, in particular, whether there is a "natural" condition under which both approaches coincide. In this section, we address these questions: Section 4.1 formally introduces the depth-first approach, discusses some issues concerning it, and presents queries for which the two semantics do not coincide; Section 4.2 identifies a natural and simple condition under which these two semantics are equivalent; Section 4.3 defines a normal form and simple optimization procedures for patterns satisfying the condition of Section 4.2

Based on the results of Section 2.3, we concentrate in the critical fragment of UNION-free graph pattern expressions.

4.1 A Depth-First Approach to Evaluate Graph Pattern Expressions

As we mentioned earlier, one alternative to evaluate graph pattern expressions is based on a "greedy" approach that computes the mappings satisfying a graph pattern expression P by traversing the parse tree of P in a depth-first manner and using the intermediate results to avoid some computations. This evaluation includes at each stage three parameters: the dataset, the subtree pattern of P to be evaluated, and a set of mappings already collected. Formally, given an RDF dataset D, the evaluation of pattern P with the set of mappings Ω, denoted by $Eval_D(P, \Omega)$, is a recursive function defined as follows:

```
$\mathit{Eval}_D$($P$: graph pattern expression, $\Omega$: set of mappings)
    if $\Omega = \emptyset$ then return($\emptyset$)
    if $P$ is a triple pattern $t$ then return($\Omega \bowtie [\![t]\!]_D$)
    if $P = (P_1 \text{ AND } P_2)$ then return $\mathit{Eval}_D(P_2, \mathit{Eval}_D(P_1, \Omega))$
    if $P = (P_1 \text{ OPT } P_2)$ then return $\mathit{Eval}_D(P_1, \Omega) ⟕ \mathit{Eval}_D(P_2, \mathit{Eval}_D(P_1, \Omega))$
    if $P = (P_1 \text{ FILTER } R)$ then return $\{\mu \in \mathit{Eval}_D(P_1, \Omega) \mid \mu \models R\}$
```

Then, the evaluation of P against a dataset D, which we denote simply by $\mathit{Eval}_D(P)$, is $\mathit{Eval}_D(P, \{\mu_\emptyset\})$, where $\mu_\emptyset$ is the mapping with empty domain.

Example 2. Assume that $P = (t_1 \text{ OPT } (t_2 \text{ OPT } t_3))$, where t_1, t_2 and t_3 are triple patterns. To compute $\mathit{Eval}_D(P)$, we invoke function $\mathit{Eval}_D(P, \{\mu_\emptyset\})$. This function in turn invokes function $\mathit{Eval}_D(t_1, \{\mu_\emptyset\})$, which returns $[\![t_1]\!]_D$ since t_1 is a triple pattern and $[\![t_1]\!]_D \bowtie \{\mu_\emptyset\} = [\![t_1]\!]_D$, and then it invokes $\mathit{Eval}_D((t_2 \text{ OPT } t_3), [\![t_1]\!]_D)$. As in the previous case, $\mathit{Eval}_D((t_2 \text{ OPT } t_3), [\![t_1]\!]_D)$ first invokes $\mathit{Eval}_D(t_2, [\![t_1]\!]_D)$, which returns $[\![t_1]\!]_D \bowtie [\![t_2]\!]_D$ since t_2 is a triple pattern, and then it invokes $\mathit{Eval}_D(t_3, [\![t_1]\!]_D \bowtie [\![t_2]\!]_D)$. Since t_3 is a triple pattern, the latter invocation returns $[\![t_1]\!]_D \bowtie [\![t_2]\!]_D \bowtie [\![t_3]\!]_D$. Thus, by the definition of Eval_D we have that $\mathit{Eval}_D((t_2 \text{ OPT } t_3), [\![t_1]\!]_D)$ returns $([\![t_1]\!]_D \bowtie [\![t_2]\!]_D) ⟕ ([\![t_1]\!]_D \bowtie [\![t_2]\!]_D \bowtie [\![t_3]\!]_D)$. Therefore, $\mathit{Eval}_D(P)$ returns

$$[\![t_1]\!]_D ⟕ \big(([\![t_1]\!]_D \bowtie [\![t_2]\!]_D) ⟕ ([\![t_1]\!]_D \bowtie [\![t_2]\!]_D \bowtie [\![t_3]\!]_D)\big).$$

Note that the previous result coincides with the evaluation algorithm proposed by the W3C for graph pattern $(t_1 \text{ OPT } (t_2 \text{ OPT } t_3))$ [17], as we first compute the mappings that match t_1, then we check which of these mappings match t_2, and for those who match t_2 we check whether they also match t_3. Also note that the result of $\mathit{Eval}_D(P)$ is not necessarily the same as $[\![P]\!]_D$ since $[\![(t_1 \text{ OPT } (t_2 \text{ OPT } t_3))]\!]_D = [\![t_1]\!]_D ⟕ ([\![t_2]\!]_D ⟕ [\![t_3]\!]_D)$. In Example 3 we actually show a case where the two semantics do not coincide.

Some issues in the depth-first approach. There are two relevant issues to consider when using the depth-first approach to evaluate SPARQL queries. First, this approach is not compositional. For instance, the result of $\mathit{Eval}_D(P)$ cannot in general be used to obtain the result of $\mathit{Eval}_D((P' \text{ OPT } P))$, or even the result of $\mathit{Eval}_D((P' \text{ AND } P))$, as $\mathit{Eval}_D(P)$ results from the computation of $\mathit{Eval}_D(P, \{\mu_\emptyset\})$ while $\mathit{Eval}_D((P' \text{ OPT } P))$ results from the computation of $\Omega = \mathit{Eval}_D(P', \{\mu_\emptyset\})$ and $\mathit{Eval}_D(P, \Omega)$. This can become a problem in cases of data integration where global answers are obtained by combining the results from several data sources; or when storing some pre–answered queries in order to obtain the results of more complex queries by composition. Second, under the depth-first approach some natural properties of widely used operators do not hold, which may confuse some users. For example, it is not always the case that $\mathit{Eval}_D((P_1 \text{ AND } P_2)) = \mathit{Eval}_D((P_2 \text{ AND } P_1))$, violating the commutativity of the conjunction and making the result to depend on the order of the query.

Example 3. Let D be the RDF dataset shown in Example 1 and consider the pattern $P = ((?X, \text{name}, \text{paul}) \text{ OPT } ((?Y, \text{name}, \text{george}) \text{ OPT } (?X, \text{email}, ?Z)))$.

Then $[\![P]\!]_D = \{\{?X \rightarrow B_1\}\}$, that is, $[\![P]\!]_D$ contains only one mapping. On the other hand, following the recursive definition of $Eval_D$ we obtain that $Eval_D(P) = \{\{?X \rightarrow B_1, ?Y \rightarrow B_3\}\}$, which is different from $[\![P]\!]_D$.

Example 4 (Not commutativity of AND*).* Let D be the RDF dataset in Example 1, P_1 = ((?X, name, paul) AND ((?Y, name, george) OPT (?X, email, ?Z))) and P_2 = (((?Y, name, george) OPT (?X, email, ?Z)) AND (?X, name, paul)). Then $Eval_D(P_1) = \{\{?X \rightarrow B_1, ?Y \rightarrow B_3\}\}$ while $Eval_D(P_2) = \emptyset$. Using the compositional semantics, we obtain $[\![P_1]\!]_D = [\![P_2]\!]_D = \emptyset$.

Let us mention that ARQ [1] gives the same non-commutative evaluation.

4.2 A Natural Condition Ensuring $[\![P]\!]_D = Eval_D(P)$

If for a pattern P we have that $[\![P]\!]_D = Eval_D(P)$ for every RDF dataset D, then we have the best of both worlds for P as the compositional approach gives a formal semantics to P while the depth-first approach gives an efficient way of evaluating it. Thus, it is desirable to identify natural syntactic conditions on P ensuring $[\![P]\!]_D = Eval_D(P)$. In this section, we introduce one such condition.

One of the most delicate issues in the definition of a semantics for graph pattern expressions is the semantics of OPT operator. A careful examination of the conflicting examples reveals a common pattern: A graph pattern P mentions an expression $P' = (P_1$ OPT $P_2)$ and a variable ?X occurring both inside P_2 and outside P' but not occurring in P_1. For instance, in the graph pattern expression shown in Example 3:

$$P = ((?X, \text{name, paul}) \text{ OPT } ((?Y, \text{name, george}) \text{ OPT } (?X, \text{email}, ?Z))),$$

variable ?X occurs both inside the optional part of the sub-pattern P' = ((?Y, name, george) OPT (?X, email, ?Z)) and outside P' in the triple (?X, name, paul), but it is not mentioned in (?Y, name, george).

What is unnatural about graph pattern P is the fact that (?X, email, ?Z) is giving optional information for (?X, name, paul) but in P appears as giving optional information for (?Y, name, george). In general, graph pattern expressions having the condition mentioned above are not natural. In fact, no queries in the W3C candidate recommendation for SPARQL [15] exhibit this condition. This motivates the following definition:

Definition 4. *A graph pattern P is* well designed *if for every occurrence of a sub-pattern $P' = (P_1$* OPT *$P_2)$ of P and for every variable ?X occurring in P, the following condition holds:*

if ?X occurs both inside P_2 and outside P', then it also occurs in P_1.

Graph pattern expressions that are not well designed are shown in Examples 3 and 4. For all these patterns, the two semantics differ. The next result shows a fundamental property of well-designed graph pattern expressions, and is a welcome surprise as a very simple restriction on graph patterns allows the users of SPARQL to alternatively use any of the two semantics shown in this section:

Theorem 5. *Let D be an RDF dataset and P a well-designed graph pattern expression. Then $\mathit{Eval}_D(P) = [\![P]\!]_D$.*

4.3 Well-Designed Patterns and Normalization

Due to the evident similarity between certain operators of SPARQL and relational algebra, a natural question is whether the classical results of normal forms and optimization in relational algebra are applicable in the SPARQL context. The answer is not straightforward, at least for the case of optional patterns and its relational counterpoint, the left outer join. The classical results about outer join query reordering and optimization by Galindo-Legaria and Rosenthal [7] are not directly applicable in the SPARQL context because they assume constraints on the relational queries that are rarely found in SPARQL. The first and more problematic issue, is the assumption on predicates used for joining (outer joining) relations to be *null-rejecting* [7]. In SPARQL, those predicates are implicit in the variables that the graph patterns share and by the definition of compatible mappings they are never *null-rejecting*. In [7] the queries are also enforced not to contain Cartesian products, situation that occurs often in SPARQL when joining graph patterns that do not share variables. Thus, specific techniques must be developed in the SPARQL context.

In what follows we show that the property of a pattern being well designed has important consequences for the study of normalization and optimization for a fragment of SPARQL queries. We will restrict in this section to graph patterns without FILTER.

Proposition 2. *Given a well-designed graph pattern P, if the left hand sides of the following equations are sub-patterns of P, then:*

$$(P_1 \text{ AND } (P_2 \text{ OPT } P_3)) \equiv ((P_1 \text{ AND } P_2) \text{ OPT } P_3), \tag{2}$$

$$((P_1 \text{ OPT } P_2) \text{ OPT } P_3) \equiv ((P_1 \text{ OPT } P_3) \text{ OPT } P_2). \tag{3}$$

Moreover, in both equivalences, if one replaces in P the left hand side by the right hand side, then the resulting pattern is still well designed.

From this proposition plus associativity and commutativity of AND, it follows:

Theorem 6. *Every well-designed graph pattern P is equivalent to a pattern in the following* normal form*:*

$$(\cdots(t_1 \text{ AND } \cdots \text{ AND } t_k) \text{ OPT } O_1) \text{ OPT } O_2)\cdots) \text{ OPT } O_n), \tag{4}$$

where each t_i is a triple pattern, $n \geq 0$ and each O_j has the same form (4).

The proof of the theorem is based on term rewriting techniques. The next example shows the benefits of using the above normal form.

Example 5. Consider dataset D of Example 1 and well-designed pattern $P = (((?X, \text{name}, ?Y) \text{ OPT } (?X, \text{email}, ?E)) \text{ AND } (?X, \text{phone}, \text{888-4537}))$. The normalized form of P is $P' = (((?X, \text{name}, ?Y) \text{ AND } (?X, \text{phone}, \text{888-4537})) \text{ OPT } (?X, \text{email}, ?E))$. The advantage of evaluating P' over P follows from a simple counting of maps.

Two examples of implicit use of the normal form. There are implementations (not ARQ[1]) that do not permit nested optionals, and when evaluating a pattern they first evaluate all patterns that are outside optionals and then *extend* the results with the matchings of patterns inside optionals. That is, they are implicitly using the normal form mentioned above. In [3], when evaluating a graph pattern with relational algebra, a similar assumption is made. First the join of all triple patterns is evaluated, and then the optional patterns are taken into account. Again, this is an implicit use of the normal form.

5 Related Work and Conclusions

Related Work. A rich source on the intended semantics of the constructors of SPARQL are the discussions around W3C document [15], which is still in the stage of Candidate Recommendation. Nevertheless, systematic and comprehensive approaches to define the semantics are not present, and most of the discussion is based on use cases.

In [15], in defining the semantics of SPARQL a notion of entailment is introduced with the idea of making the definition generic enough to support notions more general than simple entailment (e.g. OWL entailment [14], etc.). Current developments of the W3C (August 2006) have not settled yet this issue. What is clear consensus is that in the case of simple RDF any definition should coincide with subgraph matching, which is the approach followed in this paper.

Cyganiak [3] presents a relational model of SPARQL. The author uses relational algebra operators (join, left outer join, projection, selection, etc.) to model SPARQL `SELECT` clauses. The central idea in [3] is to make a correspondence between SPARQL queries and relational algebra queries over a single relation $T(S, P, O)$. Indeed a translation system between SPARQL and SQL is outlined. The system needs extensive use of `COALESCE` and `IS NULL` operations to resemble SPARQL features. The relational algebra operators and their semantics in [3] are similar to our operators and have similar syntactic and semantic issues. With different motivations, but similar philosophy, Harris [10] presents an implementation of SPARQL queries in a relational database engine. He uses relational algebra operators similar to [3]. This line of work, which models the semantics of SPARQL based on the semantics of some relational operators, seems to be very influent in the decisions on the W3C semantics of SPARQL.

De Bruin et al. [4] address the definition of mapping for SPARQL from a logical point of view. It slightly differs from the definition in [15] on the issue of blank nodes. Although De Bruin et al.'s definition allows blank nodes in graph patterns, it is similar to our definition which does not allow blanks in patterns. In their approach, these blanks play the role of "non-distinguished" variables, that is, variables which are not presented in the answer.

Franconi and Tessaris [5], in an ongoing work on the semantics of SPARQL, formally define the solution for a basic graph pattern (an RDF graph with variables) as a set of partial functions. They also consider RDF datasets and several forms of RDF–entailment. Finally, they propose high level operators (*Join*, *Optional*, etc.) that take set of mappings and give set of mappings, but currently they do not have formal definitions for them, stating only their types.

There are several works on the semantics of RDF query languages which tangentially touch the issues addressed by SPARQL. Gutierrez et al. [8] discuss the basic issues of the semantics and complexity of a conjunctive query language for RDF with basic patterns which underlies the basic evaluation approach of SPARQL. Haase et al. [9] present a comparison of functionalities of pre-SPARQL query languages, many of which served as inspiration for the constructs of SPARQL. Nevertheless, there is no formal semantics involved.

The idea of having an algebraic query language for RDF is not new. In fact, there are several proposals. Chen et al. [2] present a set of operators for manipulating RDF graphs, Frasincar et al. [6] study algebraic operators on the lines of the RQL query language, and Robertson [16] introduces an algebra of triadic relations for RDF. Although they evidence the power of having an algebraic approach to query RDF, the frameworks presented in each of these works makes not evident how to model with them the constructors of SPARQL.

Finally Serfiotis et al. [18] study RDFS query fragments using a logical framework, presenting results on the classical database problems of containment and minimization of queries for a model of RDF/S. They concentrate on patterns using the RDF/S vocabulary of classes and properties in conjunctive queries, making the overlap with our fragment and approach almost empty.

Concluding remarks. The query language SPARQL is in the process of standardization, and in this process the semantics of the language plays a key role. A formalization of a semantics will be beneficial on several grounds: help identify relationships among the constructors that stay hidden in the use cases, identify redundant and contradicting notions, study the expressiveness and complexity of the language, help in optimization, etc.

In this paper, we provided such a formal semantics for the graph pattern matching facility, which is the core of SPARQL. We isolated a fragment which is rich enough to present the main issues and favor a good formalization. We presented a formal semantics, made observations to the current syntax based on it, and proved several properties of it. We did a complexity analysis showing that unlimited used of OPT could lead to high complexity, namely PSPACE. We presented an alternative formal procedural semantics which closely resembles the one used by most developers. We proved that under simple syntactic restrictions both semantics are equivalent, thus having the advantages of a formal compositional semantics and the efficiency of a procedural semantics. Finally, we discussed optimization based on relational algebra and show limitations based on features of SPARQL. On these lines, we presented optimizations based on normal forms.

The approach followed in this paper for simple RDF can be extended to RDFS using the method proposed in [8], which introduces the notion of normal form for RDF graphs. This notion can be used to extend to RDFS the graph-theoretic characterization of simple RDF entailment. Then by replacing an RDF graph by its unique normal form, all the semantic results of this paper are preserved. Further work should include the extension of this approach to typed literals.

Acknowledgments. Pérez is supported by Dirección de Investigación, Universidad de Talca, Arenas by FONDECYT 1050701, and Arenas and Gutierrez by Millennium Nucleus Center for Web Research, P04-067-F, Mideplan, Chile.

References

1. *ARQ - A SPARQL Processor for Jena*, version 1.3 March 2006, Hewlett-Packard Development Company. `http://jena.sourceforge.net/ARQ`.
2. L. Chen, A. Gupta and M. E. Kurul. *A Semantic-aware RDF Query Algebra.* In *COMAD* 2005.
3. R. Cyganiak. *A Relational Algebra for Sparql.* HP-Labs Technical Report, HPL-2005-170. `http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html`.
4. J. de Bruijn, E. Franconi, S. Tessaris. *Logical Reconstruction of normative RDF.* In *OWLED 2005*, Galway, Ireland, November 2005
5. E. Franconi and S. Tessaris. *The Sematics of SPARQL.* Working Draft 2 November 2005. `http://www.inf.unibz.it/krdb/w3c/sparql/`.
6. F. Frasincar, C. Houben, R. Vdovjak and P. Barna. *RAL: An algebra for querying RDF.* In *WISE 2002.*
7. C. A. Galindo-Legaria and A. Rosenthal. *Outerjoin Simplification and Reordering for Query Optimization.* In *TODS* 22(1): 43–73, 1997.
8. C. Gutierrez, C. Hurtado and A. Mendelzon. *Foundations of Semantic Web Databases.* In *PODS 2004*, pages 95–106.
9. P. Haase, J. Broekstra, A. Eberhart and R. Volz. *A Comparison of RDF Query Languages.* In *ISWC 2004*, pages 502–517.
10. S. Harris. *Sparql query processing with conventional relational database systems.* In *SSWS 2005.*
11. G. Klyne, J. J. Carroll and B. McBride. *Resource Description Framework (RDF): Concepts and Abstract Syntax.* W3C Rec. 10 February 2004. `http://www.w3.org/TR/rdf-concepts/`.
12. F. Manola, E. Miller, B. McBride. *RDF Primer*, W3C Rec. 10 February 2004.
13. D. Marin. *RDF Formalization*, Santiago de Chile, 2004. Tech. Report Univ. Chile, TR/DCC-2006-8. `http://www.dcc.uchile.cl/~cgutierr/ftp/draltan.pdf`
14. Peter Patel-Schneider, Patrick Hayes and Ian Horrocks. *OWL Web Ontology Language Semantics and Abstract Syntax.* W3C Recommendation 10 February 2004, `http://www.w3.org/TR/owl-semantics/`.
15. E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF.* W3C Candidate Rec. 6 April 2006. `http://www.w3.org/TR/rdf-sparql-query/`.
16. E. L. Robertson. *Triadic Relations: An Algebra for the Semantic Web.* In *SWDB 2004*, pages 91–108
17. A. Seaborne. *Personal Communication.* April 13, 2006.
18. G. Serfiotis, I. Koffina, V. Christophides and V. Tannen. *Containment and Minimization of RDF/S Query Patterns.* In *ISWC 2005*, pages 607–623.
19. M. Vardi. *The Complexity of Relational Query Languages (Extended Abstract).* In *STOC 1982*, pages 137–146.