# On Querying OBO Ontologies Using a DAG Pattern Query Language<sup>\*</sup>

Amarnath Gupta and Simone Santini

San Diego Supercomputer Center University of California San Diego, La Jolla, CA 92093, USA {ssantini, gupta}@sdsc.edu

Abstract. The Open Biomedical Ontologies (OBO) is a consortium that serves as a repository of ontologies that are structured like directed acyclic graphs. In this paper we present a language DQL for querying a database of directed acyclic graphs. The query language has a comprehension style syntax and contains a pattern specification sub-language DPL. DPL can be viewed as an extension of tree-pattern query language like XPath. The language allows extraction of nodes, paths and subgraphs from DAGs, and permits construction of result structures by composing them. We show that using such a language on OBO ontologies (such as the gene ontology), we can express more complex and scientifically valuable queries.

## 1 Introduction

Query languages and query evaluation techniques for the retrieval and manipulation of graph-structured data have been investigated since the late 80s [1,2], through the era of object-oriented data models [3,4,5] up to the more recent general interest in semistructured data [6,7,8] and ontologies represented in RDF [9]. Graph-structured data appear naturally in many modern applications, especially in biological information systems [10], chemical structure analysis [11], and social network analysis. In these application domains, a surprisingly large fragment of graph-structured data turn out to be directed and acyclic. Specifically in the domain of biomedical and biological ontologies, the majority of the ontological structures are designed to be directed acyclic graphs (DAGs). The Open Biomedical Ontologies (http://obo.sourceforge.net/) is an umbrella consortium that serves as a repository of many different but often inter-related ontologies, where the nodes of the graphs represent terms used in the vocabulary of a specialized biological domain, and the edges between nodes are typically labelled by the strings "isa", "part-of" or "develops-from". Furthermore, given the multiplicity and categories of ontologies emerging today, new needs are developing to query across ontologies and composing ontologies together. As the ontologies grow and become more complex, searching through them will require a more complex query mechanism that natively operates on graphs, especially DAGs.

<sup>\*</sup> Supported in part by NSF ITR Grant EIA-0205061, and the NLADR grant from NSF.

U. Leser, F. Naumann, and B. Eckman (Eds.): DILS 2006, LNBI 4075, pp. 152-167, 2006.



**Fig. 1.** A simplified fragment of the Biological Process component of Gene Ontology. The names of the nodes have been abbreviated for clarity.

Despite this need, most of the systems available to life scientists are mostly operated with visual interfaces allow only simple operations like keyword based node search, descendant enumeration, shortest path finding and neighborhood operations on graphs. This paper is an early step toward searching repositories of large ontological structures using a DAG query language, and similar in its intent as [12].

Example 1. As a motivational example, consider the well known Gene Ontology (GO) (www.geneontology.org) that consists of three DAG-structured components called biological processes (BP), molecular functions (MF) and subcellular components (SC). In Figure 1, a fragment of the BP DAG is shown. Here, an edge represents an superclass relation, such that  $n_1 \rightarrow n_2$  means that the process  $n_2$  is a specialization of the process  $n_1$ . Nodes in this graph represent tuples of a relation N which, in our simplified example, has three attributes *id*, name and *definition*. To make the node names simpler, just consider that a node with the substring "\_met" is a metabolism process, a node with "\_cat" is a catabolism process and a node with "\_biosyn" is a biosynthesis process. Given this example DAG, a number of different types of queries can be asked:

- 1. Which biosynthesis processes under lipid biosynthesis are also classified as amine biosynthesis? (Q1)
- 2. How does phosphatidylethanolamine biosynthesis (phos\_biosyn in Fig. 1) derive from cellular metabolism (cell\_met)? (Q2)
- 3. Is there a case where a xenobiotic process (e.g., xen\_met) is a subprocess of at least two forms of cellular metabolism? (Q3)
- 4. construct a reduced data graph by deleting all metabolism nodes except met, and connecting the non-deleted parent(s) of a deleted node n to its non-deleted children. (Q4)

Consider the first query. Since the graph represents a classification structure (i.e., an is-a graph) we interpret the expression "A classified as B" to mean "A reachable from B" in this DAG. Thus, this query can be expressed as the pattern query

 $\begin{array}{l} \mathsf{reachable\_from}(X, \mathsf{lipid\_biosyn}) \land \mathsf{reachable\_from}(X, \mathsf{amin\_biosyn}) \land \mathsf{substr}(\texttt{`biosyn'}, X) \\ & (\mathsf{Q1'}) \end{array}$ 

```
root
    conditional phenotypes (cp)
    cell cycle defects (ccd)
    mating and sporulation defects (msd)
        mating efficiency (me)
        sporulation efficiency (se)
        inappropriate sporulation (is)
            KAR4
            RIM1 ***
            ABP1
             . . .
        other mating and sporulation defects (omsd)
    . . .
    cell morphology and organelle nutrients (cmon)
        flocculence (fl)
        budding mutants (bm)
            bud localization (bl)
            multibudded cells (mbc)
            pseudohyphae formation (phf)
                 GDH3
                 TEC1
                 RIM1 ***
             . . .
        . . .
    stress response defects (srd)
    . . .
```

**Fig. 2.** A fragment of the Yeast Phenotype Classification. Some genes (leaf level) like RIM1 have multiple parents. The edge from the parent term of a gene to the gene is "produced-by-mutating".

where the last predicate is a syntactic way to state that X is a biosynthesis process; the query would return the set of two nodes (*phos\_biosyn*, *pNm\_biosyn*). Notice that since X must be reachable from lipid\_biosyn as well as amin\_biosyn, the query expresses a DAG-pattern, notionally akin to tree pattern queries expressed by XPath queries. Of course, our data model is much simpler than XML models, in that we only consider child and descendant relationships and completely disregard order among the children of a node.

*Example 2.* Next consider the yeast phenotype classification (YPC) scheme (available at http://mips.gsf.de/genre/proj/yeast/searchCatalogFirst Action.do?db=CYGD) – the non-leaf nodes of this scheme represent phenotype terms, while the leaf nodes represent genes. Since one gene can be responsible for multiple phenotypes, the structure of the YPC (Figure 2) is a DAG when leaves are considered (such as the gene RIM1 in the figure) and is an is-a tree over the rest of the nodes. For the purpose of this discussion, we will ignore all the descriptors associated with the genes and the YPC terms – we will only use

the fact that almost all yeast genes have references to GO-ids. Hence, the YPC structure is joinable with the BP ontology through the GO-ids. This enables us to ask scientific queries like: "merge all paths  $P_1$  reachable from the node named *tanscription*, with all paths  $P_2$  reachable from the yeast phenotype *sporulation defect* such that X is a node in  $P_1$ , Y is a node in  $P_2$  and X.id = Y.GOid by creating an edge labeled e from X to Y". Biologically, query creates an association between biological processes and relevant phenotypes. Computationally, the query first retrieves  $P_1$  and  $P_2$  through a join query and then merges them through a construction.

The intent of this paper is to present a query language and its corresponding algebra for the retrieval and manipulation of DAG-structured data to achieve the capabilities described above. The query language will have a sublanguage to express query patterns, a formal way to manage collections of intermediate result graphs, and operations to manipulate and construct graphs.

## 2 The DAG Data Model

As mentioned, the DAGs we consider have nodes that represent tuples from some relation N. With no loss of generality, we can assume that the *id* attribute of nodes is globally unique, so that nodes are represented by their ids. We only consider DAGs with unlabeled edges; we also assume that the children of any node of the data DAG are unordered. Henceforth, unless explicitly mentioned, we use the term *graph* (correspondingly, sub-graph) to mean this class of DAGs and its substructures.

In this paper, our focus is to introduce a language to manipulate the structure of the graph (we use the term graph to refer to DAGs from this point on); the retrieval and manipulation of the node content is performed using standard relational algebra. To manipulate the graph structure, on the other hand, we need to have a type system with basic types and type constructors. For example, to define a set of paths, we need to have the type set(list(skolem)) where skolem is the data type of all ids, and is a data type for which no operations are defined except value equality. In our model all nodes are typed. If  $\alpha$  is the type of a node then we will use the notation  $\nu[\alpha]$  for a generic collection monoid of type  $\nu$  with elements of type  $\alpha$  (e.g., set(GO-node-type)). For this monoid three functions are defined:

- i)  $\operatorname{nil}_{\nu} : \nu(\alpha)$  (the empty collection);
- ii)  $\mathcal{U}_{\nu}: \alpha \to \nu(\alpha)$  (the singleton);
- iii)  $-\nu : \nu(\alpha) \times \nu(\alpha) \to \nu(\alpha)$  (the *join*, like the union operator for a set monoid).

We also define the projection operator  $p_i : \alpha_1 \times \cdots \times \alpha_n \to \alpha_i$  and the record construction operator  $(\_,\_,\_,\_): \alpha_1 \times \cdots \times \alpha_n \to \langle \alpha_1 \times \cdots \times \alpha_n \rangle$ . To introduce a few terms used through the paper, a node n of a graph g is terminal if it has no outgoing edges, and it is *initial* if it has no incoming edges;  $\bot(g)$  is the set of all terminal nodes of the graph g, and  $\top(g)$  is the set of initial nodes. Since our graphs are acyclic, for each g, neither  $\bot(g)$  nor  $\top(g)$  are empty.

## 3 Our Query Language DQL

We start by observing that the query language presented here assumes that the data is in the form of a DAG and not a graph containing cycles. While it is easy to show that the pattern language cannot express a cycle, we do not ask how the queries would behave if the underlying data had cycles.

#### 3.1 An Informal Introduction

We introduce the query language using the graph  $G_1$  shown in Figure 3 as the reference. First, we focus on the pattern language DP. The pattern (v = 1)matches a set of nodes for which the value of the variable v is 1. The pattern true matches all nodes of the DAG. We use the symbol - to denote an edge from the node to the left of - to the node to the right of it. Thus, the pattern (v=1)-(v=2) matches the edges  $[1,1] \rightarrow [3,2]$  and  $[2,1] \rightarrow [4,2]$ . DP allows the use of the Kleene star to refer to 0 or more occurrences of the subpattern within its scope. The pattern (v = 1)[-(v = 2)] \* -(v = 1) matches the graphs have a node with v = 1 is followed by a chain of any number of nodes with v = 2, which is then followed by another node with v = 1 (not the same node as the first since the graph is acyclic). The edge chains matching the pattern are  $[1,1] \rightarrow [3,2] \rightarrow [7,1], [1,1] \rightarrow [3,2] \rightarrow [2,1], [1,1] \rightarrow [3,2] \rightarrow [4,2] \rightarrow [8,1], and$ so on. Now, let us associate variables x and y to two elements of the pattern. The augmented pattern becomes y: (v = 1)[-(v = 2)] \* -x: (v = 1). Although here the variables are only associated with nodes, in general, variables can associated with any subpattern, such as an edge chain or a subgraph, as illustrated later in the paper. The variable association implicitly produces matches for the variables in addition to the match for the whole pattern. In this example, the pattern produces the y, x tuples  $\{([1,1], [2,1]), ([1,1], [7,1]), ([1,1], [8,1]), ([2,1], [8,1])\}$  if we eliminate duplicates.

As the final element in this section, we would like to produce a graph for each xy pair by constructing an edge from each instance of x to its corresponding y. This operation of graph creation requires us to produce a set of edges,



Fig. 3. Our running example. Each node has an id (the first number) and an attribute called v, whose value is shown in the second number.

which the pattern language cannot express because the pattern language only performs matching. To accomplish the graph construction, we place the pattern in a monoid comprehension framework, and express it as:

 $\cup [\{x - y | g \vdash y : (v = 1) [-(v = 2)] * -x : (v = 1) \leftarrow G_1 \}]$ 

which is read as: Let g be that substructure of  $G_1$  that satisfies the specified pattern  $\pi$ . Using the variables x and y of g construct the edge x - y for each instance x, y satisfying  $\pi$ , and form a set union of these edges.

#### 3.2 Formal Description of Pattern Language DP

To formalize the ideas described in the previous subsection, we observe that pattern  $\pi$  in the pattern language DP is generated by the following rules:

- i) A predicate C in which the free variables are the names of the components of the node data type is a pattern; in particular t (the value "true") is a pattern;
- ii) if  $\pi_1$  and  $\pi_2$  are patterns, then  $\pi_1 \pi_2$  is a pattern;
- iii) if  $\pi$  and  $\pi'$  are patterns, then  $\pi'[-\pi]*$  and  $[\pi-]*\pi'$  are patterns;
- iv) if  $\pi_1, \ldots, \pi_n$  are patterns, and  $\nu$  is a patterns then  $\{\pi_1, \ldots, \pi_n, -\}\nu$ , and  $\nu\{-\pi_1, \ldots, -\pi_n\}$  are patterns;
- **v)** if  $\pi_1, \ldots, \pi_n$  are patterns, then  $\pi_1 | \ldots | \pi_n$  is a pattern;
- vi) if  $\pi$  is a pattern and v a variable name, then  $v : \pi$  is a pattern;
- vii) nothing else is a pattern.

These cases are illustrated in Figure 4.

The grammar of the language is

$$\begin{aligned} <\pi> ::= C | <\pi> - <\pi> \\ | <\pi>[-<\pi>]* \\ | [<\pi>-]*<\pi> \\ | <\pi>'\{'-<\pi>\{,-<\pi>\}^{*'}\}' \\ | `\{'<\pi>-\{,<\pi>-\}^{*'}\}'<\pi> \\ | <\pi>|<\pi>|<\pi>\{|<\pi>\}^{*} \\ | (<\pi>)|:<\pi> \end{aligned}$$

Note that the brackets { and } have been placed in quotes when they appear as terminals to avoid confusion with the repetition operator of the grammar. Condition have higher precedence that the structural operators, and - has precedence over |. Parentheses can be used whenever necessary. We use the shortcut  $\# \equiv [-t] * -$  (or, equivalently, -[t-]\*), making the symbol # the notional equivalent of // in XPath.

It is important to point out a few distinctive aspects of this DAG pattern language.

In the informal example, we stated that - represents an edge between two *nodes*. In this section, we generalize this notion to represent a "connection" between two subDAGs, one satisfying pattern  $\pi_1$  and another satisfying pattern  $\pi_2$ 



Fig. 4. Examples DAGs corresponding to the different patterns described in the text. (a)  $\pi_1 - \pi_2$ , (b)  $\pi_1[-\pi_2](2,4)$ , (c)  $\tau\{-\pi_1, -\pi_2\}$  and (d)  $\{\pi_1 -, \pi_2 - \nu\}$ .

(construction rule ii above). To this end, we define a *stitch* relationship ( $|\rangle$ ), which generalizes the *child* relationship for tree-structured data. given two graphs  $g_1$ and  $g_2$ , let  $g_1|\rangle g_2$  be the graph obtained by connecting all terminal nodes of  $g_1$ to all initial nodes of  $g_2$ . Thus the semantics of rule ii is that if combined pattern  $\pi \equiv \pi_1 - \pi_2$ , and the graph g matches  $\pi$  then there are two disjoint sub-graphs of g, namely  $g_1$  and  $g_2$  such that  $g_1$  matches  $\pi_1$ ,  $g_2$  matches  $\pi_2$ , and  $g = g_1|\rangle g_2$ .

Next, we use the  $|\rangle$  operation to generalize the twig structure of tree pattern languages. For DAGs we need both the *split* structure of trees denoting branches emanating from a node, as well as a *merge* structure denoting edges converging on to a node. In this vein, the expression within the  $\{...\}$  (rule iv) is a branching where the patterns on the different branches are required to be distinct. Thus the whole pattern represents a fork  $(\tau\{...\})$  or merge  $(\{...\}\nu)$  pattern, or a combination. Formally, if  $\pi \equiv \tau - \{\pi_1 -, \ldots, \pi_n - \}\nu$ , then g matches  $\pi$  if there are disjoint sub-graphs of g g',  $g_1, \ldots, g_n, g''$  such that g' matches  $\tau$ , for each i  $g_i$ matches  $\pi_i, g''$  matches  $\nu$ , and, for all  $i g'|\rangle g_i$  and  $g_i|\rangle g''$  are sub-graphs of g.

Finally, the language offers a syntax to specify the number of recurrences of a

pattern. We use the shortcuts  $\pi^n \equiv \overline{\pi - \pi - \cdots - \pi}$ ,  $\pi'[-\pi](m, n) \equiv \pi' - \pi^m | \pi' - \pi^{m+1} | \cdots | \pi' - \pi^n$  (n > m), the shortcut  $[\pi -](m, n)\pi'$  is defined analogously),  $\pi'[-\pi](m, \infty) \equiv \pi' - \pi^m[\pi]$ \* (the shortcut  $[\pi -](m, \infty)\pi'$  is defined analogously.

#### 3.3 The DQL Language

The use of monoids and the comprehension syntax is common in query languages that allow complex types [13]. For example, the query Q1 presented in Example 1 can now be expressed as:

 $\cup [\{x|g \vdash \{substr(name, "lipid_biosyn"), substr(name, "amin_biosyn")\} \# x : substr(name, "biosyn") \leftarrow GO\}]$ 

Q2 and Q3 can be expressed with queries having a similar form. For example, Q3 can be expressed as:

$$\cup [\{z | g \vdash (name = "cell\_met") \# \{x, y\} \# z, (name = "xen\_met") \# z \leftarrow GO\}]$$

This is a conjunctive query the result variable z must satisfy two patterns. Note the x and y are implicitly existentially qualified, and by the semantics of the  $\{\ldots\}$ construct the same node cannot instantiate both variables. Query Q2 (and all queries that ask "what is the relationship between nodes satisfying condition  $C_1$ and those satisfying condition  $C_2$ ?") is an example of a graph-returning query. It is simply formulated as:

$$\cup [\{x | g \vdash x : ((name = "cell\_met") \# (name = "xen\_met")) \leftarrow GO\}]$$

where the scope for the result variable x is the entire subgraph satisfying the given pattern.

Now we turn our attention from the pattern language DP to the monoid comprehension structure in which it is embedded. Since most of the monoids we need are standard for sets, lists, and arithmetic and if-then-else constructs, we do not describe them in detail here. However, in addition to the collection and the simple monoids of the comprehension calculus, graphs come with their own monoids, each one defined on the set of graphs, and characterized by a join operation. The most important are:

merge: puts together two graphs by identifying nodes with the same id;

**gmax:** given two graphs  $g_1, g_2, g = \text{gmax}(g_1, g_2)$  is the smallest graph for which  $g_1, g_2 \subseteq g$ ;

gmin: the largest graph contained in two graphs.

All these operators can be easily extended to take a set of graphs as input. We omit the proof that these operations are associative, as required by the definition of monoid. In the previous example, the query

$$merge\{x - y | g \vdash y : (v = 1)[-(v = 2)] * -x : (v = 1)\}$$
(2)

would return as a result the graph

$$\begin{bmatrix} [8,1] \\ * \\ [2,1] \end{bmatrix} \begin{bmatrix} [7,1] \\ * \\ [1,1] \end{bmatrix} \begin{bmatrix} [2,1] \\ [1,1] \end{bmatrix}$$
 (3)

As we have seen in some examples above, any graph that matches a fragment of the pattern can be assigned to a variable. This gives us the possibility of assembling a result out of portions of the graphs in the data base. Consider the constructive query presented as Q4 in Section 1. Abstractly, the query can be modeled as: Given three conditions on nodes A (nodes with attribute name containing "\_met"), B, C (these conditions are empty in example Q4), remove from the graph all the nodes which satisfy condition A; every time one of these nodes has a parent that satisfies B and a child that satisfies C, join the parent and the child.

Consider first this query:

$$merge\{x - y | g \vdash x : ([t-] * B) - A - y : (C[-t]*)\}$$
(4)

Here x represents the subgraph "up to" the *B*-satisfying nodes and y represents the subgraph beyond the *C*-satisfying nodes. The edge-construction between these nodes effectively deletes the nodes satisfying *A* from the output. This query performs the required job for the portion of the graph that contains nodes that satisfy the conditions on *A*, *B*, *C*. However, other portions of the graph do not match any pattern, and hence, will be lost. The solution in this case is to use the negation of *A* to match all paths that do not contain *A*, and then merge the graphs thus obtained:

$$merge(merge\{x - y|g \vdash x : ([t-] * B) - A - y : (C[-t]*)\}, merge\{z|g \vdash z : ([\neg A -] * \neg A)\})$$
(5)

## 4 Translating into an Algebra

The algebra in which the patterns are translated can be divided in two parts: on one hand there are the operations that deal with the values of each nodes, on the other there are the *structural* operations that manipulate the graph structure. The first part is fairly standard (e.g., textbook operations for relational systems, [13] for object-valued data, [3] for tree-valued data and so on). In this section, we will concentrate mainly on the second. The graph operations for a graph with nodes of type  $\alpha$  work on three data types: the data type of the nodes themselves ( $\alpha$ ), that of paths of nodes (equivalent to lists of nodes, i.e. [ $\alpha$ ]), and that of graphs ( $\Gamma(\alpha)$ ), with the sub-typing relations  $\alpha < [\alpha] < \Gamma(\alpha)$ .

There are three graph manipulation operators in the algebra:

**path:** the call  $path(g, n_1, n_2, h, k)$  return the set of paths between the nodes  $n_1$  and  $n_2$  in the graph g such that the length of the path is between h and k; the typing of this function is

$$\frac{g:\Gamma(\alpha) \quad n_1, n_2:\alpha \quad h, k: \text{int}}{\text{path}(g, n_1, n_2, h, k): \{[\alpha]\}}$$
(6)

**merge:** the call  $merge(g_1, g_2)$  merges the two graphs  $g_1$  and  $g_2$  by identifying the nodes with equal value; the operator requires that the two graphs have at least one node that can be identified: it returns null for disconnected graphs; its typing is

$$\frac{g_1, g_2 : \Gamma(\alpha)}{\operatorname{merge}(g_1, g_2) : \Gamma(\alpha)}$$
(7)

 $\sigma$ : the call  $\sigma(g, P)$  returns the set of all nodes of the graph g that satisfy the predicate P; its typing is

$$\frac{g:\Gamma(\alpha) \quad P:\alpha \to \mathbf{2}}{\sigma(g,P):\{\alpha\}} \tag{8}$$

The proof of the following property is quite obvious, and we don't report it here:

**Theorem 1.** The algebra (path, merge,  $\sigma$ ) is minimal: none of its operators can be expressed as a combination of the others.

In addition to the graph operators there are two structural operators: *apply* and *chain*.

**apply:** The operator  $\operatorname{apply}[\omega](A, f)$  applies the function f to all the elements of the structure A, and collects the results in a structure of type  $\omega$ . It typing is:

$$\frac{A:\nu(\alpha) \quad f:\alpha \to \beta \cup \{\bot\}}{\operatorname{apply}[\omega](A,f):\omega(\beta)} \tag{9}$$

Formally, define the modified singleton for  $\omega$  as

$$s'_{\omega}(x) = \begin{cases} s_{\omega}(x) \text{ if } x \neq \bot \\ 0_{\omega} \text{ if } x = \bot \end{cases}$$
(10)

then, if  $A = a_1 \underbrace{\nu} \cdots \underbrace{\nu} a_n$  one has

$$\operatorname{apply}[\omega](A, f) = s'_{\omega}(f(a_1)) \underbrace{\basel{eq:apply}{\baselineskip} \cdots \baselineskip}{\baselineskip} s'_{\omega}(f(a_n)) \tag{11}$$

**chain:** given a set of paths S, a graph g that contains them, and two integers h, k, chain(g, S, h, k) builds all the chains that can be built out of paths in S taking each path between h and k times. Its typing is:

$$\frac{S:\{[\alpha]\} \quad g:\Gamma(\alpha) \quad h,k:\text{int}}{\operatorname{chain}[\omega](g,S,h,k):\{[\alpha]\}}$$
(12)

Consider now a pattern  $\pi$  for which a translation is sought in the previous algebra. Formally, the planning algorithm is a function  $plan(\pi, g, U)$  where  $\pi$  is the pattern for which a plan is sought, g is the variable name for the input graph, and U is the variable name for the set of *environments* which is the collection of instantiated pattern variables produced at any stage of the plan. The value of the function plan is a list of algebra functions and variable assignments. We give a couple of simple examples of plans, before going into the details of the algorithm. Here, as elsewhere,  $u_1, u_2, \ldots$ , and  $p_{11}, p_{12}, \ldots, p_{ij}, \ldots$  are unique variable names generated by the planning algorithm.

$$plan(z : C, g, e) =$$

$$u_1 = \sigma(g, C);$$

$$e = apply[set](u_1, fun x \Rightarrow (z \mapsto x))$$

Note that we write a list of (in this case) two elements as a;b rather than [a,b] for ease of notation, and that we use the ML-style notation "fun  $x \Rightarrow v$ " for  $\lambda x.v$ . The value " $(z \mapsto x)$ " is the environment constructor: it creates an environment in which the only assignment is that of the value x to the variable z.

$$\begin{aligned} \operatorname{plan}(z:C_1-C_2,\operatorname{g},\operatorname{e}) &= \\ u_1 &= \sigma(\operatorname{g},C_1); \\ u_2 &= \sigma(\operatorname{g},C_2); \\ p_{12} &= \operatorname{apply}[\operatorname{set}](u_1 \\ & \quad \operatorname{fun} x_1 \Rightarrow \operatorname{apply}[\operatorname{set}](u_2, \\ & \quad \operatorname{fun} x_2 \Rightarrow \operatorname{path}(x_1,x_2); \\ & \quad (1, x_2); \\ e &= \operatorname{apply}[\operatorname{set}](u_1, \\ & \quad \operatorname{fun} x_3 \Rightarrow (\operatorname{z} \mapsto x_3) \\ & ) \end{aligned}$$

We illustrate the algorithm through an example. Consider the pattern

$$y: (C_1[-t] * C_2[-t](5,7) - x: (C_3[-C_4 - C_5] * -C_6) - C_7)$$
(13)

where  $C_1, \ldots, C_7$  are suitable conditions on the nodes and t stands for the value *true*. The first rewriting consists in isolating the portions that are assigned to a variable (except for the variable that contains the whole pattern; this is necessary because, in the final algorithm we will have to create not only the sub-graphs that match the whole pattern, but also the sub-graphs that match the individual variables). We represent this rewritten pattern as follows:

Then we replace all the patterns with [-t] or [t-] with the path symbols #, -, or (a, b), which indicates a path of length between a and b:

Then we expand the "star" elements:

The planning algorithm operates on this representation.

First, each repeated pattern is eliminated: For each pattern  $[-\pi](n,m)$ , the planning algorithm is called recursively to generate a plan for  $x : \pi$ , where x is a new variable, and then the function chin is used to generate the set of structures that match the repeated pattern. In other words, we have, for a path  $[-\pi](n,m)$ , the fragment

$$plan(x_1 : \pi, g, u_1); u_2 = apply[set](u_1 fun x_2 \Rightarrow u_1(x_2) (Transform the set of environments into ); a set of graphs) p_{45} = chain(g, u_2, n, m);$$

In the representation, the star operator is replaced by the name of the path set that contains the graphs that satisfy the pattern:

Note that for the purpose of the paper, the name  $p_{45}$  to the variable that holds the path has been given for ease of exposition, since these are paths that go from nodes for which condition  $C_4$  holds to nodes for which condition  $C_5$ holds. The same convention will be followed in the paper for all variable names; the actual names used by the algorithm may, of course, vary.

Now the instructions are generated to replace each condition  $C_i$  with the set of nodes that satisfy it

$$U_1 = \sigma(g, C_1)$$
  
$$\vdots$$
  
$$U_7 = \sigma(g, C_7)$$

and the sets of paths that join contiguous nodes in the traversal of the structure are generated, with the conditions established for that path:

$$\begin{array}{l} p_{12} = \operatorname{apply}[\operatorname{set}](U_1, \operatorname{fun} x \Rightarrow \operatorname{apply}[\operatorname{set}](U_2, \operatorname{fun} y \Rightarrow \operatorname{path}(x, y, 0, \operatorname{infty})) \\ p_{23} = \operatorname{apply}[\operatorname{set}](U_2, \operatorname{fun} x \Rightarrow \operatorname{apply}[\operatorname{set}](U_3, \operatorname{fun} y \Rightarrow \operatorname{path}(x, y, 5, 7)) \\ p_{34} = \operatorname{apply}[\operatorname{set}](U_3, \operatorname{fun} x \Rightarrow \operatorname{apply}[\operatorname{set}](U_4, \operatorname{fun} y \Rightarrow \operatorname{path}(x, y, 1, 1)) \\ p_{56} = \operatorname{apply}[\operatorname{set}](U_5, \operatorname{fun} x \Rightarrow \operatorname{apply}[\operatorname{set}](U_6, \operatorname{fun} y \Rightarrow \operatorname{path}(x, y, 1, 1)) \\ p_{67} = \operatorname{apply}[\operatorname{set}](U_6, \operatorname{fun} x \Rightarrow \operatorname{apply}[\operatorname{set}](U_7, \operatorname{fun} y \Rightarrow \operatorname{path}(x, y, 1, 1)) \end{array}$$

The data structure is updated by eliminating the node sets and replacing each path symbol with the set of paths that implement it:

The next step of the algorithm is a traversal of this structure where, for each  $p_{ij}$  a loop is generated to chain (denoted by the symbol  $\sim$ ) every path in it with the paths of the following set  $p_{jk}$ . In addition, the paths that depend on a variable are joined separately, and environments are created in which the paths are assigned to the variable. The path corresponding to each variable is expanded, going from the variables deeper in the structure towards the top. In this case there is only one variable, so there will be a single expansion:

```
 \begin{array}{l} p_{36} = \operatorname{apply[set]}(p_{34}, \operatorname{fun} x_{34} => \\ & \operatorname{apply[set]}(p_{45}, \operatorname{fun} x_{45} => \\ & \operatorname{apply[set]}(p_{56}, \operatorname{fun} x_{56} => \operatorname{merge}(x_{34}, \operatorname{merge}(x_{45}, x_{56}))) \\ & ) \\ ) \end{array}
```

The structure is then updated as follows:

$$p_{12} \sim p_{23} \sim p_{36} \sim p_{67} \tag{19}$$

and an entry is made in a variable table to associate the variable x with the set  $p_{36}$ . The operation is repeated until the complete structure has been eliminated. In this case there will be only one more generation:

```
 \begin{array}{l} p_{17} = \operatorname{apply[set]}(p_{12}, \operatorname{fun} x_{12} => \\ & \operatorname{apply[set]}(p_{23}, \operatorname{fun} x_{23} => \\ & \operatorname{apply[set]}(p_{36}, \operatorname{fun} x_{36} => \\ & \operatorname{apply[set]}(p_{67}, \operatorname{fun} x_{67} => \operatorname{merge}(x_{12}, \operatorname{merge}(x_{23}, \operatorname{merge}(x_{36}, x_{67})))) \\ & ) \\ & ) \\ \end{pmatrix}
```

Now all the structures that are necessary to contain the result are contained in the p variables: the final step is the construction of the set of environments; the apply functions loop over all the structure sets associated to output variables x and y:

$$U = \operatorname{apply[set]}(p_{17}, \operatorname{fun} x_{17} \Rightarrow)$$
  

$$\operatorname{apply[set]}(p_{36}, \operatorname{fun} x_{36} \Rightarrow (\mathbf{x} \mapsto x_{36}) \oplus (\mathbf{y} \mapsto x_{17})$$
  
);

The  $\oplus$  operator creates the tuples of all x, y pairs that satisfy the plan.

The fundamental correctness result for the algorithm is the following:

**Theorem 2.** Let  $\pi$  be a pattern in  $DP^{-v}$  the variable-free fragment of DP, g a graph, and U the set of environments created by the execution of the plan  $plan(x : \pi, g, U)$ , with  $U = \{(x \mapsto q_i)\}$  then:

i)  $q_i \subseteq g;$ ii)  $q_i \models \pi.$ 

The proof, not formally presented here, is conceptually very simple: it is based on the fact that all the paths that are generated are between nodes that satisfy the corresponding end-path conditions and therefore each path corresponds to a fragment of the pattern. The semantics of the chain operator guarantees that this is true for repeated patterns as well. The way in which the sub-patterns for variables are expanded guarantees that at the end of the plan each graph that has to be assigned to a variable is present in one of the  $p_{ij}$ .

## 5 Applying DQL to Life Science Problems

We are in the process of constructing a composite ontology for disease specific information by combining relevant substructures from multiple different ontologies and other standard databases. A full description of this on-going work is beyond the scope and page limit of this paper. Here we present a few illustrations of how the features of DQL are used in the task.

ICD-10 (http://www3.who.int/icd/vol1htm2003/fr-icd.htm) is a taxonomy that categorizes diseases based on the system (e.g., cardiopulmonary) they affect. The pathway ontology (http://cvs.sourceforge.net/viewcvs. py/obo/obo/ontology/genomic-proteomic/pathway.obo) relates certain diseases with the molecular pathways they affect. The "biological processes" fragment of the Gene Ontology relates major pathways to component pathways that constitute the major pathways. These component pathways often formally refer to the molecular elements or biological processes that participate in them. Finally, genes are functionally annotated by GO-ids to terms from the gene ontology. Thus it is notionally possible to start with a family of diseases per the ICD-10 classification, ultimately relate them to the biological processes and corresponding genes. In our preliminary experiments to construct such connections, we have successfully created integrated graphs for closely related neurodegenerative disorders (like Alzheimer's disease, Parkinson's disease, Lewy body disease etc.) and identified subgraphs that are common to these diseases. In performing these exercises, we have identified a number of "query patterns" that are very convenient to express with the DQL:

- "Find node  $n_1$ 's reachability graph in  $G_1$  until some node  $n_2$  such that  $n_2$  can be joined with some descendant of  $n_3$  of graph  $G_2$ .
- "Find that subgraph of the  $n_1$ 's reachability graph that reaches  $n_2$  but not any  $n_3$  that is reachable from both  $n_4$  and  $n_5$ .
- "Merge two subgraphs found by subqueries  $S_1$  and  $S_2$  such that the merged nodes refer to the same GO-id or UMLS id". UMLS is a large vocabulary from the National Library of Medicine.

## 6 Related Work

Querying ontologies as graphs is a relatively new area of research. [12] has developed an algorithm to index DAG-structured data to make queries like transitive closure and least common ancestor more efficient. [14] has developed an algorithm to perform pattern matching queries on DAGs, and used in on the Gene Ontology. [15,16] have developed algorithms for DAG searching. However, to our knowledge this is the first attempt to develop a query language for DAG data, and apply it to address an emerging area of life sciences.

In terms of query languages, we mark distinction between DQL and schemabased graph query languages like [4,5] in that ours is a pattern language and does not operate in the paradigm of querying against a graph-schema. On the other hand, DQL is closely related to [13,17] on the one hand and XML query languages on the other. We view the primary contribution of this work in extending a monoid comprehension framework with a DAG-manipulating pattern language. We contrast our language with Lorel [6], UnQL [8] and StruQL [7] in two ways. 1) Our pattern sublanguage DP is specifically designed for DAGs (and not for general graphs) and although not shown here, can be proven to express serially connected minimal vertex series-parallel graphs (MVSPs) [18]. 2) Our language permits more powerful construction capabilities than these languages. Lorel does not have any graph restructuring operation, UnQL's graph construction operations are simpler than ours. StruQL is closer to our language; but StruQL was designed for web site construction and did not need nesting. DQL allows naturally allows nesting through environments, where at each level of nesting we can have selection, aggregation and construction.

## 7 Conclusion

In this paper, we have made the case that having the ability to query a repository of ontologies will provide a useful tool to enable new types of analysis that were not possible hitherto. To this end, we have presented the DQL query language and the DAG pattern definition sublanguage DP, a corresponding algebra, and a trace of the query planning process. In this paper, we have taken the narrow view that ontologies are merely DAGs and adopt a closed world assumption. The semantic aspect of ontologies that leads to knowledge representation and logical inference problems have been ignored. This allows us to focus on the formulation of structural queries. Even with structural queries alone, interesting life science problems can be addressed. We have not covered systems design and query evaluation algorithms in this paper.

## References

- 1. Consens, M.P.: Graphlog: Real life recursive queries using graphs. Master's thesis, Dept. of Computer Science, University of Toronto (1989)
- Agrawal, R., Jagadish, H.V.: Direct algorithms for computing the transitive closure of database relations. In: Proc. 13th Int. Conf. on VLDB. (1987) 255–266
- Subramanian, B., Zdonik, S.B., Leung, T.W., Vandenberg, S.L.: Ordered types in the aqua data model. In: Proc. of the 4th Int. Workshop on Database Programming Languages (DBPL), London, UK, Springer-Verlag (1994) 115–135
- Gyssens, M., Paredaens, J., den Bussche, J.V., van Gucht, D.: A graph-oriented object database model. IEEE Transactions on Knowledge and Data Engineering 6 (1994) 572–586
- Poulovassilis, A., Levene, M.: A nested-graph model for the representation and manipulation of complex objects. ACM Trans. Inf. Syst. 12 (1994) 35–68
- McHugh, J., Abiteboul, S., Goldman, R., Quass, D., Widom, J.: Lore: a database management system for semistructured data. SIGMOD Rec. 26 (1997) 54–66
- Fernandez, M.F., Florescu, D., Levy, A.Y., Suciu, D.: Declarative specification of web sites with strudel. VLDB Journal 9 (2000) 38–55
- Buneman, P., Fernandez, M., Suciu, D.: Unql: a query language and algebra for semistructured data based on structural recursion. The VLDB Journal 9 (2000) 76–110

- 9. Seaborne, A.: SPARQL query language for RDF. W3C Working Draft 21 (2005)
- Zimnyi, E., dit Gabouje, S.S.: Semantic visualization of biochemical databases. In: Semantics of a Networked World: Semantics for Grid Databases, LNCS 3226. (2004)
- Yan, X., Yu, P.S., Han, J.: Substructure similarity search in graph databases. In: Proc. ACM SIGMOD International Conference on Management of Data, New York, NY, USA, ACM Press (2005) 766–777
- Tri'sl, S., Leser, U.: Querying ontologies in relational database systems. In: DILS '05: Proc. 2nd International Conference on Data Integration in Life Sciences. (2005)
- Fegaras, L., Maier, D.: Towards an effective calculus for object query languages. In: ACM SIGMOD International Conference on Management of Data, San Jose, CA, ACM (1995) 47–58
- Chen, L., Gupta, A., Kurul, M.E.: Stack-based algorithms for pattern matching on dags. In: Proc. 31st Int. Conf. on Very Large Databases (VLDB), Stockholm. (2005) 493–504
- Vagena, Z., Moro, M.M., Tsotras, V.J.: Twig query processing over graphstructured xml data. In: WebDB '04: Proc. 7th International Workshop on the Web and Databases. (2004) 43–48
- Wang, H., He, H., Yang, J., Yu, P., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: ICDE '06: Proc. 22nd International Conference on Data Engineering. (2006 (to appear))
- Fegaras, L., Elmasri, R.: Query engines for web-accessible xml data. In: Proceedings of the 27th Int. Conf. on Very Large Data Bases (VLDB), San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2001) 251–260
- Bang-Jensen, J., Gutin, G.: Digraphs: Theory, Algorithms and Applications. Springer-Verlag, London (2001)