A New Approach for Processing Natural-Language Queries to Semantic Web Triplestores

Shane Peelar(✉) and Richard A. Frost

School of Computer Science, University of Windsor, Windsor, ON, Canada
{peelar,richard}@uwindsor.ca

Abstract. Natural Language Query Interfaces (NLQIs) have once again captured the public imagination, but developing them for the Semantic Web has proven to be non-trivial. This is unfortunate, because the Semantic Web offers many opportunities for interacting with smart devices, including those connected to the Internet of Things. In this paper, we present an NLQI to the Semantic Web based on a Compositional Semantics (CS) that can accommodate many particularly tricky aspects of the English language, including nested n-ary transitive verbs, superlatives, and chained prepositional phrases, and even ambiguity. Key to our approach is a new data structure which has proven to be useful in answering NL queries. As a consequence of this, our system is able to handle NL features that are often considered to be non-compositional. We also present a novel method to memoize sub-expressions of a query formed from CS, drastically improving query execution times with respect to large triplestores. Our approach is agnostic to any particular database query language. A live demonstration of our NLQI is available online.

Keywords: Natural language processing · Natural Language Query Interfaces · Compositional Semantics · Event Semantics · Quantification

1 Introduction

This is an extended version of the paper by Frost and Peelar [14] that was presented at WEBIST 2019 in Vienna, Austria. That paper was selected as one of the best papers at WEBIST 2019 and the authors were invited to submit an extended version for publication. In this paper we expand upon the compositionality of our NLQI, including the parsing framework and semantic implementations, we introduce a novel method to accommodate superlatives using compositional semantics, and we discuss a novel approach to memoization and triplestore retrieval. We also significantly expand upon how our NLQI is implemented.

Supported by NSERC of Canada.

A. Bozzon et al. (Eds.): WEBIST 2019, LNBIP 399, pp. 168–194, 2020.
https://doi.org/10.1007/978-3-030-61750-9_8

We begin by describing a Natural Language Query Interface (NLQI) that we have built. We hope that the interface will motivate readers to look into our modifications to MS. In Sect. 2, we explain how our NL Query interface (NLQI) can be accessed through the Web. In Sect. 3, we describe the compositional aspects of our NLQI. In Sect. 4, we describe the Semantic Web triplestore. In Sect. 5 we discuss example queries and their results, including examples of what are often referred to as "non-compositional" features of NL that our NLQI can handle. With each of the examples we provide an informal explanation of how the answer is, or could be, computed. In Sect. 6, we describe the new FDBR data structure which is central to our approach. In Sect. 7 and Sect. 8, we describe how our system accommodates chained prepositional phrases with superlatives. In Sect. 9, we describe how to use our approach with relational databases. In Sect. 10, we provide a system overview and implementation details on how our semantics are realized. Section 11 discusses how our work fits into the framework of existing work in this area. We close with Sect. 12 and Sect. 13 where we discuss future research directions and our conclusions.

Much of our semantics is based on MS. We differ in these ways:

1. We add events to the basic ontological concepts of entities and truth values.
2. Each event has a number of roles associated with it. Each role has an entity as a value.
3. For efficiency, we use sets of entities rather than characteristic functions of those sets as is the case in MS.
4. We define transitive n-ary verbs in terms of sets of events, each with n roles.
5. We compute FDBRs, the novel data structure presented in this paper, from sets of events and use them in the denotations of transitive verbs and in computing results of queries containing prepositional phrases. Although not referred to as an FDBR, the use of relational images in denotations of verbs was first proposed by Frost and Launchbury in 1989 [12].

We hope that this paper reawakens an interest in Compositional Semantics, in particular for NL query processing.

2 How to Access Our NLQI

Our NL interface is accessible via the following URL, and is speech enabled for both voice-in and voice-out in browsers that support the Web Speech API:

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

3 Compositionality

Compositionality is a useful property of any system as it facilitates understanding, construction, modification, extension, proof of properties, and reuse in different situations. When building our system, we tried to make it as compositional as possible: a compositional syntax processor is systematically combined with a compositional semantics.

3.1 The Compositionality of Our Syntactic Processor

Our parser is designed and built using the Haskell programming language, using parser combinators [13]. The approach enables parsers to be constructed as executable specifications of context-free grammars with explicit and implicit left-recursive productions, which is useful for defining grammars for NL. The result of applying our parser is the set of all parse trees for ambiguous grammars. The trees are represented efficiently using a Tomita-style [20] compact graph in which trees share common components.

In 2008, Frost and Hafiz [13] demonstrated that it is possible to efficiently implement context-free parsing using combinators, with their approach having $O(n^4)$ complexity in the worst case and $O(n^3)$ complexity in the average case.

The following example was featured in Frost and Hafiz [13]. To demonstrate use of our combinators, consider the following ambiguous grammar from Tomita [20]:

```
s    ::= np vp | s pp          np   ::= noun  | det noun | np pp
pp   ::= prep np               vp   ::= verb np
det  ::= "a"   | "the"         noun ::= "i"   | "man" | "park" | "bat"
verb ::= "saw"                 prep ::= "in"  | "with"
```

In this grammar, the non-terminal `s` stands for sentence, `np` for nounphrase, `vp` for verbphrase, `det` for determiner, `pp` for prepositional phrase, and `prep` for preposition. It is left recursive in the rules for `s` and `np`. The Haskell code below defines a parser for the above grammar using our combinators `term` (terminal), `<+>` (alternative), and `*>` (sequence) [13]:

```
data Label = S | ... | PREP
s    = memoize S    $ np *> vp <+> s *> pp
np   = memoize NP   $ noun <+> det *> noun <+> np *> pp
pp   = memoize PP   $ prep *> np
vp   = memoize VP   $ verb *> np
det  = memoize DET  $ term "a" <+> term "the"
noun = memoize NOUN
        $ term "i" <+> term "man" <+> term "park" <+> term "bat"
verb = memoize VERB $ term "saw"
prep = memoize PREP $ term "in" <+> term "with"
```

Parsers written in this fashion are highly compositional, and can be easily extended with new rules if needed. Parsers constructed with our combinators have $O(n^3)$ worst case time complexity for non-left-recursive ambiguous grammars (where n is the length of the input), and $O(n^4)$ for left recursive ambiguous grammars. This compares well with $O(n^3)$ limits on standard algorithms for CFGs such as Earley-style parsers [8]. The increase to n^4 is due to expansion of the left recursive non-terminals in the grammar. The potentially exponential number of parse trees for highly-ambiguous input are represented in polynomial space as in Tomita's algorithm.

3.2 The Compositionality of Our Semantics

The semantics on which our system is based is similar to Montague Semantics. All phrases of the same syntactic category have meanings of the same semantic type. The meaning of all words and phrases are functions defined over sets of base terms which are entities, events and Boolean values. The meaning of a complex phrase is obtained by applying the functions which are the meanings of its parts, to each other in an order determined by the syntactic structure of the whole. Our system was easy to construct, and is easy to extend. Additional language features are accommodated by adding their syntactic structure and then defining their semantics by viewing the semantics of words and phrases of the same syntactic category.

3.3 The Compositionality of the Whole NL Processor

Our processor is built as an executable specification of a fully general attribute grammar. Compositional semantic rules are added to each syntactic production using the technique of Frost, Hafiz and Callaghan [13]. The attribute grammar is fully general as it can accommodate left recursive context-free grammars and fully-general dependencies between inherited and synthesized attributes. Haskell allows any computational dependency between attributes to be defined. Also, Haskell's lazy evaluation strategy enables our language processor to be efficient. For example, no attribute computation is carried out until a successful parse has been obtained. We have also developed a variation of memoization using monads [13] in order to reduce the complexity of syntactic and semantic evaluation. In the paper by Frost and Peelar [14] we discuss how we accommodate, using our compositional approach, various English phrases that are often given as examples of non-compositional constructs.

4 The Triplestore that Is Queried

Our NLQI computes answers with respect to an *event-based* Semantic Web triplestore containing data about the planets, the moons that orbit them, and the people who discovered those moons, and when, where and with what implement they were discovered. Briefly, a triplestore is a database of 3-tuples, called triples, that have the form (*subject*, *predicate*, *object*), where *subject*, *predicate* and *object* are Uniform Resource Identifiers (URIs).

An *event-based triple* has a *subject* that identifies an event rather than an entity [19]. In these triples, the *predicate* identifies a *role* through which the *object* participates in the event. That is, an event-based triple (e, r, o) expresses that o participates in e through role r. We call o the event e's "*r property*". For example, in Table 1, "hall" is event "event1045"'s *subject* property. Triplestores consisting of event-based triples are called *event-based triplestores.*

The advantage of event-based triplestores is that additional information about the events and entities participating in those events is immediately available. This is not the case in an entity-based triplestore, where some form of *reification* is necessary to obtain additional information about a fact expressed in a triple. For example, obtaining the location where "hall discovered phobos" in an entity-based triplestore, described by (*hall*, *discovered*, *phobos*), is not possible without reification.

We assume that each event will at minimum contain a role *ev_type* that identifies the type of the event, with the general expectation that events of the same type will contain similar roles. This implies the existence of a schema that describes the types of roles that an event may contain. As a consequence of this, each event could be equally well be represented by a row in a relational database. We discuss this further in Sect. 9.

Going forward, when we refer to the type of an event or set of events, we are referring to their *ev_type* property. Likewise, when we refer to events of a particular type, we are referring to events whose *ev_type* property corresponds to that type. As a shorthand, we use *t*-type events to refer to events with type *t*. For example, "discover" events refers to events that have *ev_type* property "discover".

The triplestore contains triples such as those in Table 1 which represent the event in which `hall` (in the role of *subject*) discovered `phobos` (in the role of *object*) in 1877 (in the role of *year*) with the `refractor_telescope_1` (in the role of *implement*) at the `us_naval_observatory` (in the role of *location*). Events representing set membership are represented as shown in Table 2.

Table 1. Triples describing an event of type "discover" [14]. The full URIs of the events, roles, and entities have been omitted here.

Event	Role	Entity
event1045	subject	hall
event1045	object	phobos
event1045	ev_type	discover_ev
event1045	year	1877
event1045	location	us_naval_observatory
event1045	implement	refractor_telescope_1

Table 2. Triples describing an event of type "membership" [14].

Event	Role	Entity
event1128	subject	galileo
event1128	object	person
event1128	ev_type	membership

The complete triplestore, which contains tens of thousands of triples, is hosted on a remote server using the Virtuoso software [9] and can be accessed by following the link at the beginning of Sect. 2.

5 Example Queries

Our NLQI can answer millions of queries with respect to the triplestore discussed above. The NLQI can accommodate queries containing common and proper nouns, adjectives, conjunction and disjunction, intransitive and transitive verbs, nested quantification, superlatives, chained prepositional phrases containing quantifiers, comparatives and polysemantic words. In the following sections, we provide an informal explanation of how the answer is computed. If a query is syntactically ambiguous, the results from each possible interpretation of the query are denoted with a semicolon.

5.1 Queries Demonstrating the Range of NL Features that Our NLQI Can Accommodate

`phobos spins` ⇒ *True*
`phobos is a moon` ⇒ *True*

The function denoted by `phobos` checks to see if e_{phobos} is a member of the *spin* set, and secondly if e_{phobos} is a member of the *moon* set.

`a moon spins` ⇒ *True*
`every moon spins` ⇒ *True*
`an atmospheric moon exists` ⇒ *True*

The function denoted by "`a`" checks to see if the intersection of the set of *moons* and the set of *spins* is non-empty. The function denoted by "`every`" checks to see if the set of moons is a subset of the *spins* set. The denotations of `a` and `every` that we use are set-theoretic event-based versions of the denotations from MS which uses characteristic functions. The answer to the third query is obtained by checking if the intersection of the *atmospheric* set and the *moon* set is non-empty.

`hall discovered` ⇒ *True*

All of the events of type "discover" are collected together and are checked to see if e_{hall} is found as the subject role value of any of them. If so, *True* is returned.

`when did hall discover` ⇒ *1877*

The *year* property of the events returned by "`hall discover`" (treated as "`hall discovered`") are returned.

`phobos was discovered` ⇒ *True*

All of the events of type "discover" are collected together and are checked to see if e_{phobos} is found as the *object* role value of any of them. If so *True* is returned.

`earth was discovered` ⇒ *False*

Earth was not discovered by anyone, according to our data.

`did hall discover phobos` ⇒ *True*

All of the events of type "discover" are collected together and are checked to create a pair (s, *evs*) for each value of the *subject* property found in the set of events. *evs* is the set of events to which the *subject* property is related through a discovery event. Each pair is then examined to see if the function denoted by the object termphrase (in this case `phobos`) returns a non-empty set when applied to a set (called an *FDBR*, which is described in Sect. 6) generated from the set of *evs* in the pair, and if so the subject of the pair is added to the set which is returned as the denotation of the verbphrase part of the query. The denotation of the termphrase at the beginning of the query is then applied to the denotation of the verbphrase to obtain the answer to the query.

Owing to the fact that our semantics is compositional, the *subject* and *object* termphrases of the query above can be replaced by any termphrases, e.g.:

`a person or a team discovered every moon that orbits mars` ⇒ *True*

`who discovered 2 moons that orbit mars` ⇒ *hall*

"`who`", "`what`", "`where`", "`when`" and "`how`" can be used in place of the *subject* termphrase. Different role values are returned depending on which "*wh*"-word is used in the query:

`where discovered by galileo` ⇒ *padua*

`when discovered by galileo` ⇒ *1610*

`every telescope was used to discover a moon` ⇒ *True* (w.r.t. our data)

`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*

`who discovered deimos with a telescope that was used to discover every moon that orbits mars` ⇒ *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 discovered in 1877` ⇒ *refractor_telescope_1*

`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* (ambiguous because "`by two people`" could apply to "`discovered`" or "`orbited`")

`which person discovered a moon in 1877 with every telescope that was used to discover phobos` ⇒ *hall; none* (ambiguous because "`to discover phobos`" could apply to "`used`" or "`discovered`")

`who discovered in 1948 and 1949 with a telescope` ⇒ *kuiper*

5.2 Queries with "Non-compositional" Structures

We agree that natural language has non-compositional features but believe that the non-compositionality is mostly problematic when the objective is to give a meaning to an arbitrary NL expression (i.e. an NL expression without a context). It is less problematic when answering NL queries. As illustrated below, the person posing the query, or the database or triplestore can provide contexts that help resolve much of the ambiguity resulting from non-compositional features. The advantages of a using a compositional semantics include:

1. The answer to a query is as correct as the data from which it is derived,
2. The meaning of sub phrases within a query can be discussed formally,
3. The query language can be extended such that all existing phrases maintain their original meanings,
4. The definition of syntax and semantics in the compositional semantics can be used as a blueprint for the implementation of the query processor.

Some researchers have provided examples of what they claim to be non-compositional structures in NL. For example, Hirst [16] gives the example of the verb "depart" which he states is not compositional because its meaning changes with the prepositional phrase(s) which follow it, and that the definition of compositionality needs to be modified to include the requirement that the function used to compose the meaning of parts must be systematic. We claim that our semantics for verbs is systematic as the denotations of subject and object termphrases, and the possibly empty list of prepositional phrases following the verb, are treated equally and are all used in the same way to filter the set of events of the type associated with the verb, before that set is returned as the denotation of the verb phrase. This is illustrated in the following queries:

`who discovered` ⇒ *bernard bond cassini cassini_imaging_science_team christy dollfus galileo etc* ...

No *subject*, *object* or prepositional phrase is given in the query, and so all events of type "discover" are returned by the verbphrase and the denotation of the word `who` picks out the *subjects* from those events.

`where discovered io` ⇒ *padua*

No *subject*, or prepositional phrase is given in the query, and so all events of type "discover" are considered and filtered by the denotation of the *object* termphrase `io` and then, those that pass the filter are returned by the verbphrase and the word `where` picks out the location from those events.

`who discovered in 1610` ⇒ *galileo*

No *subject* or *object* is in the query so all events of type "discover" are considered and only those with the *year* property equal to 1610 pass the filter and then the denotation of the word `who` selects the subject which is returned.

`who discovered every moon that orbits mars with one telescope or a moon that orbits jupiter with a telescope` ⇒ *one. ; none. ; none. ; bernard galileo kowal melotte nicholson perrine science_team_-1 science_team_2 ; hall ; hall ; none.*

As shown above, in our semantics, the *subject* and *object* termphrases are treated as filters, as are all prepositional phrases. Note that several results are returned here because the query is syntactically ambiguous. We discuss solutions on how to best present the results of ambiguous queries to the user in Sect. 10.3.

`where discovered in 1610` ⇒ *padua*
`how discovered in padua` ⇒ *galilean_telescope_1*

5.3 Extensions to the Semantics

Some phrases containing nested quantifiers are given by some researchers as examples of non-compositionality. For example: "a US diplomat was sent to every capital" is often read as having two meanings which can only be disambiguated by additional knowledge. We argue that the person posing a query can express the query unambiguously if they are familiar with quantifier scoping conventions used by our processor, as illustrated in the following:

`christy or science_team_19 or science_team_20 or science_team_21 discovered every moon that orbits pluto` ⇒ *False*

In our semantics, quantifier scoping is always leftmost/outermost, and an unambiguous query can be formulated as follows:

`every moon that orbits pluto was discovered by christy or science_team_19 or science_team_20 or science_team_21` ⇒ *True*

Some examples of non-compositionality involve polysemantic superlative words such as "most" in, for example:

"*Who discovered most moons that orbit P. Where P is a planet.*"

If "`most`" is treated as "more than half" then:

`who discovered most moons that orbit mars` ⇒ *hall*

However, consider the answer to the alternate reading "who discovered the most moons that orbit *P*" - i.e. more than anyone else who discovered a moon that orbits *P*.:

`what discovered the most moons that orbit jupiter` ⇒ *science_team_4*

Here, the *subjects* of the "discover" events are sorted based on the cardinality of the number of things they discovered after filtering the events for objects which are moons that orbit `jupiter`. Of the 50 moons that orbit jupiter, `science_team_4` discovered 12 of them.

`how was every moon that orbits saturn discovered` ⇒ *cassini reflector_telescope_1 aerial_telescope_1 refractor_telescope_4 etc ...*

It may be surprising that *cassini* is returned in the answer since it is not a `telescope`, but is instead a `spacecraft`. However, since it was used to discover at least one `moon` that orbits `saturn`, it is considered to have fulfilled the *implement* role and is encoded as such in the triplestore.

6 The FDBR: A Novel Data Structure for Natural Language Queries

6.1 Quantifiers and Events

In 2015, Champollion [5] stated that, at that time, it was generally thought by linguists that integration of Montagovian-style compositional semantics and Davidsonian–style event semantics [7,18] was problematic, particularly with respect to quantifiers. Champollion did not agree with that analysis and presented an integration which he called "quantificational event semantics" which he claimed solved the difficulties of integration by assuming that verbs and their projections denote existential quantifiers over events and that these quantifiers always take lowest possible scope.

In this paper, we borrow much from Montague Semantics (MS), Davidsonian Event Semantics, and Champollion's Quantificational Event Semantics. However, we provide definitions of our denotations in the notation of set theory, which improves computational efficiency and, we believe, simplifies understanding of our denotations. We also believe that our semantics is intuitive, systematic, and compositional.

6.2 Montague Semantics

All quantifiers, such as "a", "every" and "more than two" are treated in MS as functions which take two characteristic functions of sets as arguments and return a Boolean value as result. Our modifications to MS are to use sets of entities instead of predicates/characteristic functions of those sets, and to pair sets of events with each entity; the set of events paired with an entity justify the entity's inclusion in the denotation. For example:

$$\|propernoun\| = \lambda p.\{(e, evs) \mid (e, evs) \in p \ \&\ e = \text{the entity associated with the proper noun}\}$$

$$\|spins\| = \{(e_{phobos}, \{ev_{1360}\}), (e_{deimos}, \{ev_{1332}\}), etc\ldots\}$$

Therefore,

$$\begin{aligned}
&\|phobos\ spins\| \\
\Longrightarrow\ &\|phobos\|\ \|spins\| \\
\Longrightarrow\ &\lambda s.\{(e, evs) \mid (e, evs) \in s \ \&\ e = e_{phobos}\}\ \|spins\| \\
\Longrightarrow\ &\{(e, evs \mid (e, evs) \in \|spins\| \ \&\ e = e_{phobos}\} \\
\Longrightarrow\ &\{(e_{phobos}, \{ev_{1360}\})\}
\end{aligned}$$

We call this set of pairs of entities and events an *FDBR*, and describe it in more detail in Sect. 6.3. In the following example, we show how the FDBR can be used to denote the quantifier `a`. The function *intersect* computes the intersection of two FDBRs based on their entities, keeping the events of the second FDBR and discarding those of the first in the result.

$$\begin{aligned} \mathit{intersect} &= \lambda m \lambda s.\{(e_1, \mathit{evs}_2) \mid (e_1, \mathit{evs}_1) \in m \ \&\ (e_2, \mathit{evs}_2) \in s \ \&\ e_1 = e_2\} \\ \|a\| &= \mathit{intersect} \end{aligned}$$

Therefore,

$$\begin{aligned} & \|a\ \mathit{moon}\ \mathit{spins}\| \\ \Longrightarrow\ & \|a\|\|\mathit{moon}\|\|\mathit{spins}\| \\ \Longrightarrow\ & \{(e_1, \mathit{evs}_2) \mid (e_1, \mathit{evs}_1) \in \|\mathit{moon}\| \ \&\ (e_2, \mathit{evs}_2) \in \|\mathit{spins}\| \ \&\ e_1 = e_2\} \\ \Longrightarrow\ & \{(e_{\mathit{phobos}}, \{\mathit{ev}_{1360}\}), (e_{\mathit{deimos}}, \{\mathit{ev}_{1332}\}),\ \mathit{etc}\ldots\} \end{aligned}$$

We can define the denotations of other quantifiers in terms of *intersect* as well. For example, consider the denotation of `every`, where *ents m* denotes the set of entities that appear in the first column of the FDBR *m*:

$$\begin{aligned} \mathit{ents} &= \lambda m.\{\mathit{ent} \mid (\exists \mathit{evs})\ (\mathit{ent}, \mathit{evs}) \in m\} \\ \|\mathit{every}\| &= \lambda m \lambda s. \begin{cases} \mathit{intersect}\ m\ s, & \mathit{ents}\ m \subseteq \mathit{ents}\ s \\ \emptyset, & \text{otherwise} \end{cases} \end{aligned}$$

Therefore,

$$\begin{aligned} & \|\mathit{every}\ \mathit{moon}\ \mathit{spins}\| \\ \Longrightarrow\ & \|\mathit{every}\|\|\mathit{moon}\|\|\mathit{spins}\| \\ \Longrightarrow\ & \mathit{intersect}\ m\ s, \quad \text{since } \mathit{ents}\ \|\mathit{moon}\| \subseteq \mathit{ents}\ \|\mathit{spins}\| \\ \Longrightarrow\ & \{(e_{\mathit{phobos}}, \{\mathit{ev}_{1360}\}), (e_{\mathit{deimos}}, \{\mathit{ev}_{1332}\}),\ \mathit{etc}\ldots\} \end{aligned}$$

Note that the events *evs* paired with the entities returned in the denotation of "`was every moon that orbits saturn discovered`" are a subset of the events of type "discover" where the *object* property of those events are moons, since the result of *intersect_fdbr* takes the events of from its second argument. This enables additional data to be accessed from those events, as illustrated in the last example query in the previous section, where "`how`" retrieves the *implement* property from those events. This allows all "*wh*"-style questions to be handled compositionally, selecting the desired properties from the events as needed.

6.3 The FDBR

In order to generate the answer to "`hall discovered every moon that orbits mars`", $\|\mathit{every}\|$ is applied to $\|\mathit{moon\ that\ orbits\ mars}\|$ (i.e. the set of `moons` that orbit `mars`), as first argument, and the set of entities that were discovered by `hall`, as the second argument. Our semantics generates this set from the set of events of type "discover" whose the subject property is "*hall*", as discussed below:

Every set of n-ary events (i.e. events with n roles) of a given type, e.g. discovery, defines $n^2 - n$ binary relations. For example, for discovery events:

$$\begin{array}{lll} discover\_rel_{subject \to object} & discover\_rel_{subject \to year} & discover\_rel_{subject \to implement} \cdots \\ discover\_rel_{object \to subject} & discover\_rel_{object \to year} & discover\_rel_{object \to implement} \cdots \\ discover\_rel_{year \to subject} & discover\_rel_{year \to object} & discover\_rel_{year \to implement} \cdots \end{array}$$

etc ... to 20 binary relations for the set of discovery events or an 5-ary discovery relation. For example:

$$discover\_rel_{subject \to object} = \\ \{(ev_{1045}, e_{hall}, e_{phobos}), (ev_{1046}, e_{hall}, e_{deimos}), etc\ldots\}$$

If we collect all of the values from the range of a relation that are mapped to by each value v from the domain (i.e. the image of v under the relation r) and create the set of all pairs $(v, image\_of\_v)$, we obtain a *Function Defined by the Relation* r, i.e. the FDBR. For example:

$$\mathrm{FDBR}(discover\_rel_{subject \to object}) \\ = \Big\{(e_{hall}, \big\{(e_{phobos}, \{ev_{1045}\}), (e_{deimos}, \{ev_{1046}\})\big\}), etc\ldots\Big\}$$

It is these functions that are created, and used, by the denotation of the transitive verb associated with the type of the events. For example in calculating the value of $\|$*who discovered every moon that orbits mars*$\|$, $\|$*every*$\|$ is applied to the set of entities which is the denotation of "`moon that orbits mars`" (i.e $\big\{(e_{phobos}, \{ev_{1045}\})\ , (e_{deimos}, \{ev_{1046}\})\big\}$) and all of the images that are in the second field of the pairs in $\mathrm{FDBR}(discover\_rel_{subject \to object})$. For the pair $(e_{hall}, \big\{(e_{phobos}, \{ev_{1045}\}), (e_{deimos},\ \{ev_{1046}\})\big\})$, $\|$*every*$\|$ returns the non-empty set $\big\{(e_{phobos}, \{ev_{1045}\}), (e_{deimos}, \{ev_{1046}\}\ )\big\}$, and the value in the first field, i.e. e_{hall}, is subsequently returned with the answer to the query.

The various FDBRs are used to answer different types of queries. For example:

`who discovered phobos and deimos` ⇒ *hall*
uses $\mathrm{FDBR}(discover\_rel_{subject \to object})$
`where discovered by galileo` ⇒ *padua*
uses $\mathrm{FDBR}(discover\_rel_{location \to subject})$
`how discovered in 1610 or 1855` ⇒ *galilean_telescope_1*
uses $\mathrm{FDBR}(discover\_rel_{implement \to year})$

7 Handling Prepositional Phrases

Prepositional phrases (PPs) such as "`with a telescope`" are treated similarly to the method above, 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.

8 Handling Superlative Phrases

A novel feature of our semantics is that we can directly accommodate superlative phrases such as "`most`" and "`the most`" inside chained prepositional phrases. Here, we take "`most`" to mean "more than half" and "`the most`" to mean "more than anything else". This makes it possible to answer queries such as "`who discovered a moon using the most telescopes`" and "`most planets are orbited by a moon`" with our NLQI.

Superlatives can be placed nearly anywhere a determiner can exist. This makes it possible to nest superlatives inside chained prepositional phrases, a property we believe to be novel in our semantics. For example, consider "`what discovered at the most places using the most telescopes`", where "`the most`" occurs inside both prepositional phrases "`at the most places`" and "`using the most telescopes`". The query is always evaluated in left-to-right order, and results are sorted by each superlative phrase in the order they appear. In this case, the results are first sorted by the number of places, followed by the number of telescopes, both in descending order. First, the denotation for "`most`" (as in "more than half") is defined as follows:

$$\|most\| = \lambda m \lambda s. \begin{cases} intersect\ m\ s, & |intersect\ m\ s| > |s|/2 \\ \emptyset, & \text{otherwise} \end{cases}$$

Providing a denotation for superlative phrases such as "`the most`" is more challenging. To achieve this and maintain compositionality, the superlatives are handled in the denotation for the transitive verbs. First, we introduce the denotation for "`the most`":

$$\|the\ most\| = \lambda m.(\text{GT}, intersect\ m)$$

"`the most`" takes a nounphrase as an argument and returns a pair consisting of the ordering GT (i.e. "greater than"), and a termphrase created using partial application of the *intersect* function. This ordering describes how the results should be sorted – in this case, in descending order.

The denotation for prepositional phrases is modified to include an ordering as third parameter, which may take on the special value *None* if the prepositional phrase does not contain a superlative phrase within it. However, if it does contain a superlative phrase, the ordering of the prepositional phrase is set to the ordering specified in the denotation of the superlative phrase.

The denotation for transitive verbs is modified such that, at the end of the prepositional phrase evaluation performed previously, where the filtered FDBR is obtained (containing only *relevant* events [19]), the resulting FDBR is passed to a new function, `filter_super`, which handles superlative evaluation. The behavior of this function is as follows. First, if no superlatives are present (i.e. the ordering in the denotation of each prepositional phrase is *None*), nothing more is done, and the behavior of the new denotation is identical to the previous one.

If superlatives are present, however, they are evaluated in the order they appear. For each superlative phrase present in the chain of prepositional phrases, the FDBR is expanded to a new data structure called a *Generalized FDBR* (or *GFDBR*) which is similar to an FDBR, except that instead of having a set of events in its second column, it has an FDBR instead. The GFDBR is formed by taking the set of events in each row of the original FDBR, and expanding them into an FDBR using the role attached in the prepositional phrase. This is used to obtain the cardinality of the number of entities that the subject is related to in that role under the FDBR (called the *object cardinality*). Now, these object cardinalities are used to partition the GFDBR into a set of GFDBRs, where the set with the highest (or lowest) object cardinality is chosen to replace the original GFDBR, depending on the ordering in the denotation of the prepositional phrase (i.e. the ordering denoted by the superlative phrase). For "`the most`", it would be the set with the highest object cardinality (since the ordering is *GT*). In the future, for "`the least`", it would be the set with the lowest object cardinality. The GFDBR is then converted back into an FDBR by keeping only the events in each row, and the process repeats until no more superlative phrases are remaining. The final FDBR is returned as the result.

This allows superlative phrases to still be handled in left-to-right evaluation order, and it also allows results to be sorted by multiple columns. For example "`who discovered the most moons in the most places`" would first sort by "`the most moons`", and following that, would sort by "`the most places`". Currently, we are not able to accommodate "`the least`", as the semantics filters out rows with empty sets of events in FDBRs before superlatives work on them. For example, if a user were to ask "`which planet has the least moons`", the answer currently would be "`earth`", as it has only one moon, and our system filters out both "`venus`" and "`mercury`" (which have no moons) before they have a chance to affect the result. This seems to be related to our original Open World Assumption, where we only include results in the result set if there is at least one accompanying event in the FDBR to justify its inclusion. It is possible that if negation could be accommodated in the semantics, then "`the least`" could be handled as well, since they seem to be related problems.

9 Our Approach with Relational Databases

Our NLQI can be easily adapted for use with conventional relational databases. First, note that each event at minimum contains a role *ev_type* that identifies the type of event, and as noted in Sect. 4, there is a general expectation that events of the same type should contain similar roles. Second, note that the event identifier in each triple is a URI and is therefore unique by definition.

Assume the roles that events of a particular type t are fixed, including optional roles. Let N be the number of roles, including optional roles, that an event of type t contains. Then an event of type t can be described as a row in a relation with N columns, each role occupying one column respectively, with optional roles taking on a special value `NULL` if they are not present in that particular event. Let this relation be called *ev_type*.

Store this relation in a relational database as a table using the event identifier as the primary key. Now, only the triple retrieval functions in Sect. 10.2 need to be modified to use this database in place of a triplestore. This architecture allows the denotations to remain unchanged and yet still work with different types of databases. Note that triplestores do have an advantage in that they need not be rebuilt if a new role is added to the event. The decision to choose one approach over the other needs to be weighed based on application specific factors.

10 Implementation of Our NLQI

We built our query processor as an executable attribute grammar using the *X-SAIGA* Haskell parser-combinator library package [15]. 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 $O(n\ lg\ n)$ 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 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.
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.
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* [25] Haskell package enables a simple interface between our semantic processor and SPARQL endpoints to our triplestores.

10.1 System Architecture

A flowchart of our system architecture is presented in Fig. 1.

The query begins as a string of text as sent to the semantics, which is then sent directly to the parser, as described in Sect. 3.1. This produces two results:

(1) A function that, given a set of triples, will evaluate the query with respect to that set of triples and return the result
(2) A "*Memo Tree*" that roughly follows the syntax tree resulting from the parse of the input string. In addition to providing a unique name to each sub-expression of the parsed input, it is also used to determine which queries need to be evaluated against the remote triplestore.

The function produced in (1) requires a set of triples to produce a result. While it is possible, given sufficient time and resources, to directly retrieve all triples from the remote triplestore and pass them directly into this function to evaluate the input, in practice it is cost prohibitive to do so.

Instead, we retrieve only *relevant triples* [19] from the remote triplestore and we create a *reduced* triplestore from them which is then passed into (1).

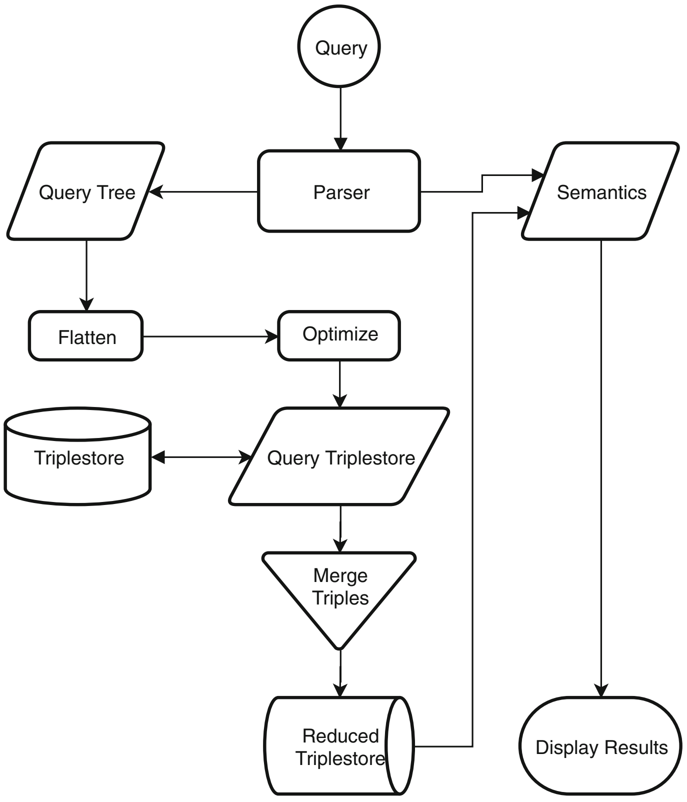

Fig. 1. Application architecture.

The Memo Tree obtained in (2) is traversed to obtain the set of all triplestore queries that are required to evaluate each sub-expression of the parsed input. These queries correspond to the `getts` family of functions described in Sect. 10.2. The results of these queries may *overlap*, i.e. share triples in common with those of other queries in the set. An optimization step is performed to eliminate these redundant queries. Domain specific knowledge could be used to improve this process where appropriate. Finally, these optimized queries are evaluated against the remote triplestore and the results are merged and stored locally in the reduced triplestore. These triples are then passed to the function produced in (1), yielding the final result. This is one area where our NLQI differs from other NLQIs to the Semantic Web – notice that nowhere do we attempt to directly translate the NL query into SPARQL or any other querying language. Instead, we rely on simple triple querying primitives which are embedded in the semantics to perform this task for us.

The architecture presented in this section lends itself to a very clean implementation in Haskell, where the semantics themselves can be written as pure functions, with the only impure parts of the NLQI being those that directly deal with querying the triplestore and with presenting these results to the user. We expand on the individual sub-components of the NLQI in the following sections.

10.2 Triple Retrieval

Remote Triplestore. Our semantics does not directly depend upon any particular query language. When querying remote triplestores, the NLQI requires only two conceptually simple functions. The first is:

```
getts_triples_entevprop_type ev_data prop_names ev_type
```

This function is used to retrieve triples belonging to the relation `ev_type`. `prop_names` is a list of columns of the relation to retrieve. Only the names of the columns of the relation that are actually required are listed here. Finally, `ev_data` is the URL used to access the remote triplestore or database. For example, in the query `what discovered`, it may be invoked as follows:

```
getts_triples_entevprop_type url ["subject"] "discover_ev"
```

This would retrieve the triples of all "discover" events that contain a *subject* property, including the triples describing the type of those events. The second function is:

```
getts_triples_members ev_data set
```

Here, `ev_data` performs the same function as it did previously, and "`set`" indicates the name of a set, for example the moons or the set of things that spin. This retrieves the triples of all "membership" events whose *object* property corresponds to that set, including the triples describing the type of those events.

Together, these two primitives can be used to retrieve triples from event-based triplestores, provided the names of the roles to be queried are known. This would typically be described in a schema, but in simple cases may be feasible to hard-code into a program. To see how these two primitives work in action, consider the following complex query, featuring chained prepositional phrases:

```
which person discovered a moon in 1877 with a telescope
```

This would invoke the following queries to the database:

```
  getts_triples_entevprop_type url ["subject", "object",
"year", "implement"] "discover_ev"
  getts_triples_members url "moon"
  getts_triples_members url "telescope"
  getts_triples_members url "person"
```

These four queries to the remote triplestore, taken together, will retrieve enough information to answer the user's query. Transitive and intransitive verbs are implemented in terms of `getts_triples_entevprop_type`. Common nouns and adjectives are implemented in terms of `getts_triples_members`. These conceptually simple functions are easy to implement in SPARQL, SQL, and as Triple Pattern Fragments [23]. An example implementation is provided in our source code, available on Hackage [15] for both Triple Pattern Fragments and SPARQL.

After all "`getts`" queries are evaluated, their results are merged together into a local *reduced* triplestore. The idea behind this triplestore is that it contains enough triples to evaluate the correct result, but no more than that. In other words, the results from passing in the entire triplestore to the semantic function in (1) and the results from passing in the reduced triplestore should be equivalent.

Reduced Triplestore. Once the reduced triplestore is passed into the semantics, however, it still needs to be queried by the semantic functions in the denotations. This is where the boundary of the impure code of the NLQI meets the pure code of the semantics. At this higher level, there are three primitives that are used to query the reduced triplestore:

- `pure_getts_triples_entevprop_type ev_data prop_names ev_type`
- `pure_getts_triples_entevprop ev_data prop_names evs`
- `pure_getts_members ev_data set`

These are very similar functions to those described previously, however they are implemented as pure functions in Haskell. The actual implementation of the reduced triplestore is opaque to the semantics, which rely strictly on these three functions to retrieve triples from the reduced triplestore. Implementing these as pure functions allows them to be embedded in the semantics, which are implemented as pure functions themselves. This provides a number of benefits, including allowing the semantics and queries to be lazily evaluated.

`pure_getts_triples_entevprop_type` performs a similar role as it did previously. `pure_getts_triples_entevprop` is a new function that, instead of specifying an event type parameter, specifies a set of events instead. This is used to implement chained prepositional phrases, where sets of events are honed down in the order that the phrases occur in (from left to right). Finally, `pure_getts_members` performs a similar function as it did previously, except this time it directly returns an FDBR from the members of the set given to the events in which the set membership is recorded.

10.3 Handling Ambiguity in the Query Interface

Syntactic Ambiguity. As queries may be ambiguous, it's important that users see how their queries were parsed to understand the result given. Our system displays the parse tree along with the query result to assist with this. The parse tree is presented in a familiar Haskell syntax to indicate scoping. As an example, consider the scoping of the simple query "`who discovered a moon that orbits mars`":

```
who (discovered (a (moon `that` (orbits mars))))
```

Here, we see that scoping of denotations is shown with parentheses. Prepositional phrases are enclosed inside square brackets, with commas to delimit chained prepositional phrases:

```
who discovered a moon in 1877 with a telescope
```

⇒ `who (discovered (a moon) [in 1877, with (a telescope)])`

This mirrors the familiar list syntax that Haskell offers and suggests to the user that the prepositional phrases will be evaluated in the order presented (left to right), allowing users to understand exactly how their query is evaluated by the system. Now, consider the following ambiguous query:

```
who discovered a moon that orbits in 1877
```

There are two possible parses of this query, depending on which transitive verb the prepositional phrase "`in 1877`" is applied to:

`who (discovered (a (moon `that` (orbits [in 1877])))))` ⇒ *none*

`who (discovered (a (moon `that` orbits)) [in 1877])` ⇒ *hall*

In the first case, the prepositional phrase "`in 1877`" is treated as though it applies to "`orbits`". However, the result is "*none*" because orbit events do not have a concept of time in our database. If we were to add a *year* role to the "orbit" relation, then all planets and moons in the solar system would be returned. In the second case, "`in 1877`" applies to "`discovered`", a relation which has the concept of a time of discovery (the *year* role). As `hall` is the only person that discovered anything in 1877, only they are included in the result.

Our system permits highly ambiguous input, providing a result for each possible parse of that input. However, it may be the case that a user has a clear understanding of how they want their query to be parsed and would gain no benefit from seeing other possible parses of their query. Fortunately, this use case is easily accommodated with a simple extension to our NLQI: allowing the scoping syntax as presented above directly in the query interface itself. For example, a user could directly query "`what (discovered (a (moon 'that' orbits)) [in 1877])`", which would exclude the other parse as mentioned in the example above. In fact, the query need not even be fully explicitly scoped to benefit from this. A partial scoping such as "**`what discovered (a moon that orbits) in 1877`**" would be sufficient to exclude the other undesirable parses from the result. We intend to implement this functionality in our NLQI in the very near future.

It may also be worthwhile to implement a simple dialogue-based approach to disambiguation, where the system could simply provide the possible parses to the user and allow them to choose which one they intended. This approach may be beneficial when using speech to interact with the system, as providing scoping with the above method directly with speech would be very inconvenient. An example dialogue could be:

User:

```
   what discovered a moon that orbits mars in 1877 with a
telescope
```

Interface:

```
   There are three possible ways I can interpret this
query.
   Which one do you mean?
   1) what (discovered (a (moon 'that' (orbits mars [in
1877, with (a telescope)])))))
   2) what (discovered (a (moon 'that' (orbits mars))) [in
1877, with (a telescope)])
   3) what (discovered (a (moon 'that' (orbits mars [in
1877]))))
[with (a telescope)])
```

User:

```
   2
```

Interface:

```
   OK -- the result of the second interpretation is
``hall''
```

If the modality of the interface is by voice, reading the scoping directly as presented above may be inconvenient to users. Fortunately, it is possible to verbally state the scoping in an intuitive way:

User:

```
   what discovered a moon that orbits mars in 1877 with a
telescope
```

Interface:

```
   I can interpret this three different ways. In the first
interpretation, the prepositions ``in 1877'' and ``with a
telescope'' apply to the verb ``orbit''. Is that what you
meant?
```

User:

```
   no
```

Interface:

```
   In the second interpretation, the prepositions ``in
1877'' and ``with a telescope'' apply to the verb ``discov-
ered''. Is that what you meant?
```

User:

```
   yes, that's what i meant
```

Interface:

```
   OK -- the result of that interpretation is ``hall''
```

Given the different nature of the user's responses compared to the queries themselves, they may be subject to a different grammar or may be handled by a different system entirely that permits more free-form responses to be given. This could be a good opportunity to integrate Machine Learning-based NLP approaches in the NLQI in the future, as they are ideally suited to use cases involving loosely structured input.

Semantic Ambiguity. Semantic ambiguity may also be accommodated by permitting multiple definitions of the same terminal in the grammar, augmenting it with a human readable description of what the terminal means. Each definition would be evaluated as though it were a different parse of the query, although each parse would have the same syntax tree. To avoid confusion, the human readable definition of the word could be printed below the tree.

10.4 Semantic Implementation

The semantics themselves are completely unaware of the structure of the underlying triplestore or the methods and query languages used to retrieve triples from it. Recall from Sect. 10.1 that the result of a parse of user input produces two items: a pure function that, given a triplestore as input will produce the result of a query and a tree that represents the query itself, including the types of queries that are required from a remote triplestore.

Applying Multiple Semantics in Parallel. The Biapplicative Bifunctor in Haskell, which is inspired from its counterpart in category theory, can serve as a generalization of function application. One possible use for it is to apply pairs of values to pairs of functions. Briefly, given two arbitrary functions f and g and two values a and b we can use the biapplicative operator `<<*>>` to apply a and b both functions in parallel: (f, g) `<<*>>` $(a, b) = (f\ a, g\ b)$. The functions themselves need not be related.

First, we introduce an operator, `>|<`, that allows us to bridge together two semantics such that they can be applied using `<<*>>`:

$$a \texttt{ >|< } b = (a, b)$$

This allows these two independent functions to be applied in parallel while parsing the input string using the exact same grammar and no code duplication, provided the `<<*>>` is used in place of function application. For example, "`a moon spins`" is evaluated as though it were written as "`a <<*>> moon <<*>> spins`" under this approach. Our NLQI uses this to construct the Memo Tree in parallel while applying the denotations of the words in the query. Consider the following example, where GIntersect and GMembers are constructors of the Memo Tree:

```
a'     = a >|< GIntersect
moon'  = moon >|< GMembers "moon"
spins' = spins >|< GMembers "spins"
```

Therefore,

```
      a' <<*>> moon' <<*>> spins'
=> (a moon spins, GIntersect (GMembers "moon") (GMembers "spin"))
```

However, this is somewhat inconvenient and unfamiliar syntax to work with. Fortunately, it is trivial to define a set of "wrapper" functions to restore the original function application syntax:

$$wrap_N\ (f, g)\ (a_1, b_1)\ (a_2, b_2)\ \ldots\ (a_N, b_N) = (f\ a_1\ a_2 \ldots a_N,\ g\ b_1\ b_2 \ldots b_N)$$

Here, the function $wrap_N$ takes a pair of functions (f, g) with arity N and then N pairs of arguments to be applied in order to f and g respectively. This allows "`a'<<*>> moon' <<*>> spins'`" above to be written as "`a' moon spins`", where $\texttt{a}'' = wrap_2\ \texttt{a}'$. Therefore, we can retain the familiar function application syntax in the semantics while taking advantage of parallel function application. By itself, this is a convenience, but let us revisit the Memo Tree once more. It has two uses. The first is as stated previously, in determining which queries need to be performed against the remote triplestore. The second is that this allows us to assign a unique identifier to each sub-expression of the parsed input.

Memoized Compositional Semantics. Consider the query "`what is orbited by a thing that was discovered by a person that discovered phobos`", containing three nested transitive verbs. One possible parse of this query yields:

```
what (is orbited [by (a (thing 'that' (was discovered [by
      (a (person 'that' (discovered phobos)))]))))])
```

A query's sub-expressions may be evaluated multiple times during the prepositional filtering of a transitive verb (i.e one evaluation for each row of the FDBR denoted in that transitive verb). This has a compounding effect when transitive verbs are nested as sub-expressions in prepositional phrases of other transitive verbs. In general, if there are m nested transitive verbs in a query, each having an FDBR with n rows. Then the complexity for evaluation is $O(n^m)$.

As it turns out, we can use the Memo Tree to memoize the results of the sub-expressions of a query, drastically reducing the number of re-evaluations performed. The memoization occurs in a more sophisticated version of the $wrap_N$ functions described previously, which use the unique identifier provided by the Memo Tree to memoize the results of the semantic functions as they are evaluated. This is completely transparent to the user, and the familiar function application syntax used in all previous examples still remains. This reduces the complexity to $O(mn)$, where m is the number of nested transitive verbs, each having an FDBR with n rows. All sub-expressions in the query are memoized, including the final result of the query expression itself.

The State monad in Haskell is used to thread the memoized state throughout the execution of the semantics. This mirrors the memoization technique used in the parser itself to provide efficient parsing using combinators [13]. We believe this two-pronged approach to triplestore retrieval and memoization is novel and has not been used in any other Compositional Semantics-based systems. We intend to expand more on our approach in a future publication, as we believe it to be useful for creating modular and efficient compositional NLQIs that can scale to the needs of the Semantic Web.

11 Related Work

Orakel [6] is a portable NLQI which uses a Montague-like grammar and a lambda calculus semantics. Our approach is similar in this respect. 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.

YAGO2 [17] 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. SPOTLX can handle queries with prepositional aspects involving time and location. However, no mention is made of chained complex PPs.

Alexandria [24] is an event-based triplestore, with 160 million triples (representing 13 million n-ary relationships), derived from FreeBase. Alexandria uses a neo-Davidsonian [18] 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`".

Blackburn and Bos [4] implemented lambda calculus with respect to natural language, in Prolog, and Van Eijck and Unger [22] have extensively and clearly discussed such implementation in Haskell. Implementation of the lambda calculus for open-domain question answering has been investigated by [1]. The SQUALL query language [10,11] 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 can handle quantification, aggregation, some forms of negation, and simple unchained prepositional phrases containing the word "at" and "in". It can also handle superlative phrases as long as they are not nested under a prepositional phrase. Notably, the scope of prepositional phrases in SQUALL are the entire sentence they reside in. 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 [11]).

12 Future Work

Negation. Our system currently relies on the Open World Assumption, where the absence of evidence cannot be treated as having evidence of absence. As a consequence of this, the system currently is unable to handle negation, and does not have a denotation for the words "`no`" and "`not`".

However, there is a clear need for handling negation in our semantics where the Closed World Assumption holds. For example, it should be possible to answer queries such as "`who did not discover a moon` "or" `what discovered no moon`". Work has been done on event-based semantics that can handle negation [5]. We believe it should be possible to accommodate negation in our semantics as well using a similar approach, and in turn provide a denotation for "`the least`" as well, as noted in Sect. 8.

DBPedia. With the addition of memoization in our semantics, we feel our approach is now scalable enough to work directly with DBPedia. We intend to expand on how our semantics can handle large triplestores such as DBPedia in a future publication. In particular, an interface to DBPedia will allow our approach to be directly evaluated with existing systems in use, such as YAGO [17].

Hardware Acceleration. Consider that the reduced triplestore described in Sect. 10.2 is stored locally in the query interface and is queried with the pure "`getts`" functions. These could make good candidates for offloading to FPGA fabric or a GPU for hardware acceleration. Work has been done in developing on FPGAs using Haskell [3]. This could allow for both low latency and low power consumption in embedded consumer devices, such as those that operate on the Internet of Things.

Non-Event-Based Triplestores. We also believe it should be possible to handle non-event based triplestores as well using our approach using a translation layer. It may be possible to use ontological information to provide an event-based view to many kinds of non-event based data. Machine Learning approaches could provide a way forward in the absence of or lacking sufficient ontological information about a triplestore.

13 Conclusions

This work comes at an appropriate time when massive triplestores, such as DBpedia [2] are being created containing billions of verified facts. We are currently looking at how such facts can be converted to event-based triples which can be queried by our interface. We are confident that, after we accommodate negation, our compositional semantics is appropriate for answering most queries that are likely to be asked of data stores containing everyday knowledge. We have shown how the FDBR data structure presented in this paper can be used to handle many kinds of complex language features, including chained prepositional phrases and superlatives. The way quantification is handled within the semantics is consistent with other work in this area, as discussed in Sect. 6.1. Our approach is extensible enough that it can accommodate queries to both relational and non-relational types of database, including Semantic Web triplestores. Our approach is also suitable for use on low power devices, which may be useful for applications on the Internet of Things (IoT).

We have shown how our system is tolerant of highly ambiguous user input and we discussed possible ways to present this in Sect. 10.3. In particular, we discussed how both semantic and syntactic ambiguity could be handled. We also presented a novel approach to memorizing compositional semantics using unique identifiers attached to sub-expressions in a query, substantially improving the time complexity of evaluation. We also showed how those unique identifiers are also useful to determine the set of queries that need to be made to the remote database.

Our next goal is to provide an NLQI to DBPedia using our approach with the techniques described here, and then evaluate the effectiveness of our system relative to other NLQIs using established benchmarks, such as QALD [21].

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. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: a nucleus for a web of open data. In: Aberer, K., et al. (eds.) ASWC/ISWC -2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_52
3. Baaij, C.: Cλash: from Haskell to hardware. Master's thesis, University of Twente (2009)
4. Blackburn, P., Bos, J.: Representation and Inference for Natural Language. A First Course in Computational Semantics, CSLI (2005)
5. Champollion, L.: The interaction of compositional semantics and event semantics. Linguist. Philos. **38**(1), 31–66 (2014). https://doi.org/10.1007/s10988-014-9162-8
6. Cimiano, P., Haase, P., Heizmann, J., Mantel, M.: ORAKEL: a portable natural language interface to knowledge bases. Technical report, Institute AIFB, University of Karlsruhe (2007)
7. Davidson, D.: The logical form of action sentences (1967)
8. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM **13**(2), 94–102 (1970). https://doi.org/10.1145/362007.362035
9. Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management, pp. 501–519. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-04329-1_21
10. Ferré, S.: SQUALL: a controlled natural language for querying and updating RDF graphs. In: Kuhn, T., Fuchs, N.E. (eds.) CNL 2012. LNCS (LNAI), vol. 7427, pp. 11–25. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32612-7_2
11. Ferré, S.: SQUALL: a controlled natural language as expressive as SPARQL 1.1. In: Métais, E., Meziane, F., Saraee, M., Sugumaran, V., Vadera, S. (eds.) NLDB 2013. LNCS, vol. 7934, pp. 114–125. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38824-8_10
12. Frost, R., Launchbury, J.: Constructing natural language interpreters in a lazy functional language. Comput. J. **32**(2), 108–121 (1989)
13. Frost, R.A., Hafiz, R., Callaghan, P.: Parser combinators for ambiguous left-recursive grammars. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 167–181. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77442-6_12
14. Frost, R.A., Peelar, S.M.: A new data structure for processing natural language database queries. In: Proceedings of the 15th International Conference on Web Information Systems and Technologies, WEBIST 2019, Vienna, Austria, 18–20 September 2019, pp. 80–87 (2019). https://doi.org/10.5220/0008124300800087
15. Hafiz, R., Frost, R., Peelar, S., Callaghan, P., Matthews, E.: The XSaiga package (2018)
16. Hirst, G.: Semantic Interpretation and the Resolution Of Ambiguity. Cambridge University Press, Cambridge (1992)

17. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. Artif. Intell. **194**, 28–61 (2013)
18. Parsons, T.: Events in the Semantics of English, vol. 5. MIT Press, Cambridge (1990)
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. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Boston (1985)
21. Usbeck, R., Gusmita, R.H., Ngomo, A.C.N., Saleem, M.: 9th challenge on question answering over linked data (QALD-9). In: Semdeep/NLIWoD@ ISWC, pp. 58–64 (2018)
22. Van Eijck, J., Unger, C.: Computational Semantics with Functional Programming. Cambridge University Press, Cambridge (2010)
23. Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., Van de Walle, R.: Web-scale querying through linked data fragments. In: LDOW. Citeseer (2014)
24. Wendt, M., Gerlach, M., Düwiger, H.: Linguistic modeling of linked open data for question answering. In: Proceedings of Interacting with Linked Data (ILD 2012) [37], pp. 75–86 (2012)
25. Wheeler, J.: The hsparql package. In: The Haskell Hackage Repository (2009). http://hackage.haskell.org/package/hsparql-0.1.2