Generating Rewrite Rules by Browsing RDF Data

Ora Lassila
Nokia Research Center Cambridge
Cambridge, MA 02142, USA

Abstract

In this paper we show how path queries in the WILBUR *query engine can be augmented via the use of a simple rule system. This system can substitute atomic query expressions with complex queries, effectively rewriting queries. We demonstrate that this approach can be used for simple reasoning in the context of RDF(S). We further present a tool that allows the user to create complex queries merely by browsing an RDF graph, and by virtue of naming the automatically generated queries can create rules for the rule system. These rules appear in the system as "virtual" RDF properties, and enable the user to interactively customize how RDF data is presented.*

1 Introduction

One view of the Semantic Web [3] is that it encompasses the application of – mainly logic-based – knowledge representation in the context of the World Wide Web. There is a strong emphasis on *machine-based interpretation* of data and *automatic* (even *autonomous*) processing. Without disputing the benefits of automated (and autonomous) operation in the context of the Semantic Web, we take a different approach in this paper: Semantic Web data can be visualized and presented to human users for browsing, exploration, querying, etc. The recent emergence of tools such as "Piggy Bank" [11] demonstrate the benefits of end user access to Semantic Web data.

We present our own solution to browsing RDF data [13, 20, 5], a system dubbed "OINK" (for "Open Integration of Networked Knowledge"). RDF data – structurally a *graph* – can be presented to the user as hypertext. While browsing, the path that the user takes through this graph will be used as a "template" to find other items of interest that are similarly related to the ones the user has encountered. In other words, the user's path is interpreted as a *query* in a *path query language*. Once discovered, these queries can be named and will become "rules" that are used in presenting the user with a customized, expanded view of the underlying RDF graph. We will demonstrate that relevant and realistic queries can be derived from the user's navigational paths. It is our belief that in many cases the automatic generation of queries is a better approach than forcing the user to formulate queries in a language that she first has to learn.

While OINK itself is a practical tool that we use in everyday work for debugging and reviewing Semantic Web data, the mechanisms for generating queries automatically and using them as a basis for rewrite rules are work in progress. Most notably, we are yet to conduct any user- or usability studies of these mechanisms.

2 A Path Query Language

OINK is implemented on top of the WILBUR Semantic Web toolkit [14, 15] and relies on WILBUR's query language WILBURQL. In this language, queries are expressed as *path patterns* that match paths between a *root node* and the members of a result set. The path patterns, effectively, are regular expressions over properties of the underlying RDF data (i.e., they are regular expressions over the edge labels of the graph). The query language is described in detail in [16, 18].[1] In the context of this paper, it is sufficient to know that path expressions are either *atomic* or *complex*: In the former case a path expression is either an RDF property, or a special token known to the query engine (some of these are described below); in the latter case, a query expression is of the form $op(e_1, \ldots, e_n)$ where $e_1, \ldots, e_n$ are typically path expressions. The operators *op* known to the query engine are:

- **Sequence** (*concatenation* in [21]): $seq(e_1, \ldots, e_n)$ matches a sequence of n steps in the graph, consisting of subexpressions $e_1, \ldots, e_n$.
- **Disjunction** (*alternation*): $or(e_1, \ldots, e_n)$ matches any one of n subexpressions $e_1, \ldots, e_n$. The subexpressions are matched in the order they are specified.

[1] A description of the actual query algorithm – limited to the features relevant to this paper – is given in the Appendix.

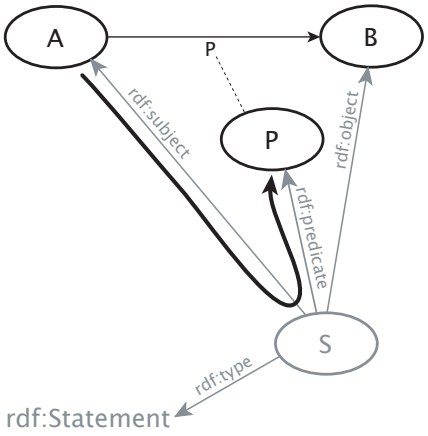

Figure 1. Graph matched by *predicate-of-subject*

- **Repetition** (*closure*): $rep(e)$ matches the transitive closure of subexpression e.

- **Inverse**: Satisfaction of $inv(e)$ requires the path defined by the subexpression e to be matched in reverse direction – this is similar to the *inversion* operator of GraphLog [7].

- **Default value**: The expression $value(r)$ will insert the value r into the result set of the query. This can be used to specify *default values*, typically using the following pattern:

$$or(q, value(r)) \tag{1}$$

  This query would return at least one value, r, in its result set, irrespective of q.

- **Restriction**: $restrict(e)$ matches the value e. For example, the query expression

$$seq(e_1, restrict(e_2), e_3)$$

  matches whatever $seq(e_1, e_3)$ would match, but only if the *node* after the step e_1 equals e_2.

- **Traversal to property nodes**: Two special tokens can be used in place of any property in a query expression: *predicate-of-subject* allows traversal *through* a node representing a *reified statement* in RDF, even if this node did not actually exist in the graph (see Figure 1), from the subject of the statement to the node naming its predicate. Similarly, *predicate-of-object* allows traversal from the object of the statement.[2]

[2]Admittedly this may seem like strange way of doing things, but it maintains the idea that *everything* in WILBURQL is expressed as graph traversal.

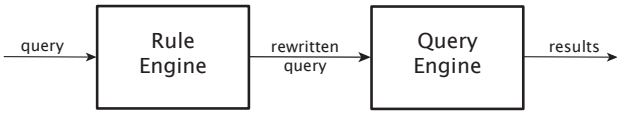

Figure 2. Simplified system architecture

Essentially, WILBURQL treats RDF graphs as *data structures*. Whether such a data structure represents the original data or its deductive closure [9] is a matter of implementation choice. Specifically, OINK relies on WILBUR's RDF(S) reasoner [15] to have access to the entailments of the stored graph(s).

The query engine is invoked via the access function $A(v, q, \mathcal{G})$, where v is a *root node* (i.e., a starting node of the query), q is a query expression, and $\mathcal{G}$ is the underlying graph database; see the Appendix for details.

3 Rewriting Path Queries

WILBUR's RDF(S) reasoner is implemented by *rewriting* access queries – expressions of the WILBURQL language – to effectively make it look like the database, in addition to the graph stored in it, contains all the entailments of this graph. Rewriting is done by recursively substituting all occurrences of those RDF properties that have a semantic theory (`rdf:type`, `rdfs:subClassOf` and `rdfs:subPropertyOf`) with a more complex query expression that effectively constitutes something loosely resembling backward-chaining rules. The rewritten queries create a "view" into the underlying graph database that contains the RDF(S) closure of the original graph; this technique is described in [15].

3.1 Using Rules to Rewrite Path Queries

The rewriting mechanism can be thought of as a simple rule system. *Rewrite rules* take the general form

$$p \rightarrow q \tag{2}$$

where p is an *atomic* expression of the query language (i.e., something that conceivably could name an RDF property), and q is an arbitrarily complex query expression that does not contain p (with one exception that's described below). The *rule engine*, upon seeing a query expression, iteratively applies the rewrite rules to the expression until no rule applies. The resulting rewritten query is then presented to the *query engine* that computes and retrieves the result set. This architecture is illustrated in Figure 2.

In queries of the form (2) it is possible for the expression q to contain subexpressions of the form $norewrite(r)$

where r is an expression that may contain the atomic expression p. The rule engine does not attempt to rewrite these expressions, and the $norewrite$ operator is ignored by the query engine, in the sense that from its point of view, $norewrite(p) \equiv p$. This mechanism allows one to augment the behavior of existing RDF properties, for example via the following pattern:

$$p \rightarrow or(norewrite(p), q)$$

Referring to (1), we can give property p the default value of r by using the following rule:

$$p \rightarrow or(norewrite(p), value(r)) \quad (3)$$

3.2 Reasoning as Path Query Rewriting

As another example of the use of $norewrite(p)$, the rule that says that all instances of a particular RDF class are also instances of any of the superclasses of this class could be expressed as follows:

$$t \rightarrow seq(norewrite(t), rep(s)) \quad (4)$$

where $t \equiv$ `rdf:type` and $s \equiv$ `rdfs:subClassOf`. Apart from a few exceptions, computation of the entire RDF(S) closure can be specified using similar rules. Assuming that the subclass rule

$$s \rightarrow rep(norewrite(s))$$

has also been specified, the full form of the rule (4) would be the following:

$$\begin{aligned} t \rightarrow or(&seq(norewrite(t), s),\\ &seq(\textit{predicate-of-object}, \texttt{rdfs:range}, s),\\ &seq(\textit{predicate-of-subject}, \texttt{rdfs:domain}, s),\\ &value(\texttt{rdfs:Resource})) \end{aligned}$$

It should be noted that in the current implementation of WILBUR, subproperties are handled by a special rewrite rule that cannot be specified using our simple rule language.

4 Generating Rewrite Rules

We now present a system that allows the user to interact with RDF data. This system, dubbed OINK, was built as a means of visualizing RDF graphs mainly for debugging purposes [17]; subsequently the system has been developed as a platform for building customized solutions for browsing complex data. We will present automatic query generation in OINK as a basis for enabling the user to define rules, and more generally to allow customization of how RDF data is presented.

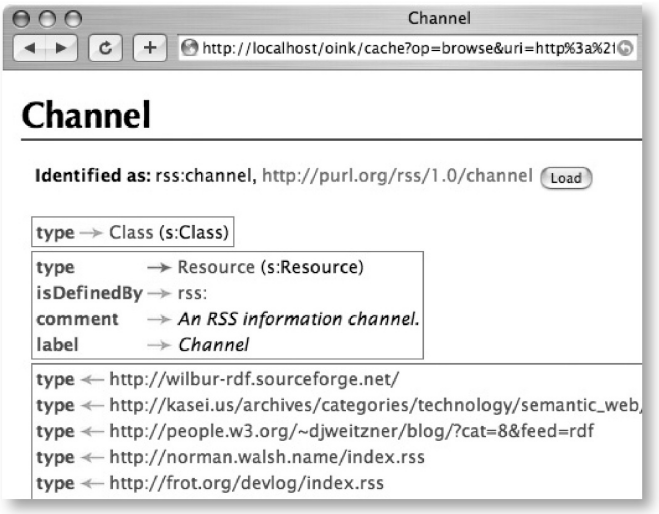

Figure 3. OINK page of the `rss:channel` class

Figure 3 shows a page from OINK visualizing the `rss:channel` class from the RSS schema[3] (this class can be used to represent syndicated news feeds such as blogs, etc.). OINK shows both outbound and inbound edges of the graph; because OINK relies on the RDF metamodel (where everything is represented in RDF itself), everything on the page is "clickable", i.e., can be navigated to.

4.1 Query-by-Browsing

The basic idea behind the automatic construction of path queries is simple: While browsing RDF data using OINK, the user takes some path through the graph. Upon finding an item of interest, she may want to know if there are other items similarly related to some item she has viewed earlier. The path she has taken provides the basis of the query to be expressed in WILBURQL. Figure 4 shows the steps to an automatically generated query; starting from the aforementioned `rss:channel` class (Figure 3), we navigate to one of its instances (an RSS profile of a blog), and then take the following steps:

1. From the `rss:channel` instance, we navigate to the home page URL of the blog in question (home pages are associated with the "feed" via the `rdfs:seeAlso` property).

2. The home page URL is associated with a `dc:title` property, giving the human-readable title for the blog.

3. Invoking a query based on the path we have taken from the `rss:Channel` class node, we get the list of the titles of all the blogs currently known to the system. The

[3] http://web.resource.org/rss/1.0/spec

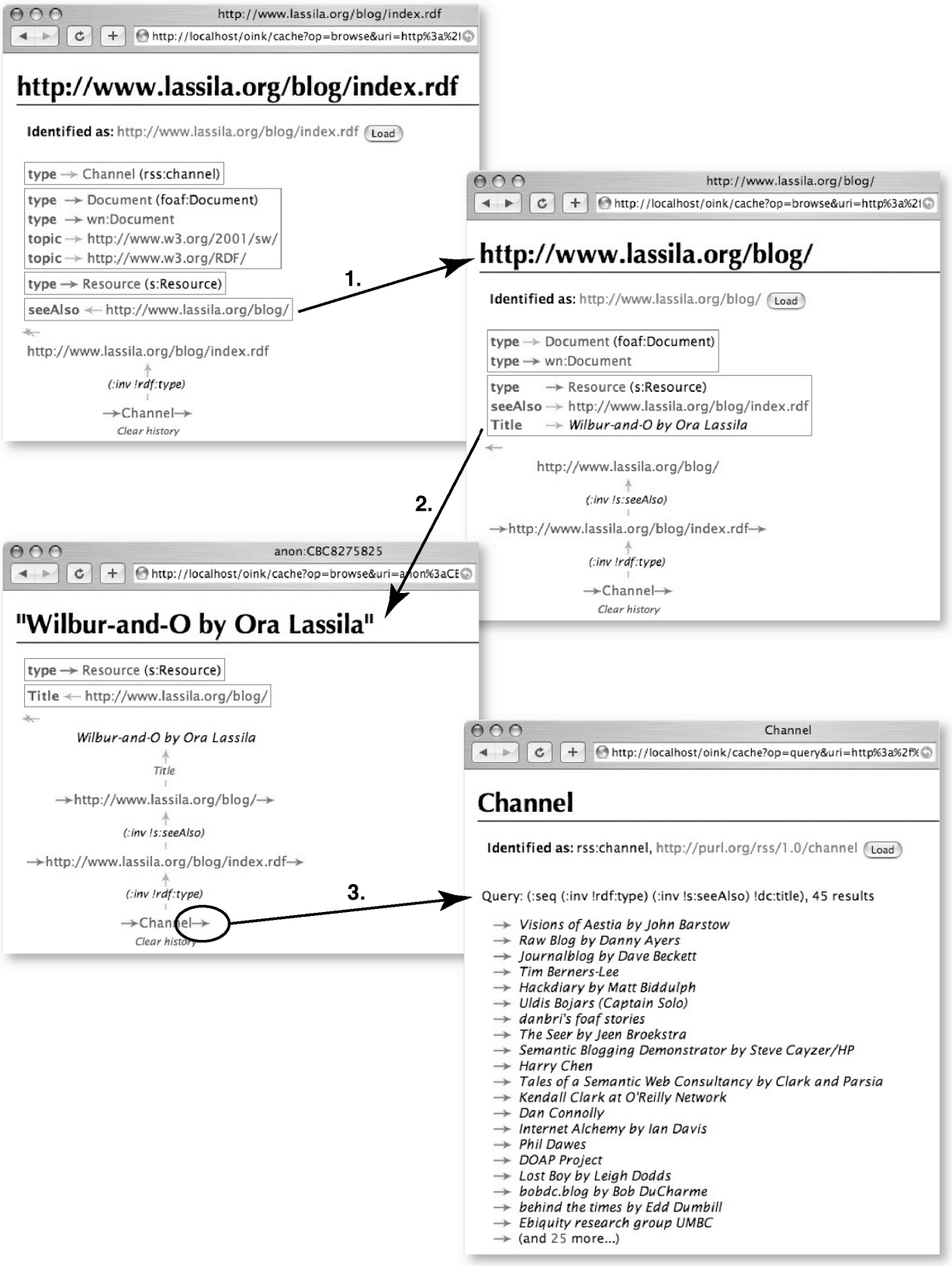

Figure 4. Steps to an automatically generated query

corresponding query is

$$seq(inv(\texttt{rdf:type}), \quad (5)$$
$$inv(\texttt{rdfs:seeAlso}),$$
$$\texttt{dc:title})$$

In this example the data was taken from the "blogroll" of the aggregated blog "Planet RDF".[4]

OINK can also augment the generated queries using simple heuristics. For example, the occurrence of the same property more than once in a row is changed to an arbitrary repetition of that property.

4.2 Using "Virtual" Properties

Virtual properties are rewrite rules that allow the values of new properties to be computed rather than being stored in the underlying database. To follow the example from the previous section, we could define a new property for blog RSS profiles, say, `ex:title` and define the following rule based on (5):

$$\texttt{ex:title} \rightarrow seq(inv(\texttt{rdfs:seeAlso}), \texttt{dc:title})$$

WILBUR provides a class `wilbur:PathRewriteRule` as a subclass of `rdf:Property`. This allows us to have virtual properties be associated with `rdfs:domain` definitions; given a virtual property p with a domain d, OINK uses this information to query for values of p whenever visualizing an instance of the class d. We are currently working on a user interface (as part of OINK) to allow users to add the domain (and other properties) to interactively generated rewrite rules.

5 Related Work

The idea of *rewriting* has been used as a general computational vehicle [10], in transforming and optimizing (especially functional) programming languages [24, 23, 12], in various practical software systems (such as the `sendmail` program[5] of UNIX or the Apache web server[6]) as a "customization" mechanism, and in performing service discovery and composition [2, 19]. It has also been used in query systems for semi-structured data for providing augmented or federated views of databases [6, 22, 8].

In comparison to most of the aforementioned work, our approach to rewriting query expressions is rather simplistic and resembles simple *macro expansion*. Most notably, our rewrite rules do not contain any variables or other types of patterns that would require matching to determine rule applicability (apart from a simple exact matching of names). In that sense, they resemble *entity references* in XML [4, section 4.1].

6 Conclusions and Further Development

The OINK system presented in this paper allows users access to RDF data via browsing. We currently use the system as a debugging tool when building Semantic Web applications. OINK also allows users to interactively build queries in the WILBURQL path query language merely by browsing their data; navigational paths are translated into path queries. By naming such queries users can define rewrite rules which, effectively, are used not only to add "virtual" RDF properties to the visualizations, but are also used to augment subsequent queries by rewriting them before the actual query engine processes them. The underlying RDF storage system uses query rewriting to implement an RDF(S) reasoner; the users are presented a view of the underlying stored RDF documents as a single graph that also contains its entailments.

Although the query generation mechanism is work in progress (e.g., no formal user testing has been conducted at the time of writing) we believe that the feature is useful in a tool intended for viewing and exploring Semantic Web data, since it allows sophisticated querying without the user having to learn the syntactic details of the underlying query language.

Planned future development of the system will focus on adding features for further customization, and will consequently take the system towards a *platform* for developing browsers/viewers for complex data. The ability to interactively add "virtual" properties (i.e., rewrite rules) allows users themselves to customize how their data is presented to them.

[4] http://journal.dajobe.org/journal/2003/07/semblogs/bloggers.rdf
[5] http://www.sendmail.org/
[6] http://httpd.apache.org/

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] B. Benatallah, M.-S. Hacid, C. Rey, and F. Toumani. Request Rewriting-Based Web Service Discovery. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *The Semantic Web - ISWC 2003, 2nd International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 242–257. Springer-Verlag, Oct. 2003.

[3] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, World Wide Web Consortium, Feb. 1998.

[5] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Working Draft, World Wide Web Consortium, Jan. 2003.

[6] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of Regular Expressions and Regular Path Queries. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 194–204, New York, NY, USA, 1999. ACM Press.

[7] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 1990.

[8] G. Grahne and A. Thomo. Query Containment and Rewriting Using Views for Regular Path Queries under Constraints. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 111–122, New York, NY, USA, 2003. ACM Press.

[9] P. Hayes. RDF Semantics. W3C Recommendation, World Wide Web Consortium, Feb. 2004.

[10] G. Huet and D. C. Oppen. Equations and Rewrite Rules: A Survey. Technical Report CS-TR-80-785, Stanford University, Stanford, CA, 1980.

[11] D. Hyunh, S. Mazzocchi, and D. Karger. Piggy bank: Experience the Semantic Web Inside Your Web Browser. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *The Semantic Web – ISWC 2005, 4th International Semantic Web Conference*. Springer-Verlag, 2005.

[12] D. Lacey and O. de Moor. Imperative Program Transformation by Rewriting. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 52–68, London, UK, 2001. Springer-Verlag.

[13] O. Lassila. Web Metadata: A Matter of Semantics. *IEEE Internet Computing*, 2(4):30–37, 1998.

[14] O. Lassila. Enabling Semantic Web Programming by Integrating RDF and Common Lisp. In *Proceedings of the First Semantic Web Working Symposium*. Stanford University, 2001.

[15] O. Lassila. Taking the RDF Model Theory Out for a Spin. In I. Horrocks and J. Hendler, editors, *The Semantic Web - ISWC 2002, 1st International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 307–317. Springer-Verlag, 2002.

[16] O. Lassila. Wilbur Query Language Comparison. Nokia Research Center technical report, available online at http://wilbur-rdf.sourceforge.net/2004/05/11-comparison.shtml, 2004.

[17] O. Lassila. Browsing the Semantic Web. Accepted to the International Workshop on Web Semantics, 2006.

[18] O. Lassila. Semantic Web Programming Using Common Lisp – Programmers' Guide to the Wilbur Semantic Web Toolkit. Technical report, Nokia Research Center, Cambridge, MA, 2006.

[19] O. Lassila and S. Dixit. Interleaving Discovery and Composition for Simple Workflows. In *Semantic Web Services, AAAI Spring Symposium Series*. AAAI, 2004.

[20] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, World Wide Web Consortium, Feb. 1999.

[21] A. O. Mendelzon and P. T. Wood. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing*, 24(6):1235–1258, Dec. 1995.

[22] Y. Papakonstantinou and V. Vassalos. Query Rewriting for Semistructured Data. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 455–466, New York, NY, USA, 1999. ACM Press.

[23] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 357–362, London, UK, 2001. Springer-Verlag.

[24] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

A Appendix: WILBURQL Query Algorithm

A description of the WILBURQL query algorithm, limited to those features that are relevant in this paper, can be given in terms of the following functions:

- $A(v, q, \mathcal{G})$ is the main access function, as introduced in section 2; (6).
- $walk(v, dfa, i, \mathcal{G})$ is a procedure for "walking" a graph, guided by a finite state automaton dfa; (7).
- $expand(v, e, \mathcal{G})$ is a function that "expands" the query from vertex v via the edge e (where e can also be one of the special tokens supported by the query engine); (8–15).
- $triple(s, p, o, \mathcal{G})$ matches edges from the graph; in a concrete implementation, such as WILBUR, this function is the main interface to the underlying graph storage; (16).
- $makedfa(q)$ constructs a DFA corresponding to the query expression q, using a standard algorithm for translating a regular expression into a finite state automaton [1, algorithm 3.5]. A DFA, in our case, is a vector of *states*, each of which is a set of pairs $\langle input, index \rangle$ where $input$ is any edge label of the queries graph (more specifically, any edge parameter e of the function $expand(v, e, \mathcal{G})$, and $index$ is a state index of the DFA. All states also record whether they are *terminal* or not.

```
1  procedure A(v, q, G)
2      N ← {}, S ← {}
3      call walk(v, makedfa(q), 0, G)
4      return N
5  end A
```
(6)

```
1  procedure walk(v, dfa, i, G)
2      if ⟨i, v⟩ ∉ S then
3          S ← S ∪ {⟨i, v⟩}
4          state ← dfa[i]
5          if is_terminal(state) then
6              N ← N ∪ {v}
7          end if
8          for ⟨input, j⟩ ∈ state do
9              for v' ∈ expand(v, input, G) do
10                 call walk(v', dfa, j, G)
11             end for
12         end for
13     end if
14 end walk
```
(7)

$$expand(v, \phi, \mathcal{G}) \tag{8}$$
$$= \{o \mid \langle s, p, o \rangle \in \text{triple}(v, \phi, *, \mathcal{G})\}$$

$$expand(v, inv(\phi), \mathcal{G}) \tag{9}$$
$$= \{s \mid \langle s, p, o \rangle \in \text{triple}(*, \phi, v, \mathcal{G})\}$$

$$expand(v, restrict(e), \mathcal{G}) = \begin{cases} \{v\}, \text{ if } v = e \\ \{\}, \text{ otherwise} \end{cases} \tag{10}$$

$$expand(v, value(e), \mathcal{G}) = \{e\} \tag{11}$$

$$expand(v, \textit{predicate-of-subject}, \mathcal{G}) \tag{12}$$
$$= \{p \mid \langle s, p, o \rangle \in triple(v, *, *, \mathcal{G})\}$$

$$expand(v, \textit{predicate-of-object}, \mathcal{G}) \tag{13}$$
$$= \{p \mid \langle s, p, o \rangle \in triple(*, *, v, \mathcal{G})\}$$

$$expand(v, inv(\textit{predicate-of-subject}), \mathcal{G}) \tag{14}$$
$$= \{s \mid \langle s, p, o \rangle \in triple(*, v, *, \mathcal{G})\}$$

$$expand(v, inv(\textit{predicate-of-object}), \mathcal{G}) \tag{15}$$
$$= \{o \mid \langle s, p, o \rangle \in triple(*, v, *, \mathcal{G})\}$$

$$\text{triple}(s, p, o, \mathcal{G}) \tag{16}$$
$$= \{\langle \sigma, \pi, \omega \rangle \in \mathcal{G} \mid (s \stackrel{*}{=} \sigma) \wedge (p \stackrel{*}{=} \pi) \wedge (o \stackrel{*}{=} \omega)\}$$
$$\text{where } (x \stackrel{*}{=} y) \equiv \begin{cases} true, \text{ if } x = * \\ x = y, \text{ otherwise} \end{cases}$$