Taking the RDF Model Theory Out for a Spin

Ora Lassila

Nokia Research Center*, 5 Wayside Road, Burlington MA 01803, USA

Abstract. Entailment, as defined by RDF's model-theoretical semantics, is a basic requirement for processing RDF, and represents the kind of "semantic interoperability" that RDF-based systems have been anticipated to have to realize the vision of the "Semantic Web". In this paper we give some results in our investigation of a practical implementation of the entailment rules, based on the graph-walking query mechanism of the WILBUR RDF toolkit.

1 Introduction

Resource Description Framework (RDF), the World Wide Web Consortium's metadata framework [13], has emerged as a basic building block for the so-called "Semantic Web" [3]. Semantic interoperability of systems processing RDF is largely anticipated to emerge because of the implied polymorphism of shared types and relations as defined using the ontological vocabulary of RDF [4]. Most RDF-based software toolkits, however, merely concentrate on producing sets of triples from XML serializations of RDF graphs, and leave the inferential part to the application programmer. The recently published model-theoretical semantics for RDF [10] formalizes the notion of inference in RDF, and provides a basis for computing deductive closures of RDF graphs. Since it can be argued that this is actually a basic requirement for interoperability of RDF-based systems, support for this should be readily available to application programmers. Not only would this ease the task of writing RDF-savvy software, but it would improve the level of interoperability between these systems. Without this minimal support for inference, RDF is largely relegated to mere structured data interchange, and its utility will be seriously jeopardized.

In this paper we will investigate the computational aspects of deductive closures of RDF graphs, and pursue an implementation based on WILBUR [12,18], Nokia Research Center's open source RDF toolkit. We will implement a "true RDF processor" by viewing RDF graphs through a node-centric "slot access function" A, defined as

$$value \in A(frame, slot) \iff \langle frame, value \rangle \in IEXT(I(slot)) \qquad (1)$$

where $I(x)$ is the RDFS-interpretation of a particular graph, and $IEXT(y)$ is a binary relational extension of a property – i.e., the set of pairs which identify the arguments for which the property is true – as defined in [10].

* Research described in this paper was supported in part by Nokia Research Center, Nokia Mobile Phones and Nokia Venture Partners.

The basic Wilbur frame API [18] provides a simple lookup implementation for A (we will call it A_{lookup}) where entailment is not considered. If D is the current database of triples $\langle s, p, o \rangle$, then A_{lookup} is basically defined as

$$value \in A_{lookup}(frame, slot) \iff \langle frame, slot, value \rangle \in D \tag{2}$$

We will demonstrate one approach to implementing A, given an implementation of A_{lookup} and other query/update facilities for D.

2 Entailment and "RDFS-Closures"

The RDF Model Theory [10] defines entailment via the generation of a deductive closure from an RDF graph. The closure is a graph consisting of every triple $\langle s, p, o \rangle$ that satisfies $\langle s, o \rangle \in IEXT(I(p))$. Computing this so-called "RDFS-closure" consists of two steps:

1. Addition of a set of new (static) triples to the RDF graph in question. These triples effectively define classes and properties (and their domains and ranges) in the basic RDF ontological vocabulary. An XML-encoded RDF file producing these triples is given as an example in Appendix A.
2. Recursive application of forward-chaining rules to generate all legal triples entailed by the graph in question. These rules could be characterized as follows:
   - *Type Rules* assign default ("root") types for resources (rules **rdf1**, **rdfs4a** and **rdfs4b** in [10]).
   - *Subclass Rules* generate the transitive closures of *subclass* → *class* and *instance* → *class* links (rules **rdfs7**, **rdfs8** and **rdfs9**).
   - *Subproperty Rules* are used to generate the transitive closures resulting from *subproperty* → *property* links. They also propagate property values up the subproperty chain (rules **rdfs5** and **rdfs6**).
   - *Domain/Range Rules* infer resource types from *domain* and *range* assignments (rules **rdfs2** and **rdfs3**).

The rules are highly redundant, and their brute-force, exhaustive, iterative application of is not a realistic way of computing the closure. For example, given a graph with only one triple, the rules in step 2 would generate 17 new triples (in addition to the 19 "static" triples added in step 1), but would also result in 493 attempts to add a redundant triple (i.e., one that was already in the database). Forward-chaining rule-based techniques – such as RETE [8] – could be used to make this processing more efficient, but another issue is that the application of the rules may result in the addition of a large number of new triples in the database (and that most of these generated results may never be needed). It is therefore interesting to investigate whether some balance could be found between computing the closure in advance vs. defining the access function A in such a manner that it can dynamically (i.e., on-demand) generate correct results.

3 Graph Queries

Wilbur exposes RDF graphs through a node-centric (i.e., "frame system") API. As part of this API, the slot access function A_{lookup} supports a query language which allows complex access paths – expressed as regular expressions of *slot names* (i.e., RDF properties) – to be used in place of atomic slot names. The query language is an extension of the query mechanism of the BEEF frame system [11] which, in turn, is an efficient implementation of a simplification of the CRL/SRL path language [9]. It resembles query languages constructed for semi-structured and graph-based data (e.g., [1,6,14,5]).

Path expressions can take the following forms (expressed here in pseudo-syntax instead of the native s-expression syntax of Wilbur):

1. **Sequence** (*concatenation* in [14]):
   $seq(e_1, \ldots, e_n)$ matches a sequence of n steps in the graph, consisting of subexpressions $e_1, \ldots, e_n$.
2. **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.
3. **Repetition** (*closure*):
   $rep(e)$ matches the transitive closure of subexpression e; $rep^+(e)$ is equivalent to $seq(e, rep(e))$.
4. **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 [5].
5. **Value**:
   $val(e)$ causes the value e to be generated in the matching process, ignoring any actual slot accesses. It is useful in specifying *default values*, typically using the idiom $or(path, val(default))$.[1]
6. **Wildcards**:
   The query language supports wildcard "atoms" matching either *any* arc label or just RDF *container membership* properties.

Given a "root" node (i.e., a search start point) and a query expression, Wilbur provides functions for retrieving the first reachable node, retrieving all reachable nodes (function A_{lookup}), and determining whether a path exists between two specified nodes.

Wilbur transforms query expressions into optimized deterministic finite state automata [2, section 3.9] and uses these to effectively "walk" the underlying RDF graphs (which are stored as RDF triples in in-core databases with hashed indices). During traversal, graph nodes are marked with DFA states as in [14, section 5] except that we do not have to restrict ourselves to simple paths (by marking the nodes with *all* applicable states Wilbur is able to find the correct answer to [14, example 8]).

[1] The current implementation cannot satisfy queries of type $inv(val(e))$.

4 Implementing Closure Generation

We implement closure generation primarily by using graph-walking techniques. Our approach is based on the following of basic assumptions:

1. Generally, we are willing delay the computation of the closure (even at the expense of the time eventually spent in the computation) and to trade memory consumption for time spent in computation.
2. The computational burden is split in two: some of the work is undertaken during every insert into D (i.e., whenever new triples are asserted), and some during every access of A.
3. Some features of RDF are more prevalent than others in "typical" data; we will base the design of the system on this perceived distribution of prevalence:
   - subclassing is common,
   - subproperty definitions are used but sparsely,
   - subproperties of `rdf:subPropertyOf` are rare.
4. Retractions from D are not considered (so far).

With regard to the dynamic computation of closures, our approach is based on the WILBUR query language and rewriting access path expressions when accessing the underlying graph. The definition of the slot access function A now takes the form

$$A(frame, path) = A_{lookup}(frame, path') \tag{3}$$

where *path′* is the path expression *path* suitably rewritten. We will express the algorithm as a set of rewrite patterns of the form *path* → *path′*.

We will first demonstrate a partial solution: it implements only the *type* and *subclass* rules discussed in section 2. We will then extend this solution to a complete one by adding support for the *subproperty* rules.

4.1 Partial Solution

For the two core relations `rdf:type` and `rdfs:subClassOf` the rewritten paths (referring to equation 3) are, correspondingly:

$$\begin{aligned} \texttt{rdf}:\texttt{type} &\rightarrow \mathit{or}(\mathit{seq}(\texttt{rdf}:\texttt{type}, \mathit{rep}(\texttt{rdfs}:\texttt{subClassOf})), \\ &\qquad \mathit{val}(\texttt{rdfs}:\texttt{Resource})) \qquad (4) \\ \texttt{rdfs}:\texttt{subClassOf} &\rightarrow \mathit{or}(\mathit{rep}(\texttt{rdfs}:\texttt{subClassOf}), \\ &\qquad \mathit{val}(\texttt{rdfs}:\texttt{Resource})) \qquad (5) \end{aligned}$$

where rewrite pattern 4 says that in order to find all values of `rdf:type` of an instance, you first traverse the atomic `rdf:type` link once, and then the atomic `rdfs:subClassOf` link an arbitrary number of times (including zero). Accessing all values of this relation computes the transitive closure of `rdfs:subClassOf`, starting from the designated classes of the instance being queried. Similarly, rewrite pattern 5 accesses the transitive closure of `rdfs:subClassOf`. Note that the effects of pattern 5 are built into pattern 4 so that these rules do not need to be applied recursively. The disjunctions in both expressions ensure that if the exhaustive search (i.e., transitive closure computation) fails, a default value is generated.

Apart from `rdf:type` and `rdfs:subClassOf`, other atomic slot names (RDF properties) are unaffected by the rewrite process, since there is no semantic theory for them. Complex path expressions are rewritten by traversing them recursively, rewriting subexpressions.

Since the Wilbur implementation of a "triple database" always loads a basic "RDF schema" into every newly created database, step 1 of the closure generation process (in section 2) is implemented by defining the static triples in this schema (see Appendix A).

Please note that the approach we have taken only makes sense for certain types of triple database implementations. In a relational database implementation – given that queries for finding transitive closures cannot be expressed in relational calculus (see, for example, [14]) – it might make more sense to populate the database with additional triples. In an "in-core" implementation like Wilbur, stepping through the graph has relatively low cost, and therefore the dynamic approach makes sense, particularly when combined with the potential memory savings.

4.2 Complete Solution

We can extend the partial solution to provide support for the *subproperty* rules. Referring to equation 3, we rewrite access paths as follows: each atomic relation r is rewritten as

$$\begin{aligned} r \rightarrow\ & or(r_1, \ldots, r_n) \\ & \text{where } r_i \in A_{lookup}(r, rep(inv(or(p_1, \ldots, p_m)))) \end{aligned} \tag{6}$$

and where $p_1, \ldots, p_m$ are the relation `rdfs:subPropertyOf` and all of its defined subproperties. Please note that this rewriting also applies to all of the atoms of the results of applying the rewrite patterns 4 and 5. When all values of A are computed the ordering of r_i does not need to be considered. An implementation might, though, apply some specificity ordering to the values based on the graph distance of r_i to r (note that $r_1 = r$).

The set of subproperties of `rdfs:subPropertyOf`, $P = \{p_i\}$, is cached. Each insert into D where the triple is of the form $\langle s, p_i, p_j \rangle$ where $p_i \in P \vee p_j \in P$ invalidates and recomputes the cache. The recomputation is effected as follows: assume P_{old} is the current value of the cache, and P_{new} is the recomputed value of the cache; then

$$\begin{aligned} P_{new} =\ & A_{lookup}(\texttt{rdfs:subPropertyOf}, inv(rep(or(p_1, \ldots, p_n)))) \\ & \text{where } \forall i \in [1, n], p_i \in P_{old} \end{aligned} \tag{7}$$

In addition to caching subproperty information of `rdfs:subPropertyOf`, the implementation offers other opportunities for caching results. Not only could more of the subproperty information be cached (that is, information about subproperties of *all* relations, not just `rdfs:subPropertyOf`), but other results computed by A could be cached as well.

4.3 About Domain/Range Rules

The *domain* and *range* constraints of RDF Schema were originally introduced to allow RDF data to be validated (e.g., by a metadata editor). The *domain/range rules* of the Model Theory make it impossible to use RDF Schema for this purpose since they effectively treat *domain* and *range* generatively and not restrictively – and these are the *only* validation constraints of the language. We believe these rules should not be part of the Model Theory in the first place.

If one did want to implement the *domain/range rules* using the access path rewriting technique, one would have to add additional triples to the database during insertions. For example, if for every triple $\langle s, p, o\rangle$ inserted into D one would insert the triples $\langle o, \mathtt{rr}, p\rangle$ and $\langle s, \mathtt{dr}, p\rangle$ into D, one could rephrase the rewrite pattern 4 as follows:

$$\begin{aligned}\mathtt{rdf:type} \rightarrow or(&seq(\mathtt{rdf:type}, rep(\mathtt{rdfs:subClassOf})),\\ &seq(\mathtt{rr}, rep(or(p_1,\ldots,p_m)), \mathtt{rdfs:range}),\\ &seq(\mathtt{dr}, rep(or(p_1,\ldots,p_m)), \mathtt{rdfs:domain})\\ &val(\mathtt{rdfs:Resource}))\end{aligned} \tag{8}$$

where $p_1, \ldots, p_m$ are the relation `rdfs:subPropertyOf` and all of its subproperties. This approach, however, would lead to an expansion of D and would thus work against the goals for this implementation in general.

4.4 Implementation Summary

The following table summarizes how our approach implements the rules of the Model Theory:

Rule	**Implementation**
rdf1	during insertions to D
rdfs2	*not implemented* (see section 4.3)
rdfs3	*not implemented* (see section 4.3)
rdfs4a	rewrite pattern 4 (default clause)
rdfs4b	rewrite pattern 4 (default clause)
rdfs5	rewrite pattern 6
rdfs6	rewrite pattern 6 + caching during insertions to D
rdfs7	rewrite pattern 5 (default clause)
rdfs8	rewrite pattern 5
rdfs9	rewrite pattern 4

5 Future Work

The complexity of path queries has been studied extensively (for example [16,17,14] just to name a few). Even though the general problems tend to be NP-complete [14], several restricted variations of the problem have lower complexity. Most of the graph processing required for our solution is reduced to the computation of transitive closures which can be accomplished in polynomial time [17]. Our future plans include not only analyzing the complexity of the current solution but also comparing it with others, such as generative forward-chaining rule-based approaches – e.g., CWM [19] – and backward-chaining theorem proving approaches – e.g., Euler [20] as well as SiLRI and TRIPLE [7,15].

Additional future work on this system will include dealing with retractions from D (for practical completeness – this will affect how rules **rdf1** and **rdfs6** are implemented), getting statistical data to back up the assumptions made of the distribution of various RDF features in “real-world” RDF data, and extending the underlying WILBUR system to deal with certain types of queries that include the pattern $inv(val(r))$.

6 Conclusions

“Semantic interoperability” of RDF-based systems has long been anticipated to materialize because of the polymorphism of shared types and relations as defined by the RDF Schema specification, but most RDF-based software packages merely concentrate on producing sets of triples from XML serializations of RDF graphs and leave the inferential part to the application programmer. The model theory for RDF formalizes this notion of inference in RDF. We have argued that the inferential mechanism is a basic minimum requirement for interoperability of RDF-based systems, and support for this should be readily available for application programmers. Not providing this support may compromise the interoperability between RDF-based systems.

The “true RDF processor” presented in this paper provides a “slot access function” which effectively allows the underlying graph to be viewed *as if* its RDFS-closure had been generated. RDF-based applications, if written using a toolkit like this, would not have to worry about the inferential implications specified by the standard, and would thus enable true interoperability with other similar systems.

Acknowledgements

The author would like to thank the following individuals for help in the preparation of this article: Mark Adler, Jos de Roo, Pat Hayes, Marcia Lassila, Deborah McGuinness, Esko Nuutila, Louis Theran and the anonymous reviewers of the ISWC program committee. The research work described in this paper was supported in part by Nokia Research Center, Nokia Mobile Phones and Nokia Venture Partners.

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
3. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.
4. D. Brickley and R.V.Guha. Resource Description Framework (RDF) Schema Specification 1.0. W3C Candidate Recommendation, March 2000.
5. 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.
6. I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, pages 323–330, 1987.
7. S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for RDF. In *W3C Query Languages Workshop (QL'98)*, December 1998.
8. C. L. Forgy. A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.
9. M. S. Fox. Knowledge representation for decision support. In Methlie and Sprague, editors, *Knowledge Representation for Decision Support Systems*. Elsevier, 1985.
10. P. Hayes. RDF Model Theory. W3C Working Draft, February 2002.
11. J. Hynynen and O. Lassila. On the Use of Object-Oriented Paradigm in a Distributed Problem Solver. *AI Communications*, 2(3):142–151, 1989.
12. O. Lassila. Enabling Semantic Web Programming by Integrating RDF and Common Lisp. In *Proceedings of the First Semantic Web Working Symposium*. Stanford University, 2001.
13. O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, February 1999.
14. A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, December 1995.
15. M. Sintek and S. Decker. TRIPLE—A Query, Inference, and Transformation Language for the Semantic Web. Accepted to ISWC 2002.
16. R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.
17. M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 230–242, 1990.
18. http://purl.org/net/wilbur/.
19. http://www.w3.org/2000/10/swap/doc/cwm.html.
20. http://www.agfa.com/w3c/euler/.

A Basic RDF and RDFS Schema

This RDF document provides the initial (static) triples required by the RDFS closure generation algorithm (see [10, section 6]).

```
<?xml version="1.0"?>

<!DOCTYPE uridef [
  <!ENTITY rdf ‘‘http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY rdfs ‘‘http://www.w3.org/2000/01/rdf-schema#">
]>

<rdf:RDF xmlns:rdf="&rdf;" xmlns:rdfs="&rdfs;">

  <rdfs:Class rdf:about="&rdfs;Resource"/>
  <rdfs:Class rdf:about="&rdf;Property"/>
  <rdfs:Class rdf:about="&rdfs;Class"/>

  <rdf:Property rdf:about="&rdf;type">
    <rdfs:domain rdf:resource="&rdfs;Resource"/>
    <rdfs:range rdf:resource="&rdfs;Class"/>
  </rdf:Property>

  <rdf:Property rdf:about="&rdfs;subClassOf">
    <rdfs:range rdf:resource="&rdfs;Class"/>
    <rdfs:domain rdf:resource="&rdfs;Class"/>
  </rdf:Property>

  <rdf:Property rdf:about="&rdfs;subPropertyOf">
    <rdfs:range rdf:resource="&rdf;Property"/>
    <rdfs:domain rdf:resource="&rdf;Property"/>
  </rdf:Property>

  <rdf:Property rdf:about="&rdfs;range">
    <rdfs:range rdf:resource="&rdfs;Class"/>
    <rdfs:domain rdf:resource="&rdf;Property"/>
  </rdf:Property>

  <rdf:Property rdf:about="&rdfs;domain">
    <rdfs:range rdf:resource="&rdfs;Class"/>
    <rdfs:domain rdf:resource="&rdf;Property"/>
  </rdf:Property>

  <rdfs:Class rdf:about="&rdfs;Literal"/>

</rdf:RDF>
```

B Slot Access Example

Assume a simple example with instances, classes, subclasses and a subproperty for `rdf:type`, as defined by the following RDF document:

```
<?xml version="1.0"?>

<!DOCTYPE uridef [
  <!ENTITY rdf ‘‘http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY rdfs ‘‘http://www.w3.org/2000/01/rdf-schema#">
  <!ENTITY x ‘‘http://www.lassila.org/schemata/Example#">
]>

<rdf:RDF xmlns:rdf="&rdf;" xmlns:rdfs="&rdfs;" xmlns:x="&x;">

  <rdfs:Property rdf:about="&x;type">
    <rdfs:subPropertyOf rdf:resource="&rdf;type"/>
  </rdfs:Property>

  <rdfs:Class rdf:about="&x;A">
    <rdfs:subClassOf>
      <rdfs:Class rdf:about="&x;B"/>
    </rdfs:subClassOf>
  </rdfs:Class>

  <rdf:Description rdf:about="&x;foo">
    <x:type rdf:resource="&x;A"/>
  </rdf:Description>

</rdf:RDF>
```

When the above document is loaded into Wilbur's database it will produce the following 6 triples:

$$\begin{aligned}
&1 : \langle \mathtt{x:type}, \mathtt{rdf:type}, \mathtt{rdfs:Property}\rangle \\
&2 : \langle \mathtt{x:type}, \mathtt{rdfs:subPropertyOf}, \mathtt{rdf:type}\rangle \\
&3 : \langle \mathtt{x:A}, \mathtt{rdf:type}, \mathtt{rdfs:Class}\rangle \\
&4 : \langle \mathtt{x:A}, \mathtt{rdfs:subClassOf}, \mathtt{x:B}\rangle \\
&5 : \langle \mathtt{x:B}, \mathtt{rdf:type}, \mathtt{rdfs:Class}\rangle \\
&6 : \langle \mathtt{x:foo}, \mathtt{rdf:type}, \mathtt{x:A}\rangle
\end{aligned}$$

Then, calling the new access function $A(\mathtt{x:foo}, \mathtt{rdf:type})$ will yield the result $\{\mathtt{x:A}, \mathtt{x:B}, \mathtt{rdfs:Resource}\}$. The execution of the function call will result in the following lower-level calls:

$$1 : \begin{cases} A_{lookup}(\mathtt{rdf:type}, inv(\mathtt{rdfs:subPropertyOf})) \rightarrow \{\mathtt{x:type}\} \\ A_{lookup}(\mathtt{x:type}, inv(\mathtt{rdfs:subPropertyOf})) \rightarrow \{\} \\ A_{lookup}(\mathtt{rdfs:subClassOf}, inv(\mathtt{rdfs:subPropertyOf})) \rightarrow \{\} \end{cases}$$

$$2 : \begin{cases} A_{lookup}(\mathtt{x:foo}, \mathtt{rdf:type}) \rightarrow \{\} \\ A_{lookup}(\mathtt{x:foo}, \mathtt{x:type}) \rightarrow \{\mathtt{x:A}\} \\ A_{lookup}(\mathtt{x:A}, \mathtt{rdfs:subClassOf}) \rightarrow \{\mathtt{x:B}\} \\ A_{lookup}(\mathtt{x:B}, \mathtt{rdfs:subClassOf}) \rightarrow \{\} \\ A_{lookup}(\mathtt{x:B}, val(\mathtt{rdfs:Resource})) \rightarrow \{\mathtt{rdfs:Resource}\} \end{cases}$$

Step 1 represents the rewriting step for `rdf:type`: the accesses to A_{lookup} are the result of the addition of triple 2 above having invalidated the sub-subproperty cache (subsequent similar calls would be able to rely on the cached information). Step 2 represents the traversal of the rewritten path.