An Extensible Natural-Language Query Interface to an Event-Based Semantic Web Triplestore

Richard A. Frost and Shane Peelar

School of Computer Science, University of Windsor,
401 Sunset Ave, Windsor, Ontario, N9B3P4, Canada
{richard,peelar}@uwindsor.ca

Abstract. An advantage of the semantic web is that information can be easily added to existing data. Ideally, it should be possible to query the original and new data, using an extensible natural-language query interface. (NLQI). In order to facilitate the extension of an NLQI, it helps if the query language is based on a compositional semantics, and if the language processor is highly modular. Such an NLQI to an online event-based triplestore has been constructed and is available through a web interface. The NLQI is implemented as an executable specification of a highly modular attribute grammar using parser combinators in a pure functional programming language. The parser/interpreter can handle ambiguous left-recursive grammars and fully dependent synthesized and inherited attributes. This enables the integration of semantic rules with syntactic rules, in the style of Montague Grammars, which facilitates the addition of new constructs to the query language.

Keywords: Natural Language Processing, Natural Language Semantics, Semantic Web, Montague Semantics, Event Semantics

1 Introduction

One of the major advantages of the semantic web is that data on a topic can be added with little knowledge of the way in which existing data is stored. This is particularly the case with reified semantic web triplestores, where people can add many properties such as the time, location, and implement used, to a triple such as "<hall> <discovered> <phobos>". For example, consider an event-based triplestore containing the following triples, amongst others, where we use bare names for URIs:

```
<event1030> <type> <discovery> .
<event1030> <subject> <hall> .
<event1030> <object> <phobos> .
```

Additional data can be added as follows:

```
<event1030> <date> <1877> .
<event1030> <implement> <refractor_telescope_1> .
<event1030> <location> <us_naval_observatory> .
```

Ideally, it should be possible to query these and other triples, using an extensible natural-language query interface. (NLQI).

In order to facilitate the extension of an NLQI, it helps if the query language is based on a compositional semantics such as Montague Semantics (MS) [8] or a version of it, and if the language processor is highly modular. Such an NLQI to an online event-based triplestore has been constructed and is available through a web interface, which is discussed in section 2. The NLQI can accommodate common and proper nouns, adjectives, conjunction and disjunction, nested quantifiers, intransitive and transitive n- ary verbs, and chained complex prepositional phrases (PPs). The NLQI is implemented as an executable specification of a attribute grammar (EAG) using parser combinators in the pure functional programming language Haskell. Our parser/interpreter can handle ambiguous left-recursive grammars and fully dependent synthesized and inherited attributes. We begin in section 2 with a demonstration of our NLQI. In section 3, we discuss our modification to Montague Semantics and our compositional event semantics. In section 4 we discuss quantifier scoping. In section 5, we describe our implementation of the NLQI as an EAG. In section 6, we discuss how our interface can be extended. In section 7, we discuss related work. We conclude in section 8.

The motivation for this work is to rekindle an interest in Montague-like compositional semantics for query processing. Compositional semantics have many benefits which have been exploited in many computing applications, including the facilitation of extensibility.

2 A Demonstration of our NLQI

We have built a small "Solarman" triplestore containing approximately 22,000 facts about the moons in our solar system, the planets they orbit, and the people who discovered them, when, where and with which telescope. Our NLQI can answer many questions with respect to the "Solarman" triplestore, but no other questions yet. We have installed our NLQI to Solarman on a server and also on a home wireless router to ensure that our approach requires only minimal computing power (the answer time on the router is as fast as on the server.) The triplestore is stored using the Virtuoso semantic web software which supports a SPARQL endpoint. Our NLQI is accessible through the following web page, which contains example queries, and lists of words and categories of words that can be used in queries:

http://speechweb2.cs.uwindsor.ca/solarman4/demo_sparql.html

Event-based triples consist of three fields: an event identifier, a relationship type, and a type or entity identifier. For example, the facts that Hall discovered Phobos in 1877 using a refractor telescope at the US Naval Observatory can be encoded with the triples above. The extra facts, in addition to the type, subject and object of the event enable evaluation of queries containing PPs.

The triples in our triplestore can be accessed by following the link to our prototype NLQI. Our processor uses basic SPARQL retrieval commands to retrieve sets of entities and events of a given type, and entities which are the actors/properties of a given event. Our NLQI takes advantage of the optimized retrieval operators in the SPARQL endpoint to our triplestore. The functions defined by sets of events are computed in Haskell as needed during the evaluation of the queries.

The following queries illustrate that quantifier scoping is leftmost outermost:

- every telescope was used to discover a moon: *True*
- a moon was discovered by every telescope: *False*
- a telescope was used by hall to discover two moons: *True*
- which moons were discovered with two telescopes: *halimede laomedeia sao themisto*
- hall discovered a moon with two telescopes: *False*
- who discovered deimos with a telescope that was used to discover every moon that orbits mars: *hall*; *hall*
- who discovered a moon with two telescopes: *nicholson science_team_18 science_team_2*
- how was sao discovered: *blanco_telescope canada-france-hawaii_telescope*
- how many telescopes were used to discover sao: 2
- who discovered sao: *science_team_18*
- how did science_team_18 discover sao: *blanco_telescope canada-france-hawaii_telescope*
- which planet is orbited by every moon that was discovered by two people: *saturn*; *none*
- which person discovered a moon in 1877 with every telescope that was used to discover phobos: *hall*; *none*

3 Our Modification to Montague Semantics

3.1 Modifying MS to a "set-based" MS

We use sets of entities rather than characteristic functions (unary-predicates) in order to make the implementation of the semantics computationally tractable:

Original MS [8]:

$$
\begin{aligned}
& \|every\ moon\ spins\| \\
\Longrightarrow\ & (\|every\ moon\|)\ \|spins\| \\
\Longrightarrow\ & (\lambda p\ \lambda q\ \forall x(p\ x \Rightarrow q\ x)\ moon\_pred)\ spins\_pred \\
\Longrightarrow\ & (\lambda q\ \forall x\ (moon\_pred\ x \Rightarrow q\ x))\ spins\_pred \\
\Longrightarrow\ & \forall x\ moon\_pred\ x \Rightarrow spins\_pred\ x \\
\Longrightarrow\ & True
\end{aligned}
$$

Requiring *moon_pred* to be applied to all entities in the universe of discourse. In the set-based MS, the denotation of a noun, adjective or intransitive verb is

a set of entities of type denoted by *es*. The denotation of the word "every" is modified to:

$$\begin{aligned} & \|every\ moon\ spins\| \\ \Longrightarrow\ & (\|every\ moon\|)\ \|spins\| \\ \Longrightarrow\ & (\lambda s\ \lambda t\ s \subseteq t)\ moon\_set\ spin\_set \\ \Longrightarrow\ & (\lambda t\ moon\_set \subseteq t)\ spin\_set \\ \Longrightarrow\ & moon\_set \subseteq spin\_set \\ \Longrightarrow\ & True \end{aligned}$$

Which is computationally tractable if the representations of sets of moons and things that spin can be readily retrieved. Proper nouns denote functions of type $es \rightarrow Bool$.

$$\|phobos\| = \lambda s\ e_{phobos} \in s$$

Then:

$$\begin{aligned} & \|phobos\ spins\| \\ \Longrightarrow\ & (\lambda s\ e_{phobos} \in s) \\ \Longrightarrow\ & e_{phobos} \in spin\_set \\ \Longrightarrow\ & True \qquad \text{(if Phobos spins.)} \end{aligned}$$

All denotations are modified to use sets rather than characteristic functions.

3.2 Events

In order to accommodate n-ary verbs ($n > 2$) and PPs, we integrate event semantics with MS using ideas from Davidson et al. [7], Rothstein [20] and Champollion [5]. Our basic idea is to modify the above to return sets of pairs (rather than sets of entities) as intermediate results from evaluating the denotations of phrases. Each pair contains an entity paired with a set of events. In some cases, the set of events can be thought of as justifying why the entity is in the result. For example the result of evaluating the phrase "discover phobos" contains the pair $(e_{hall}, \{ev_{1030}\})$.

3.3 An Explicit Denotation for Transitive Verbs

MS does not provide an explicit denotation for transitive verbs and deals with them using a syntactic manipulation rule at the end of rewriting the expressions containing them (see page 216 of [8]). The basic idea underlying our approach is to regard each n-ary verb as defining $n^2 - n$ functions between the entities in the n roles in the events associated with that verb.

A Denotation of transitive verbs without events in the semantics In a simple version of our semantics, 2-place transitive verbs denote functions from a possibly empty list of at most one termphrase to a set of pairs of type (e, es) where e is an entity and es is a set of entities. The function computes the answer by using data that is retrieved from the datastore as needed. Consider the verb "discover" which we use in our examples. In our triplestore, each discovery event has 5 roles: *subject* (agent), *object* (theme), *implement*, *year* and *location*. The triplestore defines 20 binary relations between these 5 roles: *subject* → *object*, *subject* → *implement*, *subject* → *year*, *subject* → *location*, *object* → *subject*, etc. For example, the facts that Hall discovered Phobos and Deimos and Kuiper discovered Miranda and Nereid, are represented as follows:

$$\begin{aligned}&\mathit{discover\_rel}_{\mathit{subject}\rightarrow\mathit{object}} = \\ &\{(e_{hall}, e_{phobos}), (e_{hall}, e_{deimos}), (e_{kuiper}, e_{miranda}), (e_{kuiper}, e_{nereid}), \mathit{etc}\ldots\}\end{aligned}$$

For every n-ary verb there are $n^2 - n$ binary relations represented by the events associated with that verb. Each binary relation can be converted to a function, by "collecting", into a set, all values in the codomain that are associated with each value in the domain of the relation, and creating a pair consisting of the value from the domain paired with that set. In 2016, Peelar called this induced function the *Function Defined by a Binary Relation (FDBR)* [19]:

$$\text{FDBR}(\mathit{rel}) = \{(x, \mathit{image}_x) \mid (\exists e)\ (x, e) \in \mathit{rel}\ \&\ \mathit{image}_x = \{y \mid (x, y) \in \mathit{rel}\}\}$$

For example, the function defined by the $\mathit{discover\_rel}_{\mathit{subject}\rightarrow\mathit{object}}$ relation above is:

$$\begin{aligned}&\text{FDBR}(\mathit{discover\_rel}_{\mathit{subject}\rightarrow\mathit{object}}) = \\ &\qquad\{(e_{hall}, \{e_{phobos}, e_{deimos}\}), (e_{kuiper}, \{e_{miranda}, e_{nereid}\}), \mathit{etc}\ldots\}\end{aligned}$$

In such functions, we shall refer to the first value in a pair as the "subject" and the value in the second place as the "set of objects". Consider the query "who discovered phobos": the function which is the denotation of "discovered" computes $\text{FDBR}(\mathit{discover\_rel}_{\mathit{subject}\rightarrow\mathit{object}})$ and then applies the function which is the denotation of "phobos" to the set of objects in every pair $(\mathit{subj}, \mathit{objs})$ which is a member of $\text{FDBR}(\mathit{discover\_rel}_{\mathit{subject}\rightarrow\mathit{object}})$. For every pair which returns a value of *True*, the subject of the pair is added to the result. The final result of "discovered phobos" is a set of pairs, each consisting of every subject which was mapped by $\text{FDBR}(\mathit{discover\_rel}_{\mathit{subject}\rightarrow\mathit{object}})$ to a set of objects which contains e_{phobos}. That is, every entity that discovered phobos paired with the set of events which justify that entity being in the answer. The answer to this example query includes e_{hall}. Similarly, the query "who discovered a moon" is processed analogously to the above, with the denotation of "a moon" being applied to the set of objects in every pair in $\text{FDBR}(\mathit{discover\_rel}_{\mathit{subject}\rightarrow\mathit{object}})$, and if *True*, the associated subject is added to the result. Every entity that discovered a moon is in the result.

If no termphrase follows the transitive verb, all subjects of pairs that are in FDBR($discover\_rel_{subject \rightarrow object}$) are returned as the answer. For example, the answer to the query "who discovered" is the set of all entities who discovered anything.

Denotation of transitive verbs with events In order to take advantage of the extra knowledge represented by events, we modify the above so that the denotation of a transitive verb is a function from a list of at most one termphrase and a possibly empty list of PPs to a set of pairs of type (e, evs) where e is an entity and evs is a set of events. The function first computes a discover relation from subjects of discover events to those events:

$$discover\_rel\_evs_{subject} = \{(e_{hall}, ev_1), (e_{hall}, ev_2), (e_{kuiper}, ev_3), (e_{kuiper}, ev_4), etc \ldots\}$$

The FDBR of this binary relation is then computed:

$$\text{FDBR}(discover\_rel\_evs_{subject}) = \{(e_{hall}, \{ev_1, ev_2\}), (e_{kuiper}, \{ev_3, ev_4\}), etc \ldots\}$$

Consider the query "who discovered phobos": the function which is the denotation of "discovered" computes FDBR($discover\_rel\_evs_{subject}$) and then applies the function which is the denotation of "phobos" to the set of objects of the events in every pair $(subj, evs)$ which is a member of FDBR($discover\_rel\_evs_{subject}$). For every pair which returns *True*, the subject and set of events is added to the resulting denotation. The final resulting denotation of "discovered phobos" is a set of pairs consisting of subjects which were mapped by $discover\_rel\_evs_{subject}$ to a set of events whose objects contains e_{phobos}. The answer to this example query includes $(e_{hall}, \{ev_{1030}\})$. Similarly, the query "who discovered a moon" is processed analogously to the above, with the denotation of "a moon" being applied to the set of objects from the set of events in every pair in FDBR($discover\_rel\_evs_{subject}$), and if *True*, the pair is added to the result. If no termphrase or PP follows the transitive verb, all pairs in FDBR($discover\_rel\_evs_{subject}$) are returned as the answer . For example, the answer to the query "who discovered" is the set of all entities who discovered anything, paired with the set of events of type discovery in which they were the subject.

Dealing with prepositional phrases We begin by noting that we treat passive forms of verbs, such "discovered by hall" similarly to "discovered with a telescope" [19]. Prepositional phrases such as "with a telescope" are treated similarly to the method described in Section 3.3 except that the termphrase following the preposition is applied to the set of entities that are extracted from the set of events in the FDBR function, according to the role associated with the preposition. The result is a "filtered" FDBR which is further filtered by subsequent PPs. For example, consider the query:

"who discovered in 1948 and 1949 with a telescope" → kuiper

The calculation here involves computing FDBR($discover\_rel\_evs_{subject}$), then filtering it with the denotation of "with a telescope".

Choosing the FDBR to compute The denotation of a verb, for example "discover", needs to know which FDBR to compute before PPs are applied. For example, the query "what was used to discover two moons" needs FDBR($discover\_rel\_evs_{implement}$), whereas "who discovered two moons" needs FDBR($discover\_rel\_evs_{subject}$). In our approach, the choice is made depending on the context in which the verb appears. The denotation of a transitive verb contains the "active" and "passive" properties to be queried depending on the verb voice, along with the event type that corresponds to the underlying relation. The grammar determines whether a transitive verb is used in the active or passive voice and selects the corresponding property in the denotation to form the domain of the FDBR. In the above examples, when "used" is in the active voice, it selects the "subject" property, but if it is in the passive voice, it selects the "implement" property. In both cases, the "type" property of the events that the FDBR is built from is "discover_ev".

4 Quantifier Scope

We have integrated a Montague-like [8] compositional semantics with our own version of event semantics. There has been much debate by linguists concerning the viability of integrating compositional and event semantics, particularly with respect to quantifier scope (see for example, Champollion [5] who argues that analysis of quantifier scope does not pose any special problems in an event semantic framework and presents an implementation of a quantificational event semantics that combines with standard treatments of scope-taking expressions in a well-behaved way. The following examples in subsection 4.1, which have been tested with our interface, suggest that our approach returns appropriate results for scope-ambiguous queries. In fact, the answer returned is exactly what is expected if the queries are treated as having leftmost, outermost quantifier scope. Below each query-answer pair, we briefly explain how our system computes the answer.

4.1 Example Queries Illustrating Quantifier Scoping

a. every moon that orbits mars and was discovered with a telescope was discovered by a person – *True*

The evaluation begins by retrieving all of the "moon" entities and then intersecting this set with the set returned by evaluating $\|orbits\ mars\|$, which is obtained by use of the function $f\_orbit_{subject \to object}$ from subjects to the set of objects which they orbit. The function that is the denotation of "mars" is then applied by the function denoted by "orbit" to all sets of objects that are in the range of the function $f\_orbit_{subject \to object}$. This returns the set of subjects that orbit Mars. Then, This set is intersected with the set of all moons that were discovered with a telescope (which is computed using the function $f\_discover_{object \to implements}$).The set resulting from this intersection is then passed as the first argument to the denotation of "every". The second

argument to $\|every\|$ is the set obtained by evaluating the phrase "discovered by a person" which is computed by use of the function $f\_discover_{object \rightarrow subject}$ from objects discovered to subjects who discovered them. The function that is the denotation of "a person" is applied by the function denoted by "discover" to all sets of subjects that are in the range of the function $f\_discover_{object \rightarrow subject}$ This returns the set of objects that were discovered by a person. $\|every\|$ applies the subset operator to the two arguments and returns *True* if and only if the set of objects $\|moon\ that\ orbits\ mars\ and\ was\ discovered\ with\ a\ telescope\|$ is a subset of $\|discovered\ by\ a\ person\|$. In our triplestore, this is the case.

b. every moon that orbits mars and was discovered by a person was discovered with a telescope – *True*

Similar explanation to that for query a.

c. every moon that orbits Neptune was discovered by a person or a team – *True*

Scoping does not require the person or the team to be the same for all discoveries of the moons that orbit Neptune. $\|discovered\ by\ a\ person\ or\ a\ team\|$ returns everything that was discovered by any person or any team. This set is tested by $\|every\|$ to see if it includes all of the entities returned by $\|moon\ that\ orbits\ Neptune\|$.

d. a telescope or voyager_2 was used to discover every moon that orbits Neptune – *False*

No single telescope nor Voyager 2 was used to discover every moon that orbits Neptune

e. every moon that orbits neptune was discovered with a telescope or voyager_2 – *True*

Voyager 2 or at least one, not necessarily the same, telescope was used to discover each of the moons that orbit Neptune.

f. every moon that was discovered with a telescope was discovered by hall – *False*

Some moons were discovered with a telescope but not discovered by Hall.

g. every moon that was discovered by hall was discovered with a telescope – *True*

Hall used a telescope in all of his discoveries of moons.

Our approach appears to be consistent with the "Scope Domain Principle" described by Landman [14]. That is, all quantificational noun phrases must take scope over the event argument. For example, in our semantics, the answer to the query "hall discovered every moon" is computed by checking to see if, for every moon m, there exists an event of type discovery, with subject Hall and object m. Our approach does not compute the answer to "cumulative" readings of queries such as "which moons were discovered by two telescopes(used simultaneously)".

4.2 Example Queries Illustrating the Scoping of Chained Prepositional Phrases

The following examples illustrate how queries with chained PPs are answered. It should be noted that Halimede, Laomedeia, Sao and Themisto are the only moons that were discovered using two telescopes separately (see queries a to e) and that Nicholson used two telescopes to discover a total of 4 moons, but did not discover any one moon with two telescopes (see queries g to i).

a. which moons were discovered with two telescopes:
   halimede laomedeia sao themisto
b. who used two telescopes to discover a moon:
   nicholson science_team_18 science_team_2
c. who discovered sao: *science_team_18*
d. who discovered themisto: *science_team_2*
e. which moon was discovered by science_team_18 with two telescopes:
   halimede laomedeia sao
f. what was used to discover sao:
   blanco_telescope canada-france-hawaii_telescope
g. what did nicholson discover with two telescopes:
   sinope lysithea carme ananke
h. which moon was discovered by nicholson with two telescopes: *none*
i. which moon was discovered by nicholson with one telescope:
   ananke carme lysithea sinope
j. how was sinope discovered: *refractor_telescope_2*
k. how was carme discovered: *hooker_telescope*
l. how was ananke discovered: *hooker_telescope*
m. how was lysithea discovered: *hooker_telescope*
n. what did nicholson discover with one telescope: *nothing*
o. what did nicholson discover with a telescope:
   sinope lysithea carme ananke

Since Nicholson used multiple telescopes to discover multiple objects, n) returns *nothing*. On the other hand o) relaxes this restriction, yielding the expected result (see Section 8).

5 Implementation

We built our query processor as an executable attribute grammar using the X-SAIGA Haskell parser- combinator library package. The *collect* function which converts a binary relation to an FDBR is one of the most compute intensive parts of our implementation of the semantics. However, in Haskell, once a value is computed, it can be made available for future use. We have developed an algorithm to compute FDBR(*rel*) in $O(n\ log\ n)$ time, where n is the number of pairs in *rel*. Alternatively, the FDBR functions can be computed and stored in a cache when the NLQI is offline. Our implementation is amenable to running on

low power devices, enabling it for use with the Internet of Things. A version of our query processor exists that can run on a common consumer network router as a proof of concept for this application. The use of Haskell for the implementation of our NLQI has many advantages, including:

1. Haskell's "lazy" evaluation strategy of Haskell only computes values when they are required, enabling parser combinator libraries to be built that can handle highly ambiguous left-recursive grammars in polynomial time. The accommodation of left recursive grammars simplifies the integration of semantic and syntactic rules in the EAGs, enabling the query processor to be highly modular and extensible.
2. The higher-order functional capability of Haskell allows the direct definition of higher-order functions that are the denotations of some English words and phrases. For example: *termand* $s\ t = (\lambda v)\ s\ v\ \&\ t\ v$
3. The ability to partially apply functions of n arguments to 1 to n arguments allows the definition and manipulation of denotation of phrases such as "every moon", and "discover phobos".
4. The availability of the *hsparql* [26] Haskell package enables a simple interface between our semantic processor and SPARQL endpoints to our triplestores.

6 Extensibility

A contribution of this paper is to raise awareness of the importance of extensibility of NLQIs to the semantic web. We use the term "extensibility" in the sense that it is used in Software Engineering, meaning the extent to which the implementation takes future growth into consideration, and a measure of the ability to extend the NLQI and the level of effort required to implement the extension.

6.1 Design for extensibility

A number of design decisions facilitate future extension of our NLQI:

1. Our query processor is implemented as a highly-modular executable specification of an attribute grammar (AG). AGs were introduced by Knuth [13] and are widely used to define both the syntax and semantics of programming languages. Each syntax rule has one or more attribute rules associated with it. The attribute rules define how the value of synthesized and inherited attributes of the non-terminal defined by the associated syntax rule are computed from attribute values of the terminals and non-terminals that appear on the right-hand side of the syntax rule. There is a close similarity between AGs and Montague Grammars (MGs), although they were developed independently by a Computer Scientist and a linguist respectively. An executable attribute grammar (EAG) is an AG whose defined language processor is implemented in a programming language such that the program code for the language processor closely resembles the textbook notation for the AG defining the language to be processed. EAGs are ideally suited for implementation of language interpreters for MGs.

2. Our semantics is based on the highly modular and compositional semantics. The similarity of MGs and AGs suggested to us that it should be comparatively easy to implement a Montague-style natural-language query processor as an executable attribute grammar.
3. The dictionary in the Haskell code to facilitate the addition of new words and categories of words to the query language. Our NLQI Haskell code can be accessed at:

   `http://speechweb2.cs.uwindsor.ca/solarman4/src/`

   The code contains a dictionary consisting of entries such as the following:

   `("person", Cnoun, [NOUNCLA_VAL $ get_members "person"])`

   Which defines the word "person" to be a common noun (`cnoun`) whose meaning is a list of attributes, comprising one attribute of type NOUNCLA_VAL whose value is a list of entities extracted from the triplestore by the `get_members` function which returns all entities that are subjects of events of type `member` and whose object is a "`person`". Our parser combinators include a combinator that creates interpreters for different categories of terminals. For example:

   `cnoun = memoize_terminals_from_dictionary Cnoun`

   The combinator `memoize_terminals_from_dictionary` scans the dictionary and creates the interpreter `cnoun` (for the terminal category of common nouns) by "orelsing" all of the basic interpreters that it constructs for words in the first field of every triple in the dictionary that has the "constructor" `Cnoun` in the second field. The list of attributes in the third field is integrated into each basic interpreter constructed. The resulting interpreter for the syntactic category is memoized so that its results can be reused in any subsequent pass over the query string by the same interpreter. The query language can be easily extended with new words and new categories of words by simply adding new entries to the dictionary. Note that only bare names need to be used in the dictionary, as the first part of the URI is added by the combinator that makes the basic interpreter for that word.
4. Our EAG implementation is such that individual parsers can be applied to phrases of English rather than whole queries. This allows us to define new words in terms of existing phrases for which we have defined an interpreter. For example:

   `discoverer = meaning_of nouncla "person who discovered something"`

5. Construction of the interpreter as an EAG accommodates ambiguous and left-recursive grammars greatly facilitates the extension of the query language to include new constructs. When grammars are converted to non-left-recursive form (which is often the case when modular top-down parsers are used) can complicate the specification of semantic rules. For example, the specification of the syntax and associated semantic rules for converting a bit string to its decimal value is much easier if the grammar chosen is left recursive.

6.2 Examples of extending the NLQI

We have given examples, in sub-section 6.1 of how we can add single words such as "person" and "discoverer" to the query language by simply adding an entry to the in-program dictionary. The query Language can also be easily extended with new language constructs by adding new syntax rules to the EAG together with their associated attribute rules. For example, suppose that we want to be able to ask questions such as "tell me all that you know about hall discovering a moon that orbits mars" The phrase "hall discovering a moon that orbits mars" could be processed using the interpreter for questions which would return the set of two events where Hall discovered Phobos and Deimos. The meaning of the phrase "tell me all that you know about" could be designed so that, for each event, a string could be generated: "hall discovered phobos with refractor_telescope_1 in 1987 at the us_naval_observatory" and also any other data that had been added about ev_{1030} Another type of question could be "who discovered which moons". The meaning of the word "which" could be changed temporarily to that of "a" and the question "who discovered a moon" answered. The resulting FDBR could be returned from the latter question and then used to generate pairs of people and the list of moons they discovered as answer to the original query.

Adding superlatives and graded quantifiers The second contribution of this paper is to recognize that each FDBR contains more information than we have taken advantage of so far. For example, in computing the answer to the query "who discovered every moon". We consider each pair $(subj, objs)$ in FDBR($discover\_rel_{subject \rightarrow object}$) independently and apply the meaning of "every moon" to the set *objs* in order to determine if the subj should be in the answer. However, for a question such as "who discovered the most moons that orbit mars" the whole of the FDBR needs to be processed so that the result contains the subject with the most events representing that subject discovering a moon that orbits mars. This requires the addition of "the most" to the set of quantifiers and the appropriate modification to the denotation of transitive verbs to take advantage of the information available in the appropriate FDBR. A similar modification could be made to accommodate queries containing the words "earliest" "most recently", using FDBR($discover\_rel_{object \rightarrow year}$) etc., and queries such as "which telescope was used to discover most moons" using FDBR($discover\_rel_{implement \rightarrow object}$) , etc.

7 Related Work

Orakel [6] is a portable NLQI which uses a Montague-like grammar and a lambda-calculus semantics to analyze queries. Our approach is similar to Orakel in this respect. However, in Orakel, queries are translated to an expression of first-order logic enriched with predicates for query and numerical operators. These expressions are translated to SPARQL or F-Logic. Orakel supports negation, limited quantification, and simple prepositional phrases. Portability is achieved

by having the lexicon customized by people with limited linguistic expertise. It is claimed that Orakel can accommodate n-ary relations with $n \geq 2$. However, no examples are given of such queries being translated to SPARQL.

YAGO2 [11] is a semantic knowledge base containing reified triples extracted from Wikipedia, WordNet and GeoNames, representing nearly 0.5 billion facts. Reification is achieved by tagging each triple with an identifier. However, this is hidden from the user who views the knowledge base as a set of "SPOTL" quintuples, where T is for time and L for location. The SPOTLX query language is used to access YAGO2. Although SPOTLX is a formal language, it is significantly easier to use than is SPARQL for queries involving time and location (which in SPARQL would require many joins for reified triplestores). SPOTLX does not accommodate quantification or negation, but can handle queries with prepositional aspects involving time and location. However, no mention is made of chained complex PPs.

Alexandria [25] is an event-based triplestore, with 160 million triples (representing 13 million n-ary relationships), derived from FreeBase. Alexandria uses a neo-Davidsonian [17] event-based semantics. In Alexandria, queries are parsed to a syntactic dependency graph, mapped to a semantic description, and translated to SPARQL queries containing named graphs. Queries with simple PPs are accommodated. However, no mention is made of negation, nested quantification, or chained complex PPs.

The systems referred to above have made substantial progress in handling ambiguity and matching NL query words to URIs. However, they appear to have hit a roadblock with respect to natural-language coverage. Most can handle simple PPs such as in "who was born in 1918" but none can handle chained complex PPs, containing quantifiers, such as "in us_naval_observatory in 1877 or 1860". There appear to be three reasons for this: 1) those NLQIs that were designed for non-reified triplestores, such as DBpedia, do not appear to be easily extended to reified triplestores that are necessary for complex PPs. 2) those NLQIs that were designed for non-reified or reified triplestores, and which translate the NL queries to SPARQL, suffer from the fact that SPARQL was originally designed for non-reified triplestores. Although SPARQL was extended to handle "named graphs" [3] which support a limited form of reification but appear to be suitable only for provenance data. SPARQL was also extended to accommodate triple identifiers. 3) The YAGO2 system is the only system that has an NLQI for a reified triplestore that does not translate to SPARQL. However, YAGO2 can only accommodate PPs related to time and location and does not support quantification.

Blackburn and Bos [2] implemented lambda calculus with respect to natural language, in Prolog, and [22] have extensively discussed such implementation in Haskell. Implementation of the lambda calculus for open-domain question answering has been investigated by [1]. The SQUALL query language [9, 10] is a controlled natural language (CNL) for querying and updating triplestores represented as RDF graphs. SQUALL can return answers directly from remote triplestores, as we do, using simple SPARQL-endpoint triple retrieval commands.

It can also be translated to SPARQL queries which can be processed by SPARQL endpoints for faster computation of answers. SQUALL syntax and semantics are defined as a Montague Grammar facilitating the translation to SPARQL.SQUALL can handle quantification, aggregation, some forms of negation, and chained complex prepositional phrases. It is also written in a functional language. However, some queries in SQUALL require the use of variables and low-level relational algebraic operators (see for example, the queries on page 118 of [10]).

8 Concluding Comments

We have presented a compositional event semantics for computing the answers to English questions, and have shown how it can be used to query a remote event-based triplestore. We are currently working on three enhancements: 1) scaling up the NLQI to work with triplestores containing millions of events, and 2) increasing the coverage of English to accommodate negation, fusion events where roles can be assigned more than one value, and 3) modifying our approach of PPs to more consistently handle chained PPs.

Extensible NLQIs are necessary if the potential of the semantic web is realized and new data is added to existing triplestores by people who may not have been involved in the creation of those triplestores. We hope that other researchers who are familiar with Haskell will download and experiment with the software that we have developed; all of which is available through the links that we have provided. Our event-based triplestore is also available for remote access at from the URL links that we have given.

Our semantics could be easily extended accommodate very simple negation as in the query "no moon orbits two planets". However, the query "which person orbits no moons?" would not return the correct answer. The reason is that list returned by evaluating the denotation corresponding to "orbits no moons" would only contain entities that orbits something but not a moon. It would not contain entities that orbit nothing. This problem is related to how the "closed world assumption" is implemented. To solve tis problem, we will begin by investigating the methods used in the experimental NLQIs Orakel [6], PANTO [24], Pythia [21], and TBSL [12]. We shall also consider theoretical computational linguistic approaches for dealing with negation and quantification in event-based semantics, e.g. [4].

In this paper, we have not addressed the problems that result from the user's lack of knowledge of the URIs used in the triplestores. Significant progress has been made by others, e.g. [23], in tackling this problem.

Also, we have not considered how our semantics can be automatically tailored for a particular triplestore. We shall begin by considering how Aqualog [16], PowerAqua [15] and Orakel [6] achieve portability with respect to the different ontologies used in different triplestores.

It should be noted that our current treatment of PPs is linguistically naive and suffers from problems with entailment (deriving logical consequences), as discussed by Partee [18], when certain kinds of prepositional or adverbial phrases are

chained together. However, our proposed approach will accommodate many types of queries correctly, and the problem with entailment, which is also problematic for all existing triplestore NLQIs, will also be investigated in future work.

Acknowledgments

This research was funded by a grant from NSERC of Canada.

References

1. Ahn, K., Bos, J., Kor, D., Nissim, M., Webber, B.L., Curran, J.R.: Question answering with qed at trec 2005. In: TREC (2005)
2. Blackburn, P., Bos, J.: Representation and inference for natural language. A first course in computational semantics. CSLI (2005)
3. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: proc. of the 14th int. conf. on World Wide Web. pp. 613–622. ACM (2005)
4. Champollion, L.: Quantification in event semantics. In: Talk given at the 6th int. symp. of cogn., Logic and commun. (2010)
5. Champollion, L.: The interaction of compositional semantics and event semantics. Linguistics and Philosophy **38**(1), 31–66 (2015)
6. Cimiano, P., Haase, P., Heizmann, J.: Porting natural language interfaces between domains: an experimental user study with the orakel system. In: Proc. 12th Intnl. Conf. on intell. User Interfaces. pp. 180–189. ACM (2007)
7. Davidson, D., Harman, G.: Semantics of natural language, vol. 40. Springer Science & Business Media (2012)
8. Dowty, D., Wall, R., Peters, S.: Introduction to Montague Semantics. D. Reidel Publishing Company, Dordrecht, Boston, Lancaster, Tokyo (1981)
9. Ferre, S.: Squall: A controlled natural language for querying and updating rdf graphs. In: proc. of CNL 2012. pp. 11–25. LNCS 7427 (2012)
10. Ferré, S.: Squall: a controlled natural language as expressive as sparql 1.1. In: International Conference on Application of Natural Language to Information Systems. pp. 114–125. Springer (2013)
11. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: Yago2: A spatially and temporally enhanced knowledge base from wikipedia. Artificial intell. **194**, 28–61 (2013)
12. Höffner, K., Unger, C., Bühmann, L., Lehmann, J., Ngomo, A.C.N., Gerber, D., Cimiano, P.: User interface for a template based question answering system. In: Knowledge Engineering and the Semantic Web, pp. 258–264. Springer (2013)
13. Knuth, D.E.: Semantics of context-free languages. Mathematical systems theory **2**(2), 127–145 (1968)
14. Landman, F.: Plurality. Handbook of Contemporary Semantics. Ph.D. thesis, ed. S. Lappin, 425- 457. Blackwell: Oxford (1996)
15. Lopez, V., Fernández, M., Motta, E., Stieler, N.: Poweraqua: Supporting users in querying and exploring the semantic web. Semantic Web **3**(3), 249–265 (2012)
16. Lopez, V., Uren, V., Motta, E., Pasin, M.: Aqualog: An ontology-driven question answering system for organizational semantic intranets. Web semant.: sci., serv. and Agents on the World Wide Web **5**(2), 72–105 (2007)
17. Parsons, T.: Events in the Semantics of English, vol. 5. Cambridge, Ma: MIT Press (1990)

18. Partee, B.H.: Formal semantics. In: Lectures at a workshop in Moscow. http://people.umass. edu/partee/RGGU_2005/RGGU05_formal_semantics. htm (2005)
19. Peelar, S.: Accommodating prepositional phrases in a highly modular natural language query interface to semantic web triplestores using a novel event-based denotational semantics for English and a set of functional parser combinators. Master's thesis, University of Windsor (Canada) (2016)
20. Rothstein, S.: Structuring events: A study in the semantics of aspect, vol. 5. John Wiley & Sons (2008)
21. Unger, C., Cimiano, P.: Pythia: Compositional meaning construction for ontology-based question answering on the semantic web. In: NLDB 2011, LNCS 6716. pp. 153–160 (2011)
22. Van Eijck, J., Unger, C.: Computational semantics with functional programming. Cambridge University Press (2010)
23. Walter, S., Unger, C., Cimiano, P., Bär, D.: Evaluation of a layered approach to question answering over linked data. In: The Semantic Web–ISWC 2012. pp. 362–374. Springer (2012)
24. Wang, C., Xiong, M., Zhou, Q., Yu, Y.: Panto: A portable natural language interface to ontologies. In: The Semantic Web: Research and appl., pp. 473–487. Springer (2007)
25. Wendt, M., Gerlach, M., Düwiger, H.: Linguistic modeling of linked open data for question answering. proc. of interact. with Linked Data (ILD 2012)[37] pp. 75–86 (2012)
26. Wheeler, J.: The hsparql package. In: The Haskell Hackage Repository. http://hackage.haskell.org/package/hsparql-0.1.2 (2009)

All links were last followed on April 3 2018.