Title: Constructive mapping semantics for SPARQL

Author: Fred Zemke
Date: August 18, 2006

References

[SPARQL CR] "SPARQL query language for RDF", Candidate Recommendation, http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/

[rq24] "SPARQL query language for RDF", Editor's draft, http://www.w3.org/2001/sw/DataAccess/rq23/24 (the version I printed was dated 2006/06/12)

[Constructive/destructive]Fred Zemke, "SPARQL semantics: constructive or destructive?"

1. Introduction

1.1 Motivation

Many participants in the 8 August 2006 telecon expressed support (or at least interest) in basing the semantics of SPARQL on the mapping technique outlined in section 2.5.2 "SPARQL basic graph pattern matching" of [SPARQL CR] (= section 5.2 of [rq24]). This paper attempts to flesh out what the formal semantics might look like in that case, following the construtive approach that I outlined in [Constructive/destructive].

In addition, I personally have come to the conclusion that the "General framework" (section 2.5.1 in [SPARQL CR], section 5.1 in [rq24]) does not provide a satisfactory basis for the complete semantics of SPARQL. The problem is that, with this framework, the scope of a blank node identifier appears to be at most rule [21] FilteredBasicGraphPattern (see the email thread inaugarated by http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0189.html ; most respondents agreed with option 1 in my message). I believe this will be unacceptable in important usage scenarios because the scope does not extend across an OPTIONAL operator.

For example, a user might begin crafting a query to obtain name, homepage and mailbox:

```
SELECT *
FROM G
WHERE { _:a foaf:name ?name .
        _:a foaf:homepage ?homepage .
        _:a foaf:mbox ?mailbox }
```

Assuming semantics based on entailment, the user's query can be translated into the following pseudocode (I switch to lowercase to indicate this is not SPARQL):

```
select *
from G
where { G entails
        { G union { _:a foaf:name ?name .
                    _:a foaf:homepage ?homepage .
                    _:a foaf:mbox ?mailbox . } } }
```

(Here _:a must be taken as identifying a blank node that is distinct from all blank nodes in G.)

Alternatively, using an impromptu extension of SPARQL that makes existential quantifiers explicit (and again using lowercase to indicate this is not SPARQL), the user's query is:

```
select *
from G
where { (there exists _:a)
           { _:a foaf:name ?name .
             _:a foaf:homepage ?homepage .
             _:a foaf:mbox ?mailbox } }
```

So much for translations; back to the hypothetical user. After the user confirms that the SPARQL query is working, he then wishes to make the homepage and mailbox independently optional, so he edits the query slightly, as follows:

```
SELECT *
FROM G
WHERE { _:a foaf:name ?name .
        OPTIONAL { _:a foaf:homepage ?homepage }
        OPTIONAL { _:a foaf:mbox ?mailbox } }
```

However, using the entailment framework, the revised query will not do what the user expects, because there is no association between the three occurrences of _:a. Using the pseudocode with explicit entailment, the user's revised query is

```
select *
from G
where { G entails { G union { _:a foaf:name ?name } }
        optional { G entails { G union
                      { _:a foaf:homepage ?homepage } } }
        optional { G entails { G union
                      { _:a foaf:mbox ?mailbox } } } }
```

or using explicit existential quantifiers, the user's revised query is translated:

```
select *
from G
where { (there exists _:a) { _:a foaf:name ?name }
```

```
        optional { (there exists _:a)
                     { _:a foaf:homepage ?homepage } }
        optional { (there exists _:a)
                     { _:a foaf:mbox ?mailbox } } }
```

But what the user wants is to keep the existential quantifier “there exists _:a” at the outer level rather than pushing it down to each FilteredBasicGraphPattern. The desired translation with explicit quantifiers should be

```
select *
from G
where { (there exists _:a)
          { _:a foaf:name ?name .
            optional { _:a foaf:homepage ?homepage }
            optional { _:a foaf:mbox ?mailbox } }
```

However, the entailment framework does not support this semantic, because the blank node identifier is scoped to the FilteredBasicGraphPattern.

UNION has the same problem; for example

```
SELECT *
FROM G
WHERE { _:a foaf:name ?name
        { _:a foaf:homepage ?homepage }
        UNION { _:a foaf:mbox ?mailbox } }
```

Here again, the user would like _:a in the outer triple pattern to bind to the same value as _:a in either of the branches of the UNION.

Even group graph patterns have this problem. Consider

```
SELECT *
FROM G
WHERE { { _:a foaf:name ?name }
        { _:a foaf:homepage ?homepage }
        { _:a foaf:mbox ?mailbox } }
```

Here the user has taken the original query and simply enclosed each triple pattern in braces. With the entailment framework, each pair of braces delimits a FilteredBasicGraphPattern, and therefore a separate scope for _:a. Thus adding braces is more significant than just grouping (lparenthesizing); it also performs scoping. I believe that users will find this feature non-intuitive, though it is not so serious as the problem with OPTIONAL and UNION, since with group graph patterns the user has the syntactic possibility of erasing the braces.

1.2 Outline of the proposal

I believe that users will want that a blank node identifier, such as _:a in the examples, refers to the same thing uniformly throughout a query; that is, that the scope of a blank node identifier should

be the entire query rather than just a FilteredBasicGraphPattern. This paper hopes to show a way to do this by elaborating on the mapping technique found in [SPARQL CR] 2.5.2 “SPARQL basic graph pattern matching” (= [rq24] 5.2).

1.2.1 Bindables

A mapping approach must bind both variables and blank node identifiers, therefore I found it convenient to have a term for either a variable or a blank node identifier. For this, I propose “bindable”.

1.2.2 Mappings

In this paper I use “mapping” to mean a partial function from bindables to the scoping set. This is of course a broader definition than [SPARQL CR]. I use the term “variable mapping” for a partial function from variables to the scoping set, which is the same as the notion of “mapping” in [SPARQL CR].

1.2.3 Trees

Given a SPARQL query, one can represent the query as a tree, with interior nodes to represent the fundamental operations of SPARQL (triple pattern, conjunction, FILTER, OPTIONAL, UNION, and GRAPH). Constructing such tree representations from expression syntax is routine. I have chosen a particular tree representation, but the details could be changed readily.

With the mapping semantic proposed here, FilteredBasicGraphPattern loses its significance as the scope of a blank node identifier — deliberately so, as stated in the motivation. A FilteredBasicGraphPattern is logically equivalent to a conjunction of triple patterns and constraints. Hence there is no node type for FilteredBasicGraphPattern (or basic graph pattern). Also, there is no distinction between a conjunction expressed using a group graph pattern and the conjunction implicit in a FilteredBasicGraphPattern — a lesser objective mentioned in the motivation.

I have not fleshed out the details of the tree beneath a FILTER node; this task should be straightforward. The challenging part is UNION, OPTIONAL and GRAPH nodes.

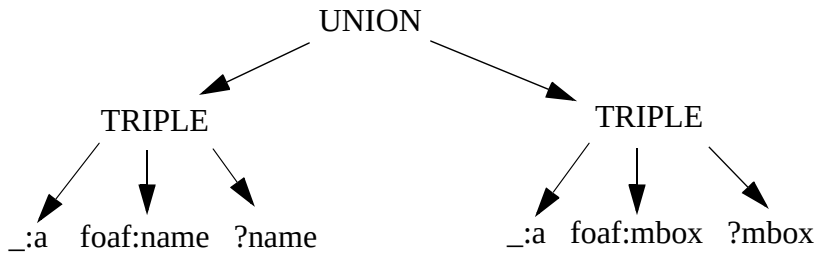
1.2.4 Paths

After converting a SPARQL query to a tree representation, I define the notion of a path through the tree. Conceptually, a path represents a subset of the tree that might be used in forming a particular solution or set of solutions.

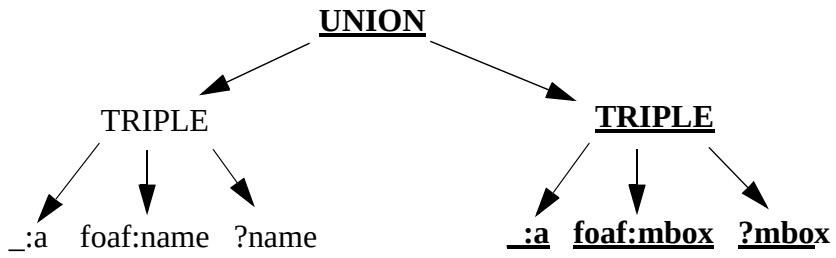
For example, consider a query involving a UNION:

```
SELECT ?name, ?mbox
WHERE {         { _:a foaf:name ?name }
        UNION   { _:a foaf:mbox ?mbox } }
```

Graphically, the WHERE clause might be represented



A solution only needs to satisfy one of the two triple patterns. Thus there are two paths through this graph that might be taken to generate a solution. One such path is shown by highlighting the nodes along one path in bold underlined in the following diagram:



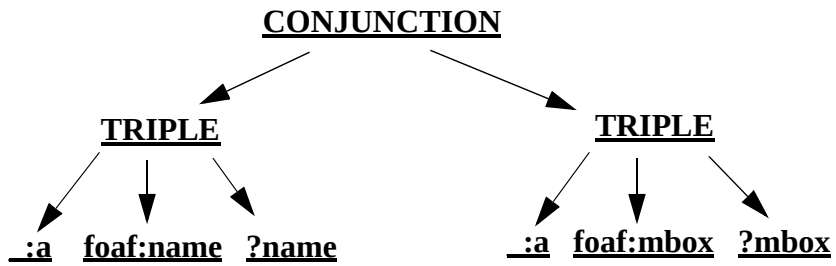
Of course there is also a path that passes through the left-hand child of UNION rather than the right-hand child.

I say that the nodes along a path are *activated* by that path.

Beneath a CONJUNCTION (used to represent a group graph pattern, as well as FilteredBasicGraphPatterns with more than one triple or FILTER), all children of an activated node must be activated. For example, starting with the query

```
SELECT ?name, ?mbox
WHERE { { _:a foaf:name ?name }
        { _:a foaf:mbox ?mbox } }
```

I derive this graph



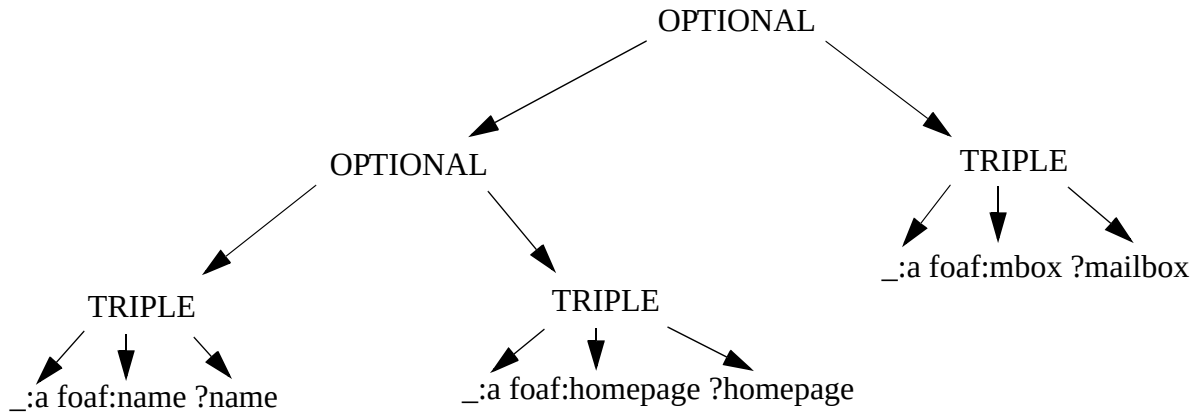
In this example, there is only one path, consisting of the nodes shown in bold underlined above.

OPTIONAL patterns require special treatment. If an OPTIONAL node is activated, then its left-hand child (the mandatory pattern) must also be activated, whereas its right-hand child (which I propose to call the supplementary pattern) may or may not be activated. Thus an OPTIONAL node implies two possible paths.

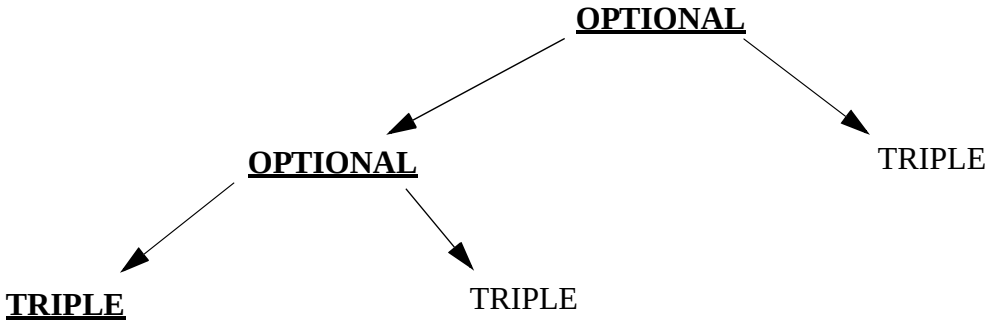
For example, the user's query in the introduction was

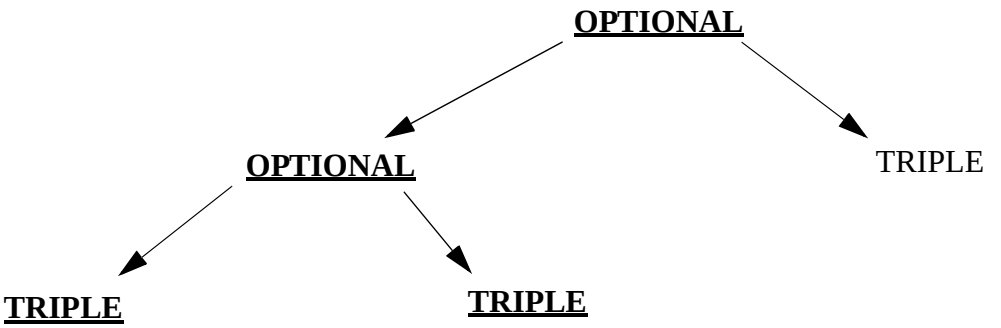
```
SELECT *
FROM G
WHERE { _:a foaf:name ?name .
        OPTIONAL { _:a foaf:homepage ?homepage }
        OPTIONAL { _:a foaf:mbox ?mailbox } }
```

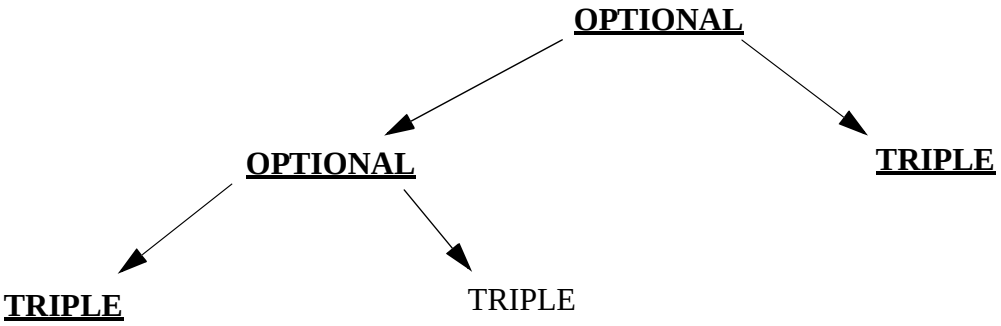
The WHERE clause is diagrammed (without highlighting any paths):

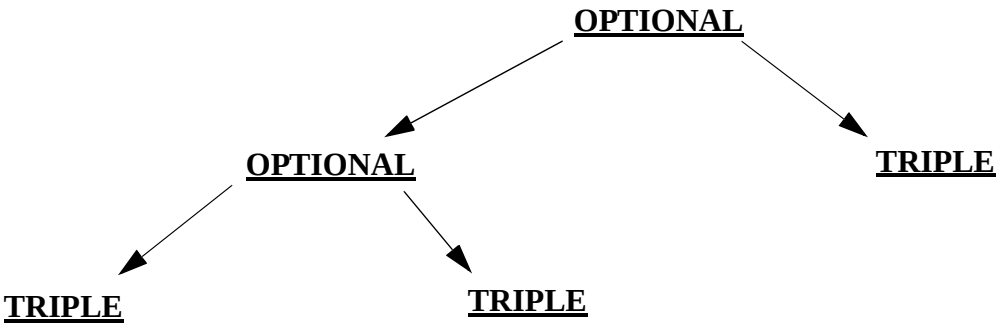


There are all together four paths through this tree, shown schematically below:









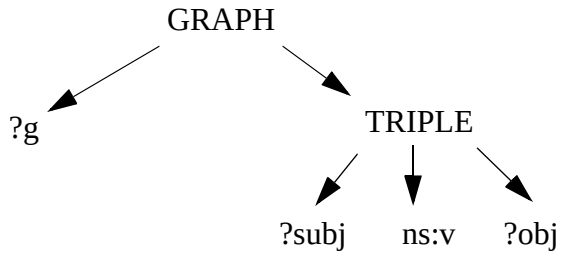
The reader may wonder why I adopted the technique of paths. Certainly the tree representation is routine and may be derived from the existing specification, and the path technique is also latent in the notions of UNION and OPTIONAL patterns, but do we need it as an explicit mechanism?

The answer is that I am trying to create a single unifed statement of the entire semantics of SPARQL. In [Constructive/destructive], I came to the conclusion that a bottom-up approach, which generates solutions from the leaves of the tree moving up the tree to the root, is probably

better than a destructive approach that starts at the root and tosses out solutions as it moves down the tree. However, in developing the current proposal, I considered the GRAPH pattern, such as

```
SELECT ?g, ?subj, ?obj
FROM NAMED ns:graph1
FROM NAMED ns:graph2
WHERE { ?g GRAPH { ?subj ns:v ?obj } }
```

which is diagrammed



One cannot start to solve this at the three right-hand leaves because a pattern must be solved in the context of a particular default graph. This requires a preliminary pass which comes down the graph, assigning the default graph to be used when solving each interior node. This first pass down the graph needs to decide which forks to activate as it encounters UNION or OPTIONAL nodes. (More precisely, the downward pass needs to generate all permissible activations of sub-trees.) Thus I arrived at the concept of a path through the tree.

Formally, a path is defined as a characteristic function on the tree, that is, a function whose codomain is {0, 1}, with 1 indicating that a node of the tree is activated and 0 indicating that it is not activated.

1.2.5 Constructions

This term is motivated by the idea that a construction builds a mapping on top of a path through a tree. A construction C on a path P is a pair of functions (CD, CM) consisting of:

1. The default graph construction, a function CD that maps the activated nodes of P to graphs in the dataset.
2. The mapping construction, a function CM that maps activated nodes of P to mappings.

Naturally there are a lot of conditions imposed on CD and CM, which make precise the requirements to solve any particular pattern along a particular path. These conditions are stated in a declarative form as a recursive definition. The idea is that the definition states how to recognize that C = (CD, CM) is a construction, without telling you how to actually obtain a construction.

In practice, the definition lends itself to a brute-force implementation in several passes, one to generate paths, another one to generate default graph constructions along each path, and finally a pass to generate mapping constructions along each path. Of course, actual implementations are free to improve upon the brute-force algorithm.

1.2.6 Restricted constructions

In [SPARQL CR] a solution only binds variables, so it is necessary to consider how to restrict a construction to V, the set of variables.

If C = (CD, CM) is a construction, then the restricted construction derived from C is CR = (CD, CMR), where for all activated nodes n, CMR(n) is CM(n) restricted to V.

Forming restricted constructions can also be viewed using equivalence relations. Two unrestricted constructions $C_1 = (CD_1, CM_1)$ and $C_2 = (CD_2, CM_2)$ are equivalent if they are constructed on the same path, $CD_1 = CD_2$, and for all activated nodes n, $CM_1(n)$ and $CM_2(n)$ are the same variable mapping when restricted to V. Thus restricted constructions are a way of factoring out the bindings of blank node identifiers.

1.2.7 Solutions

If CR = (CD, CMR) is a restricted construction, let r be the root node of the tree. CMR(r) is a variable mapping. CMR(r) is called a solution of the pattern on the dataset.

1.2.8 Solution sequences

Let $A = \{ CR_1, CR_2, \ldots \}$ be the set of all restricted constructions. For all i, let CMR_i be the mapping construction of CR_i. Let A be placed into an arbitrarily ordered sequence $AS = \{ CR_1, CR_2, \ldots \}$. Let r be the root node of the pattern tree. The sequence $SS = \{ CMR_1(r), CMR_2(r), ... \}$ is a solution sequence of the pattern.

1.2.9 Duplicate solutions

The definition factors out bindings to blank node identifiers when restricted construction are formed from constructions. This eliminates duplicates due to blank node identifier bindings, but otherwise it retains duplicates. (By duplicates I mean duplicates in the solution sequence prior to any projections.) I know of two ways that duplicates can arise:

1. From different selections of the default graph using a GRAPH pattern.
2. From different activated forks in the graph owing to UNION.

I give examples of each of these.

Duplicates from a GRAPH pattern: if the first operand of a GRAPH pattern (what I call the default graph selector) is a blank node identifier, then a duplicate solution might arise from more than one named graph. For example:

```
SELECT ?subj, ?obj
FROM NAMED ns:graph1
FROM NAMED ns:graph2
WHERE { _:g GRAPH { ?subj ns:v ?obj } }
```

In this example, it may happen that a particular variable mapping, say (?subj -> “Tom”, ?obj -> “football”), is a solution in both named graphs ns:graph1 and ns:graph2. In that case this variable mapping will arise once with a construction whose default graph chooses ns:graph1 and once with a construction whose default graph chooses ns:graph2.

Duplicates from a UNION: For example,

```
SELECT ?subj, ?obj
WHERE { { ?subj ns:v ?obj } UNION { ?subj ns:v ?obj } }
```

The tree for this query has two paths, which compute identical solutions. My definition of restricted constructions only factors out the bindings of blank node identifiers, it does not factor out the path. Therefore every solution will occur twice in the solution sequence.

2. Formal constructive mapping semantics

2.1 Preliminary terminology

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

A dataset consists of a distinguished graph (the default graph of the dataset) and zero or more named graphs, D = { G_0, (<u_1, G_1>), . . . , (<u_n, G_n>) }.

If G is a graph, let

I(G) be the set of all IRIs in G;

L(G) be the set of all literals in G;

B(G) be the set of all blank nodes in G;

T(G) = I(G) union L(G) union B(G) = set of all RDF terms in G.

Let V be the set of all variables (tokens that match rule [90] VARNAME), and let BNI be the set of all blank node identifiers (tokens that match rule [70] BLANK_NODE_LABEL, with any whitespace between '_:' and NCNAME removed). I regard BNI as distinct from $B(G_i)$, for all graphs G_i in the dataset.

It will be useful to have a term for something that is either a variable or a blank node identifier. I propose “bindable”. Let W be the set of bindables, W = (V union BNI).

A binding is an ordered pair (w, t), where w is a bindable and t is an RDF term. A binding associates a bindable to an RDF term.

A mapping is a function whose domain is a subset of W and whose codomain is the set of RDF terms. If S is a mapping, let dom(S) denote the domain of S.

Let M be the set of mappings.

Let e be the empty mapping, that is, the mapping whose domain is empty.

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), $S_1(v) = S_2(v)$.

2.2 Graph patterns

A graph pattern is defined as an ordered, labeled tree such that the following conditions are all true:

1. The labels on the leaves of a graph pattern are bindables, IRIs or literals.
2. The labels on interior nodes of a graph pattern are one of the keywords TRIPLE, CONJUNCTION, UNION, OPTIONAL, GRAPH, FILTER, and whatever additional node types are used to handle the details of FILTER.
3. If an interior node is labeled TRIPLE, then the node has three children, which are all leaves, called the subject, predicate and object of the TRIPLE. The predicate may not be a blank node identifier.
4. If an interior node is labeled CONJUNCTION or UNION, then all its child nodes are interior (ie, shall be labeled TRIPLE, CONJUNCTION, UNION, OPTIONAL, GRAPH or FILTER).
5. If an interior node is labeled OPTIONAL, then there are two child nodes, called the mandatory graph pattern and the supplementary graph pattern. These child nodes shall both be interior nodes.
6. If an interior node is labeled GRAPH, then there are two child nodes. The first child is called the graph selector, and may be either a variable or an IRI. The second child is called the graph graph pattern, and shall be an interior node.

If GP is a graph pattern, let Bindable(GP) be the set of bindables that are labels of leaf nodes of GP. Let Var(GP) be the set of variables that are labels of leaf nodes of GP.

2.3 Paths

If GP is a graph pattern, then a path through GP is a function P whose domain is GP and whose codomain is {0, 1}, with the following properties:

1. If r is the root node of GP, then P(r) = 1.
2. If n is a node of GP and P(n) = 0, then P(c) = 0 for all children c of n.

3. If n is a UNION node of GP, P(n) = 1, and c_1, ... c_m are the children of n, then for exactly one i, P(c_i) = 1 (i.e., P(c_j) = 0 for all j <> i).
4. If n is an OPTIONAL node, and P(n) = 1, then let m be the mandatory child and let s be the supplementary child. P(m) = 1 and P(s) may be either 0 or 1.
5. If n is any other kind of interior node and P(n) = 1, then P(c) =1 for all children c of n.

If P is a path through GP, then a node n in GP is said to be activated by path P if P(n) = 1, otherwise n is deactivated by P.

2.4 Constructions

Given a graph pattern GP, a dataset D = { G_0, (<u_1, G_1>), . . . , (<u_n, G_n>) }, and a path P, then a construction C is a pair of functions (CD, CM), such that CD assigns to each activated node n of GP a graph, and CM assignes to each activated node a mapping. CM and CD must satisfy the following properties:

1. CD is determined by walking the tree from the root to the leaves, as follows:

   a) If r is the root node, CD(r) = G_0.

   b) If n is an activated non-root node, let p be the parent of n.

      i) If p is a GRAPH node whose graph selector s is an IRI, then s is the name of a graph G_i (i.e., s = u_i for some i). Then CD(n) = G_i.

      ii) If p is a GRAPH node whose graph selector s is a bindable, then CM(p) shall bind s and CM(p)(s) is the name of a graph G_i. Then CD(n) = G_i.

      iii) Otherwise, CD(n) = CD(p).
2. For all activated nodes n, CM(n) is a mapping whose domain is a subset of Bindable(n).
3. For all non-root activated nodes n, let p be the parent of n; then CM(n) is a subset (= restriction) of CM(p).
4. CM is determined primarily by working from the leaves to the root, as follows:

   a) If n is an activated leaf node of GP and n is not bindable, then CM(n) is e, the empty mapping. (This condition is redundant; it is already implied by point 2 above.)

   b) If n is an activated leaf node of GP and n is bindable, then CM(n) is a mapping whose domain is n.

   c) If n is an activated TRIPLE node and CD(n) = G_i, then CM(n) is a mapping whose domain is Bindable(n). (This condition is redundant, following from point 4b.) Let s, p and o be the subject, predicate and object of n, respectively. Let S be the substitution defined by CM(n), i.e., S(t) = t if t is an RDF term,

S(t) = CD(n)(t) if t is a bindable. Then the triple (S(s), S(p), S(o)) is an element of G_i.

d) If n is an activated CONJUNCTION or GRAPH node, let c_1, ..., c_k be the child nodes of n. Then CM(n) = CM(c_1) union CM(c_2) ... union CM(c_k).

e) If n is an activated UNION node, let c be the activated child node of n (recall that a path must designate a single activated child of an activated UNION). Then CM(n) = CM(c).

f) If n is an activated OPTIONAL node, let m be the mandatory child and let s be the supplementary child. One of the following subcases applies:

i) s is activated and CM(n) = (CM(m) union CM(s)).

ii) s is not activated, CM(n) = CM(m), and there does not exist a path P_2 and a construction $C_2 = (CD_2, CM_2)$ along P_2 such that the following are all true:

1) s is activated by P_2

2) If p is a node that is not equal to or a descendent of s, then $P_2(p) = P(p)$ — i.e., P_2 restricted to the nodes that are not equal to or descendents of s activates the same nodes as P.

3) If p is a node that is not equal to or a descendent of s, and p is activated by P_2, then:

I) $CD_2(p) = CD(p)$

II) $CM_2(p) = CM(p)$

— i.e., CD_2 and CM_2 restricted to the activated nodes that are not equal to or descendents of s are the same as CD and CM, respectively, restricted to those nodes.

h) If n is a FILTER node, then the condition defined by the tree below the FILTER node is true when evaluated using CM(n) to substitute for binadables in the FILTER.

2.5 Restricted constructions

If C = (CD, CM) is a construction for graph pattern GP and dataset D, then the restricted construction derived from C is CR = (CD, CMR), where CMR is defined as follows: for all activated nodes n in GP, CMR(n) = CM(n) restricted to V.

2.6 Solutions

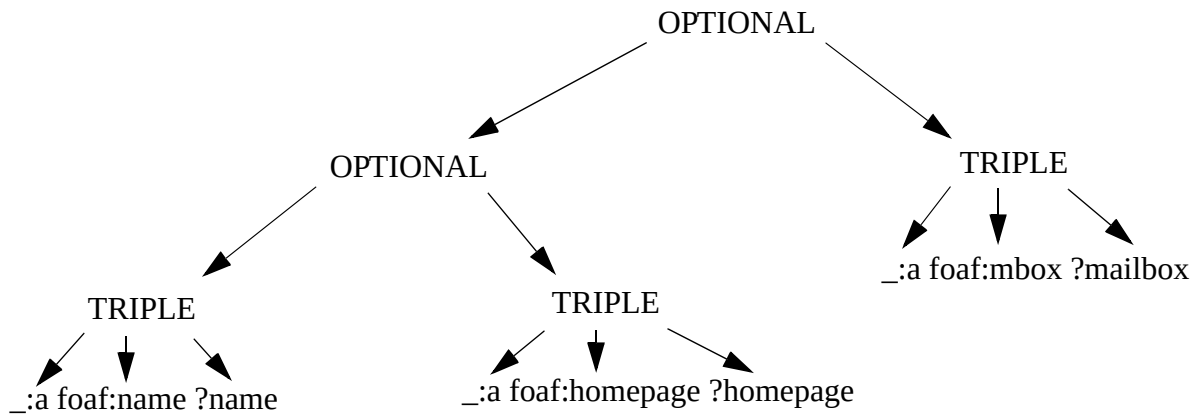
If CR = (CD, CMR) is a restricted construction, let r be the root node of the tree. CMR(r) is a variable mapping. CMR(r) is called a solution of the pattern on the dataset.

2.7 Solution sequences

Let A = { CR_1, CR_2, . . . } be the set of all restricted constructions. For all i, let CMR_i be the mapping construction of CR_i. Let A be placed into an arbitrarily ordered sequence AS = { CR_1, CR_2, . . . }. Let r be the root node of the tree. The sequence SS = { $CMR_1(r)$, $CMR_2(r)$, ... } is a solution sequence of the pattern.

3. Entailment reconsidered

After going through this exercise, I noticed that if there is no GRAPH node in the tree, then every path through the tree can be used to define a basic graph pattern. Let us go back to the example in the introduction. The tree is



and as explained earlier, there are four paths through this tree. Let us collect the triple patterns at the leaves of these four paths; this gives the following four basic graph patterns:

```
1. { _:a foaf:name ?name }

2. { _:a foaf:name ?name .
     _:a foaf:homepage ?homepage }

3. { _:a foaf:name ?name .
     _:foaf:mbox ?mailbox }

4. { _:a foaf:name ?name .
     _:a foaf:homepage ?homepage .
```

```
      _:a foaf:mbox ?mailbox }
```

The query as a whole can be expressed with the following pseudocode using entailment:

```
/* path 1 */
select ?name
where { G entails { G union { _:a foaf:name ?name } }
        and not (there exists ?homepage)
           { G entails
               { G union { _:a foaf:name ?name .
                           _:a foaf:homepage ?homepage } }
           }
        and not (there exists ?mailbox)
           { G entails
               { G union { _:a foaf:name ?name .
                           _:a foaf:homepage ?mailbox } }
           }
      }

union /* path 2 */
select ?name, ?homepage
where { G entails { G union { _:a foaf:name ?name .
                              _:a foaf:homepage ?homepage }
                  }
        and not (there exists ?mailbox)
           { G entails
               { G union { _:a foaf:name ?name .
                           _:a foaf:homepage ?mailbox } }
           }
      }

union /* path 3 */
select ?name, ?mailbox
where { G entails { G union { _:a foaf:name ?name .
                              _:a foaf:mbox ?mailbox }
                  }
        and not (there exists ?homepage
           { G entails
               { G union { _:a foaf:name ?name .
                           _:a foaf:homepage ?homepage } }
           }
      }

union /* path 4 */
select ?name, ?homepage, ?mailbox
where { G entails { G union { _:a foaf:name ?name .
```

```
                                        _:a foaf:homepage ?homepage .
                                        _:a foaf:mbox ?mailbox }
                        }
        }
```

Thus, using the notion of path, it is possible to create pseudocode that uses entailment on basic graph patterns to explicate pattern matching of OPTIONAL, and still have the uniform treatment of blank nodes that the user desires. The same can clearly be done for group graph patterns and UNIONs.

The obstacle I see in this course is GRAPH patterns, which potentially change the default graph for their descendents. Consider this query:

```
SELECT ?g, ?obj
FROM ns:graph0
FROM NAMED ns:graph1
FROM NAMED ns:graph2
WHERE { _:a ns:v1 ?g . ?g GRAPH { _:a ns:v2 ?obj } }
```

Translating this into pseudocode with explicit entailment, it says

```
select ?g, ?obj
from ns:graph0
from named ns:graph1
from named ns:graph2
where { ns:graph0 entails { ns:graph0
                            union { _:a ns:v1 ?g } }
        and
        graph (?g) entails { graph (?g)
                             union { _:a ns:v2 ?obj } }
      }
```

And this is as far as I can go; I don't see a way to unify this into a single entailment, and thereby provide the uniform treatment of the blank node identifier _:a. It seems that the _:a contained in the GRAPH pattern is inherently in a different scope from the _:a in the outer triple pattern.

If we wish to retain the entailment-based semantics, or at least an option for it in the future, then I think it may be reasonable to specify a syntactic restriction that prohibits having the same blank node identifier inside a GRAPH pattern and outside the same GRAPH pattern.

- End of paper -