Title: SPARQL semantics: constructive or destructive?

Author: Fred Zemke
Date: July 27, 2006

1. Introduction

In a private conversation, Eric Prud'hommeaux asked me if my semantics was "constructive or destructive". In the flow of that conversation, I never answered the question. This paper contains my thoughts on that question.

My background is in relational databases. SQL defines the semantics of a query in two steps:

1. Form a cross product of all tables in the query.
2. Remove any rows that fail the WHERE clause.

The first step is constructive, the second step is destructive. (Note this is only the definition of the semantics; actual query engines are free to pursue other tactics as long as they compute the same result). The constructive step is typically expressed with very little verbiage, not much more than I have in item 1 above. The bulk of the specification is in the destructive step. Thus we could say that SQL's semantics is predominantly destructive.

This is the perspective that I brought to SPARQL. From this perspective, the first step (conceptually) is to form the set (or maybe multiset) of all possible rows, then determine which ones to retain in the answer.

Eric's question made me ask myself if there is another way. Filtered basic graph patterns appear amenable to the relational strategy: form the set of all mappings, and retain only those that match the pattern and the constraints. Group graph patterns are defined as an intersection, which also fits the "construct, then take away" mold. RDF dataset graph patterns also fit the destructive mold readily. Union and optional graph patterns can be shoehorned into this pattern as well, which I did in my previous draft of a formal semantics (titled "An attempt at a formal semantics for SPARQL", dated 14 June 2006). However, after performing the exercise in this paper, I now believe that union and optional graph pattern matching will be better expressed using a more constructive model.

This paper summarizes concisely two different semantics, a destructive-centric and a constructive-centric. I originally proposed the destructive-centric, and I now believe that constructive-centric is superior. Hopefully this work will contribute to a final specification of SPARQL semantics.

2. Preliminary terminology

I use the following terminology and symbols in both the destructive-centric and constructive-centric semantics.

I use union, intersect and minus as binary operators on sets, denoting set union, set intersection and set difference, respectively.

Let D be the dataset specified by the zero or more FROM clauses (if there are none, I presume some implementation-defined default dataset is used). Let D = { G_0, (<u_1, G_1>), . . . , (<u_n, G_n>) }. Let SC be the scoping set of G and let SC_0, SC_1, . . . , SC_n be the scoping sets of G_0, G_1, . . . , G_n. Let USC = SC_0 union SC_1 union SC_2 ... union SC_n.

Let V be the set of all variables. If Q is a query or a pattern, let Var(Q) be the set of variables that appear in Q.

A mapping is a function whose domain is a subset of V and whose codomain is USC. If S is a mapping, let dom(S) denote the domain of S.

If S_1 and S_2 are mappings, we say that S_1 is a restriction of S_2 if S_1 is a subset of S_2. Equivalently, S_1 is a restriction of S_2 if dom(S_1) is a subset of dom(S_2), and for all variables v in dom(S_1), $S_1(v) = S_2(v)$.

If S_2 is a mapping and U is a set of variables, then the restriction of S_2 to U is the mapping S_1 whose domain is (dom(S_2) intersect U), and such that for all v in (dom(S_2) intersect U), $S_1(v) = S_2(v)$.

In view of UNION patterns, a solution might occur more than once in a solution sequence. For definiteness, I believe we should define precisely how many times a solution appears in a solution sequence. Accordingly, my proposed semantics (both the destructive-centric and the constructive-centric) define a cardinality with every solution. I have heard a contrary opinion that the number of repetitions of a solution does not matter. If that view prevails, it is easy to strip the cardinality calculations out of my proposal.

3. Destructive-centric semantics

This is a semantics that is oriented around removing solutions from a cross product. The outline of such a semantics is as follows:

Constructive phase

Let Q be the query of interest.

Let M be the set of all mappings whose domain is a subset of Var(Q).

Destructive phase

The destructive phase is a recursive definition based on the syntactic possibilities of SPARQL grammar. The mappings that survive destruction are called solutions.

Filtered basic graph pattern: There seems to be a consensus that the proper scope for blank node identifiers in the query is the filtered basic graph pattern rather than the graph pattern (see http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0189.html http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0190.html , http://lists.w3.org/Archives/Public/public-rdf-dawg/2006JulSep/0005.html, and maybe http://lists.w3.org/Archives/Public/public-rdf-dawg/2006JulSep/0007.html). By filtered basic graph pattern I mean the graph pattern corresponding to syntax rule [21] FilteredBasicGraphPattern. Hence I write the following proposal to clarify section 2.5.1 “General framework”.

Let FBGP be a filtered basic graph pattern, consisting of a basic graph pattern BGP and a set of constraints C. Let G be the graph in which FBGP will be evaluated, with scoping set SC. (If a filtered basic graph pattern is not nested within an RDF dataset graph pattern, then $G = G_0$; otherwise $G = G_i$ for some i.) Let S be a mapping. Let S(BGP) be the set obtained by replacing all variables v in BGP with S(v), and replacing all blank node identifiers in BGP with blank nodes that are distinct from all blank nodes in G.

S is a solution of FBGP if the following conditions are all true:

1. dom(S) contains Var(BGP).

   Note that dom(S) does not need to contain Var(FBGP); otherwise there is no point to the BOUND predicate, which would always be true.

2. S(BGP) does not contain a triple whose subject is a literal. (Therefore S(BGP) is an RDF graph).

3. G entails (G union S(BGP)). Note that by design, the blank nodes of S(BGP) are already distinct from the blank nodes of G.

4. Every constraint in C evaluates to true in (G union S(BGP)).

   A constraint is evaluated in (G union S(BGP)) as follows:

   a) BOUND(v) is true iff v is in dom(S).

   b) For all other occurrences of variables in an expression in C, S is used to map the variable to a value. If an expression contains a variable that is not bound by S, the value of the variable is treated as an error.

   c) Expressions are evaluated from the leaves to the root of the parse tree of the expression, as explained in section 11 “Testing values”.

The cardinality of each solution of a filtered basic graph pattern is 1.

Optional graph pattern: The first step in treating an OPTIONAL is to identify its first operand. I believe that the approach proposed in Andy Seaborne’s message http://lists.w3.org/Archives/

Public/public-rdf-dawg/2006AprJun/0175.html looks like the best way to identify the first operand.

The second operand of an optional graph pattern is plainly the graph pattern contained in the curly braces following the keyword OPTIONAL.

After identifying the operands, the next step is to define the solutions. I use the definition proposed in http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0176.html , with some additional bug fixes that I have since discovered. Let A and B be the graph patterns that are the first and second operands, respectively. S is a solution of the optional graph pattern OPT(A,B) if the following conditions are true:

1. S restricted to Var(A) is a solution of A.
2. Exactly one of the following is true:

   a) S restricted to Var(B) is a solution of B.

   b) S is undefined on (Var(B) minus Var(A)), and there is no mapping which when restricted to Var(B) is a solution of B and whose restriction to Var(A) is S.

The cardinality of S as a solution to OPT(A,B) is the product of two numbers:

1. Let m be the cardinality of S restricted to Var(A) as a solution of A.
2. Let n be:

   a) If S is a solution of B, then the cardinality of S as a solution of B.

   b) Otherwise, 1.

The cardinality of S as a solution of OPT(A,B) is m*n.

Union graph pattern: Let UGP be a union graph pattern, ie, a set of graph patterns { GP_1, . . . GP_k }. S is a solution of UGP if there exists an i such that the following conditions hold:

1. S is a solution of GP_i.
2. For all j <> i, S is undefined on (Var(GP_j) minus Var(GP_i)).

   (The preceding condition seems to be the semantics intended by the examples in section 6.1 “Joining patterns with UNION”).

The cardinality of S as a solution of GGP is the sum as i ranges from 1 through k of the cardinality of S as a solution of GP_i (taking the cardinality to be 0 in case S is not a solution of GP_i).

Group graph pattern: Let GGP be a group graph pattern, i.e., a set of graph patterns { GP_1, . . . GP_k }. S is a solution of GGP if for all i, S is a solution of GP_i. The cardinality of S as a solution of GGP is the minimum as i ranges from 1 through k of the cardinality of S as a solution of GP_i.

RDF dataset graph pattern: Let RDGP be an RDF dataset graph pattern GRAPH (g, P). Then S is a solution of RDGP if either of these conditions is true:

1. g is an IRI, g = <u_i> for some i, and S is a solution of P on dataset { G_i, (<u_1>, G_1) . . . , (<u_n, G_n>) }.
2. g is a variable, S(g) = <u_j> for some j, and S is a solution of P on dataset { G_i, (<u_1>, G_1) . . . , (<u_n, G_n>) }.

The cardinality of S as a solution of RDGP is the cardinality of S as solution of P in either case.

4. Constructive-centric semantics

In the constructive-centric semantics, there is no destructive phase. In addition, the construction does not create a cross product. Instead, the construction builds solutions "from the ground up", that is, by recursion starting at the leaves of the query pattern and progressing to interior nodes of the conceptual tree of operators.

A key difference from the destructive-centric semantics is that a solution S of a pattern P is necessarily undefined on variables that do not appear in P. That is, dom(S) is a subset of Var(P).

Viewing the process of solving a query as a recursion on the query's subpatterns, in the constructive-centric semantics, a variable does not become bound until a pattern explicitly binds it.

The definitions for the various kinds of graph patterns are almost word-for-word identical. For completeness, I repeat these definitions, using underlining to indicate new words and ~~strikeout~~ to indicate deletions.

Filtered basic graph pattern: There seems to be a consensus that the proper scope for blank node identifiers in the query is the filtered basic graph pattern rather than the graph pattern (see http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0189.html http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0190.html , http://lists.w3.org/Archives/Public/public-rdf-dawg/2006JulSep/0005.html, and maybe http://lists.w3.org/Archives/Public/public-rdf-dawg/2006JulSep/0007.html). By filtered basic graph pattern I mean the graph pattern corresponding to syntax rule [21] FilteredBasicGraphPattern. Hence I write the following proposal to clarify section 2.5.1 "General framework".

Let FBGP be a filtered basic graph pattern, consisting of a basic graph pattern BGP and a set of constraints C. Let G be the graph in which FBGP will be evaluated, with scoping set SC. (If a filtered basic graph pattern is not nested within an RDF dataset graph pattern, then G = G_0; otherwise G = G_i for some i.) Let S be a mapping. Let S(BGP) be the set obtained by replacing all variables v in BGP with S(v), and replacing all blank node identifiers in BGP with blank nodes that are distinct from all blank nodes in G..

S is a solution of FBGP if the following conditions are all true:

1. dom(S) contains Var(BGP) and is contained in Var(FBGP).

2. S(BGP) does not contain a triple whose subject is a literal. (Therefore S(BGP) is an RDF graph).
3. G entails (G union S(BGP)). Note that by design, the blank nodes of S(BGP) are already distinct from the blank nodes of G.
4. Every constraint in C evaluates to true in (G union S(BGP)).

   A constraint is evaluated in (G union S(BGP)) as follows:

   a) BOUND(v) is true iff v is in dom(S).

   b) For all other occurrences of variables in an expression in C, S is used to map the variable to a value. If an expression contains a variable that is not bound by S, the value of the variable is treated as an error.

   c) Expressions are evaluated from the leaves to the root of the parse tree of the expression, as explained in section 11 “Testing values”.

The cardinality of each solution of a filtered basic graph pattern is 1.

Optional graph pattern: The first step in treating an OPTIONAL is to identify its first operand. I believe that the approach proposed in Andy Seaborne’s message http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0175.html looks like the best way to identify the first operand.

The second operand of an optional graph pattern is plainly the graph pattern contained in the curly braces following the keyword OPTIONAL.

After identifying the operands, the next step is to define the solutions. I use the definition proposed in http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0176.html , with some additional bug fixes that I have since discovered. Let A and B be the graph patterns that are the first and second operands, respectively. S is a solution of the optional graph pattern OPT(A,B) if the following conditions are true:

1. S restricted to Var(A) is a solution of A.
2. Exactly one of the following is true:

   a) S restricted to Var(B) is a solution of B.

   b) S is undefined on (Var(B) minus Var(A)), and there is no mapping which when restricted to Var(B) is a solution of B and whose restriction to Var(A) is S.

3. dom(S) is a subset of (Var(A) union Var(B)).

The cardinality of S as a solution to OPT(A,B) is the product of two numbers:

1. Let m be the cardinality of S restricted to Var(A) as a solution of A.
2. Let n be:

a) If S restricted to Var(B) is a solution of B, then the cardinality of S restricted to Var(B) as a solution of B.

b) Otherwise, 1.

The cardinality of S as a solution of OPT(A,B) is m*n.

Union graph pattern: Let UGP be a union graph pattern, ie, a set of graph patterns { GP_1, . . . GP_k }. S is a solution of UGP if there exists an i such that the following ~~two~~ condition~~s~~ holds:

1. S is a solution of GP_i.
2. ~~For all j <> i, S is undefined on (Var(GP_j) minus Var(GP_i)).~~

   (The preceding condition is not necessary with constructive semantics because a solution of a pattern is never defined on variables outside the pattern.)

The cardinality of S as a solution of GGP is the sum as i ranges from 1 through k of the cardinality of S as a solution of GP_i (taking the cardinality to be 0 in case S is not a solution of GP_i).

Group graph pattern: Let GGP be a group graph pattern, i.e., a set of graph patterns { GP_1, . . . GP_k }. S is a solution of GGP if for all i, S restricted to Var(GP_i) is a solution of GP_i. The cardinality of S as a solution of GGP is the minimum as i ranges from 1 through k of the cardinality of S restricted to Var(GP_i) as a solution of GP_i.

RDF dataset graph pattern: Let RDGP be an RDF dataset graph pattern GRAPH (g, P). Then S is a solution of RDGP if either of these conditions is true:

1. g is an IRI, g = $<u_i>$ for some i, and S is a solution of P on dataset { G_i, ($<u_1>$, G_1) . . . , ($<u_n$, $G_n>$) }.
2. g is a variable, S(g) = $<u_j>$ for some j, and S restricted to Var(P) is a solution of P on dataset { G_i, ($<u_1>$, G_1) . . . , ($<u_n$, $G_n>$) }.

The cardinality of S as a solution of RDGP is the cardinality of S as solution of P in either case.

5. Contrasting the semantics

With constructive-centric semantics, the solutions of a subpattern are never defined on variables that do not appear in that subpattern. Destructive-centric semantics, on the other hand, starts by forming the set of all mappings, so that solutions of subpatterns are just as "wide" as solutions of the whole pattern, being potentially defined on variables that do not appear in the subpattern.

One difference is in queries with unrestricted variables, such as

```
SELECT ?x
WHERE { }
```

Applying the grammar rules in Appendix A, the WHERE clause contains a single FilteredBasicGraphPattern (rule [21]) having no BlockOfTriples and no Constraint. Thus the rules for filtered basic graph pattern apply.

Under destructive-centric semantics, the constructive phase creates every possible mapping of the set of variables { ?x } to the scoping set. In the destructive phase, none of these mappings is eliminated, because:

1. G entails (G union S(BGP)). Note that BGP is the empty set, so S(BGP) is empty, so this reduces to the trivial condition that G entails G.
2. The set of constraints is empty, and therefore does not eliminate any mappings.

Under constructive-centric semantics, there is only one solution to the empty filtered basic graph pattern, namely, the empty mapping (the mapping whose domain is the empty set) because of the requirement that the domain of a solution must be a subset of Var(BGP), which is empty. One could still say that when the solution is widened to account for the ?x in the SELECT list, then every possible assignment of ?x is considered. However, it seems more natural to say that the SELECT list does not create new bindings, so ?x remains unbound and the solution sequence consists of just the empty mapping.

Another difference is in optional patterns with an empty first operand. For example

```
SELECT ?x
WHERE { OPTIONAL { ?x <v> <n> } }
```

The first operand of the OPTIONAL is an empty pattern. Under constructive-centric semantics, the empty first operand is matched by the empty pattern. Any matches for ?x <v> <n> would provide bindings for ?x; if there is no match, then the empty pattern is the result of the query as a whole.

Under the destructive-centric semantics, the empty first operand is matched by all possible mappings of ?x, and then there is no point to the OPTIONAL clause except to discard the empty mapping if there is a match. I think this result is non-intuitive, so that the destructive-centric semantics would need a patch to treat an OPTIONAL with an empty first operand as a special case to avoid this, probably to get the same result as the constructive-centric semantics.

Overall, I am coming to the conclusion that constructive-centric semantics is the better way to specify SPARQL. I can also see from clues sprinkled throughout the document that this was probably the intent of the authors before I joined the process anyway. If so, then I hope this paper will help us to express that intent even better.

- End of paper -