Common Logic for an RDF Store

Robert MacGregor, Ph.D.
Franz Inc.
Oakland, CA
bob.macgregor@gmail.com

***Abstract* — The advent of commercial tools that support reasoning and management of RDF data stores provides a robust base for the growth of Semantic Web applications. There is as yet no analogous set of tools and products to support advanced logic-based applications. This article examines issues that arise when seeking to combine the expressive power of Common Logic with the scalability of an RDF store.**

***Index Terms* — Common Logic, RDF, Semantic Web, higher order logic.**

I. Introduction

Franz Inc is researching the possibility of implementing a Common Logic [1] parser and query processor for our RDF [2] store, AllegroGraph. There is a very wide scope for how large a subset of the language is implemented, and more significantly, what kinds of reasoning are supported. Here we discuss several of the issues and possibilities.

A primary goal for marrying Common Logic (CL) to AllegroGraph is to achieve scalable logical inference over a dynamic fact base. We believe the AllegroGraph infrastructure is well suited for crafting a scalable reasoner, however, expressive logics are inherently non-scalable, and there are severe trade-offs that must be made to achieve reasoning over billions of statements.

As part of the presentation we will discuss our proposal for implementation and requirements for the intelligence community. We will discuss a number of the tradeoff as described below.

II. Common Logic Implementation

A full-fledged implementation would include the following features:

(i) A user-friendly query language, based on CL, that supports arbitrary boolean expressions in the "where" clause (here, we are imagining augmenting a SQL-like select-from-where syntax).

(ii) Optional support for logic operators that make the closed-world assumption and unique id assumptions. We include these because classical negation and universal quantification operators are inherently non-scalable.

(iii) A CL-based rule language.

(iv) CL definitions.

(v) Extensible operators.

Just having a CL-based query language would be a big improvement on the current state of RDF-based tools. Unlike SPARQL (introduced below), it would have a "real" (model-theoretic) semantics, it would have clean syntax that assumes a calculus-like rather than an algebraic formulation of clauses, it would be expressive, and it wouldn't break your fingers when you type it.

The inclusion of a definition language that subsumes OWL (easy) would allow for a calculus that spans the range of RDF-based languages with a single syntax.

There are several "sweet spots" that could be supported; a sweet spot being a language subset that supports sound and complete reasoning while being at least moderately scalable. We will note these as we examine various trade-offs.

III. A Baseline System

The baseline is a simple query language with atomic ground assertions. Here, we explicitly exclude the possibility of rules or definitions. Such a language would allow arbitrary CL expressions to be evaluated against the fact base. However, both universal quantifiers and classical negation operators will always evaluate to false (or unknown) in this scheme; the former due to the absence of any kind of "closure" operator, and the latter due (notionally) to the absence of statements that can logically contradict one another.

The simplest way to "achieve closure" is to admit operators that define local closed world assumptions. For example, a negation-as-failure operation assumes that the facts within the scope of a single query are situationally-complete. A closed-world universal quantifier makes a similar assumption. Both of these operators are completely scalable. In other words, we can add them without reducing our expectations on how scalable our language is. The strategy of embedding the closure within a language operator, rather than, say, within a predicate, minimizes the scope of the closures, and allows both open and closed-world reasoning to be applied to the same model.

Alternatively, one could permit assertion of OWL-like operators (e.g., max-cardinality and all-values-from) to achieve closure. Introducing these operators immediately eliminates the possibility of scalable reasoning, or complete reasoning, or both. Here, we are discounting databases where large numbers of asserted and derived have been laboriously loaded and then compiled, yielding a base that melts on the first update; a "dynamic" scalable application will support real-time updates as a matter of course.

Our baseline would not be complete without a transitivity operator. Transitive closure is the most important of all the classes of inference. Simplest would be to include the equivalent of an 'owl:TransitiveProperty' declaration, but practical experience has shown that the addition of a transitive closure operator to the query language syntax has important benefits. Specifically, it is useful to be able to define transitive closures over compound binary expressions; something that OWL can't do. AllegroGraph includes specialized accelerators for computing transitive closures.

IV. Adding Rules

The addition of Horn-like rules significantly increases the utility of the language. Here we face a choice. If our rules are recursive, then syntactic constraints must be placed upon both the heads and bodies of our rules if we are to retain completeness. This results in a Prolog-like semantics, with a CL syntax. The accompanying reasoning can be made moderately scalable.

Alternatively, we can degree that our rules are non-recursive. Here, we restrict our rules to having atomic heads, but we can permit arbitrarily expressive tails. This scheme is both scalable and complete. We know this because these rules do not actually increase the expressive power of the query language. Instead they are a convenience (a major convenience). There are relatively few examples of systems built using non-recursive rules; however, limited practical experience has revealed that most recursive Horn rules can be reformulated into equivalent non-recursive rules combined with an (expressive) transitive closure operator. The most serious drawback to this scheme (non-recursive rules) is that it is theoretically uninteresting. There is nothing semantically to write about, so there are no papers on the subject.

Finally, one could design a system that combines recursive and non-recursive rules. This is a quite viable option. The only caveat is that only highly-disciplined users are likely to reformulate as many rules as possible into non-recursive equivalents. The benefits of doing so would be orders of magnitude increases in query performance, but your average user might not master the technique.

V. Definitions

We face another choice when we add definitions into the mix. If we interpret our definitions as if-and-only-if rules, then we have abandoned hope of scalable inference. Alternatively, we can apply an asymmetric (if but not only-if) interpretation to Horn-like definitions to achieve an expressivity equivalent to Horn rules These are superior to one-directional rules, because the only-if portion can be reserved for constraint-checking/data validation. A single syntax should suffice for either interpretation of a definition (asymmetric or bi-directional); one can envision using a single set of definitions for both scalable inference and small-scale but rich inference.

VI. Inference

Tableaux-based reasoners appear to be inherently non-scalable over dynamic databases. Instead, we focus chiefly on rule-based reasoners. There are three basic classes of rules: (1) backward-chaining rules, (2) forward-chaining rules, and (3) rewrite rules. Backward-chaining rules are the best-behaved. They are relatively insensitive to database updates (cache-busting will occur, but it is manageable) and they are moderately scalable.

Forward-chaining rules are more powerful (from a completeness standpoint) than backward rules. However, some form of truth maintenance is required to manage derived facts, and bitter experience has shown that truth maintenance does not scale. Hence, this option is not viable for large scale applications.

Rewrite rules (also called "triggers") are essentially forward rules that don't bother to clean up after themselves when updates are made. Instead, they are interpreted as having a sematics external to the system. This makes them highly useful, but it is "buyer beware" when it comes to semantics. Rewrite systems have difficulty managing the trigger portion of very expressive rules. For the handling of expressive rewrite rules, we recommend the introduction of "trigger" clauses into the syntax. The assumption is that such rules will fire only when updates to the fact base are detectible by the trigger portion(s) of the rule; other (presumably more expensive) clauses in the rule will not be monitored. Most uses of rewrite rules (e.g., Jess rules) are applied only to modest sized databases.

The extensible operator feature allows arbitrarily complex operators to be added to the language. This allows for exotic operators like "cut" or modals to be added. This is possible because the specialist mechanism includes hooks into the internals of the query executor. High-end inference can be achieved by adding additional operators to the rule engine that include their own logic interpreters.

AllegroGraph's implementation of CL will use the extension mechanism to provide access via CL to its built-in geospatial, temporal and social network analysis features.

VII. Common Logic and RDF

Scalable logic-based applications will most likely be built on top of an RDF triple store. It makes sense to ask what contribution Common Logic can make in this context. In fact, a query language based on Common Logic would have a number of advantages. This is due in part to the fact that SPARQL, the defacto standard in the RDF world, has a number of serious deficiencies that discourage its use for higher-level logic applications.

SPARQL [3] is a W3C-recommended query language for RDF data. It has been designed to enable expressions of common, everyday queries in a style that mimics a syntax used elsewhere to express atomic ground assertions. The majority of developers of RDF stores provide implementations of SPARQL; this has significantly spurred the growth of RDF-based tools and technology.

One serious drawback of SPARQL is that it takes the "kitchen sink" approach to syntax. SPARQL has two "and" operators, two "or operators, and an awkward division between predicates evaluated against the store versus predicates evaluated by other means (e.g., equality, inequality, etc.). Rather than treating the context/graph dimension as simply one additional argument (to a triple), it adds orthogonal syntactic constructs that interleave with the already cumbersome triples and filters. While simple SPARQL queries are fairly readable, when complexities such as disjunctions are utilized, SPARQL queries become very difficult to compose and interpret.

In the logic world, a primary weapon to counter syntactic complexity is to base the semantics of a logic on a small number of primitive operators, and to define the remaining operators as compositions of the primitives. In this case, the bulk of language syntax may be regarded as syntactic sugar; this makes the job of implementing the language much more manageable. This is how, for example, KIF [4] and Common Logic have been defined. SPARQL has taken the opposite approach; it has a large number of different semantic operators, and is defined in terms of a procedural semantics rather than a declarative semantics. That means that the traditional compositional semantics approach cannot be applied to SPARQL.

The combination of bloated syntax and an essentially non-existent semantics means that SPARQL cannot readily server as a foundation for the addition of rules, modal operators, and other higher level constructs. This leaves the field open to competing languages such as Common Logic.

VIII. Expressive Power Examples

In this section, we look at some simple examples where the expressive power of Common Logic can be applied to treat representational problems that are difficult or impossible to solve using a SPARQL-like language. We will use a KIF-like syntax to express our rules.

A common claim made by many RDF advocates is that "the Semantic Web is open world". Practical experience indicates that this statement is a complete falsehood; in fact, not only are there "pockets" of assertions in most semantic networks best treated using close-world semantics, but these "pockets" tend to be the locus of the highest-valued information. Therefore, a practical Semantic Web language will include constructs to treat close-world models.

Consider the predicate "single", as in "not married". It is conventional to treat the definition of the "single" predicate as the closed-world negation of the predicate "married", e.g.,

```
(<= (married ?p)
     (exists (?s) (spouse ?p ?s)))

(<= (single ?p)
     (not (married ?p)))
```

In other words, if you don't know that a person is married, assume s/he is single. This isn't guaranteed to be true; but its the way that personnel data is utilized a great deal of the time.

The semantics of closed-world negation can either be assumed to attach to the underlying domain model, or to be attributed to a logic operator. In the latter case, since 'not' denotes classical negation, we would replace 'not' by a specific negation-as-failure operator (variously called 'thnot', 'unsaid', etc.) to achieve the desired semantics, e.g.,

```
(<= (single ?p)
     (unsaid (married ?p)))
```

Next, consider universal quantification. The rule below states that you are "off the hook" if all of your children have graduated from college:

```
(<= (off-the-hook ?p)
     (forall (?c)
          (implies (child ?p ?c)
               (graduated-from-college ?c)
```

The trick to evaluating this predicate in a practical domain lies chiefly in determining if the set of children known for an individual Fred constitutes the complete set of Fred's children. This kind of information is typically hard to locate. Instead of looking for a guaranteed answer, it is more typical to query for all of Fred's children, and ask if each of those retrieved has graduated. This answer can be trusted as far, and only as far, as the closed-world assumption holds.

The ability to make closed-world assumptions about sets of entities is critical to many real-world applications. Having a universal quantifier in the language enables this reasoning to be computed endogenously, rather than relegating it to the procedurally-evaluated portion of an application.

One would also like aggregate entities to be treatable within a logic. Here is a (somewhat simplistic) definition of the term "family":

```
(<= (family ?p ?fam)
   ?fam = (setof (?r) (or (spouse ?p ?r)
                          (child ?p ?r))))
```

Query languages such as SQL and SPARQL do not allow for explicit universal quantifiers in their syntax. This has two consequences: (i) it limits the kinds of universal quantification expressible in these languages (SQL has various aggregate operators; SPARQL makes no provisions for universal quantification); (ii) it requires that scope rules for variables be implicit rather than explicit, which works well most of the time, but not always. Here is an example representing a simplification of an application that this author encountered, where the lack of an explicit existential quantifier (and accompanying scoping) made composing the query difficult. The (simplified) problem is to query for two degrees of distance from Kevin Bacon, based on a 'knows' relationship. Here is the query expressed *without* reference to existential quantification:

```
(select ?x (where
      (or (?x = Kevin)
         (and (knows Kevin ?x1)
            (or (?x = ?x1)
               (and (knows ?x1 ?x2)
                  (?x = ?x2))))))
```

The query succeeds only if the variable ?x1 is the same throughout the query. In many quantifier-free languages (e.g., SPARQL) variables in parallel disjuncts can have the same name but not be considered the same variable. This is done for a very good reason; however, it means that we can't be sure how the above query will be evaluated without a detailed inspection. The actual query found in the application was more complex than this, because the entities were related by more than one predicate. If you replace

`(knows ?x1 ?x2)` above by

`(or (knows ?x1 ?x2) (likes ?x1 ?x2)`

then you will have a better approximation of the complexity of the query in the application. Doing so makes the scoping that much more tenuous. In fact, the query language used in the application turned out to have scoping rules that assumed that the variable ?x1 was *not* unique across the query. This made it necessary to rewrite the query, approximately doubling its size. On the other hand, if we have an explicit existential quantifier, none of this "guessing" is necessary:

```
(select ?x (where
     (or (?x = Kevin)
         (exists (?x1)
            (and (knows Kevin ?x1)
              (or (?x = ?x1)
                  (and (knows ?x1 ?x2)
                      (?x = ?x2)))))))
```

Lastly, a host of logic-based applications find it useful (and in a cognitive-sense, "necessary") that the language support n-ary predicates and n-ary functions. The Franz product features a suite of geospatial, temporal, and semantic network reasoners that are best exploited using queries that employ n-ary predicates. The query below evaluates:

Retrieve important people known to Bob who attended a meeting in or near Berkeley, CA in November, 2008.

```
(select (?p)
      (and
      (ego-group bob knows ?group 2)
      (actor-centrality-members
              ?group knows ?p ?importance)
      (participant ?event ?p)
      (instance ?event Meeting)
      (interval-during ?event "2008-11-01"
                              "2008-11-05")
      (contains (geo-box-around
              (location Berkeley) 5 miles)
              (location ?event))))
```

Here the 'ego-group' predicate is a distance-2 Kevin Bacon computation (note how much simpler it is than the previous query).

Another comment on the "Kevin Bacon" query: When the relationship predicate is the same on all layers, then a built-in version of the computation can be expected to execute significantly faster than the same computation phrased in logic. However, the original query referenced a different predicate at the first level than the second, and referenced four different predicates at that second level, so a built-in operator was not available. The moral being that built-ins are not a universal panacea for expressiveness.

This section has surveyed a sample of Common Logic language constructs to suggest that users benefit both by (i) the ability to program a larger portion of their applications within the logic, rather than resorting to procedural manifestations, and (ii) that use of more expressive constructs can reduce the complexity of the resulting rules and queries, making the language more usable by humans.

IX. SUMMARY

Adding a Common Logic interface and interpreter to an RDF store would provide a spectrum of possible benefits. At one end, a careful exploitation of CL features would provide "heightened" versions of semi-conventional query processing, over a dynamic, scalable platform. At the high-end, one can contemplate experimenting with combinations of powerful reasoners operating over relatively small sets of data interacting with the large-scale query engine.

The implementation community for Common Logic needs to produce a target specification that is both "doable" and useful to a significant class of applications. There is a chicken-and-egg component, since one needs to have an expressive language available to appreciate why and how one can use it.

REFERENCES

[1] International Standard ISO/IEC 24707 Information technology — Common Logic (CL): a framework for a family of logic based languages. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c039175_ISO_IEC_24707_2007(E).zip.
[2] RDF/XML Syntax Specification (Revised) W3C Recommendation 10 February 2004. URL: http://www.w3.org/TR/rdf-syntax-grammar/.
[3] SPARQL Query Language for RDF W3C Recommendation 15 January 2008. URL: http://www.w3.org/TR/rdf-sparql-query/.
[4] Knowledge Interchange Format draft proposed American National Standard (dpANS) NCITS.T2/98-004. http://logic.stanford.edu/kif/dpans.html.