An Event-Driven Approach for Querying Graph-Structured Data Using Natural Language

Richard A. Frost, Wale Agboola, Eric Matthews and Jon Donais
School of Computer Science
University of Windsor, Canada
richard@uwindsor.ca

ABSTRACT

An ideal way for people to query graph-based knowledge, including triplestores in the semantic web, would be for them to ask questions in a natural language (NL). However, existing NL query interfaces to graph-based data have limited expressive power and cannot accommodate arbitrarily-nested quantification (i.e. phrases such as "a gangster who joined every gang") together with multiple complex prepositional phrases, such as "in a city located in Illinois in 1918 using a set of keys that was stolen from a gangster". It would appear that the commonly-used "entity-based" triplestores, together with what has become the de-facto approach of converting NL queries to SPARQL queries before being evaluated, hinders the development of expressive NL query processors. The reason is that entity-based triples are not conducive to the development of semantic theories of complex prepositional phrases, and the development of such theories is made considerably more complex when translation to SPARQL has to be taken into account. An alternative approach, which uses "event-based" triplestores, treats (bracketed) English queries as expressions of the lambda calculus which can be evaluated directly with respect to the triplestore. This approach facilitates the development of a formal denotational semantics of English queries which easily accommodates complex prepositional phrases. The approach described here could be used to develop a denotational semantics for a highly-expressive NL query language, and then that semantics could be used to guide the design of an NL query to SPARQL translator, thereby taking advantage of SPARQL optimizations.

Categories and Subject Descriptors

H.2.4 [**database management**]: Query processing; H.5.2 [**user interfaces**]: Natural language

General Terms

Theory

1. INTRODUCTION

The fact that Al Capone joined the Five Points Gang can be represented by the following triple:

```
(.../capone, .../joined, .../fpg)
```

where .../ are Uniform Resource Identifiers (URIs) for namespaces, and .../capone is a URI for a person. However, there is a problem with this approach. It is difficult to add related data such as the fact that Capone joined the FPG gang in 1914. It is insufficient to simply add the triple:

```
(.../capone,.../year_joined_gang, .../1914)
```

as this does not provide the necessary link between the two facts (the link is necessary because Capone joined several gangs). Various approaches are available to overcome this problem, only one of which concerns us in this paper. The aproach in which we are interested is one which uses events rather than entities as subjects in the triples. For example, the fact that Al Capone joined the Five Points Gang can be represented as follows (note that from now on, we use ENT "capone" in place of ".../capone" etc.

```
{(EV 1001, REL "type",    TYPE "join_ev"),
 (EV 1001, REL "subject", ENT  "capone"),
 (EV 1001, REL "object",  ENT  "fpg")}
```

We can add the fact that Capone joined the Five Points Gang in 1914, with:

```
(EV 1001,  REL "year", ENTNUM 1914)
```

A particular advantage of this approach is that the use of events facilitates the creation of a powerful denotational semantics for NL queries to graph-based data. In particular, use of events enables us to create an NL semantics with the following six properties: 1) the semantics is denotational in the sense that English words and phrases have a well-defined mathematical denotation (meaning), 2) the meaning of a composite phrase can be created by applying simple operations to the meanings of its components, 3) it is referentially transparent in the sense that the meaning of a word or phrase (after syntactic disambiguation) is the same no matter in what context it appears. 4) there is a one-to-one correspondence between the semantic rules describing how the meaning of a phrase is computed from its components and the syntactic rules describing the structure of the phrase, 5) it is computationally tractable, and 6) the meanings of words are defined directly in terms of primitive triple-store retrieval operations.

These six properties enable NL triple-store query processors to be implemented as highly modular syntax-directed interpreters. The advantage of this is that processors for individual language constructs can be built and tested separately. Consequently, the query processors can easily be extended to accomodate new language constructs such as prepositional phrases.

The semantics that we have developed is complex and is the result of two revisions that we have made to a well-known formal semantics of English, called Montague Semantics (MS) [6]. We first modified MS to create a computationally tractable form called FLMS which is suitable as a basis for NL query interfaces to relational databases. We then modified FLMS to a form which we call EV-FLMS which is suitable as a basis for querying event-based triple stores. Owing to the complexity of these modifications, this paper describes the two revisions in two separate sections.

The paper is structured as follows: in 2.1 we introduce Montague Semantics. In 2.2 and 2.3 we show how MS can be extended and converted to a computationally tractable form which can be used as a basis for NL query interfaces to conventional relational databases. In section 3 we discuss in more detail how knowledge can be represented in triple stores. In 4 we introduce some primitive retrieval operators for triple stores. In 5 we introduce a new event-based version of FLMS, called EV-FLMS and show how the meaning of words and phrases can be defined in terms of the triple-store retrieval operators. In 5.2 we show how the semantics accommodates prepositional phrases, thereby achieving all six of the properties listed above. In 6 we give examples of how complex queries such as "which gangster who stole a car in 1908 or 1918 in Manhattan joined a gang that was joined by Torrio?" are answered. In 7, we briefly discuss the use of a parser to disambiguate queries. In 8 we mention related work. We conclude in 9 with work yet to be done.

2. A COMPOSITIONAL SEMANTICS FOR NL RELATIONAL DB QUERIES

We begin by discussing MS and show how the meaning of a sentence in English can be composed from the meanings of its component words and phrases. We then show how MS can be extended and modified for use as a basis for NL query interfaces to conventional relational databases.

2.1 Montague Semantics

If we ignore that part of MS which deals with intensional and modal aspects of language, common nouns such as "thief" and intransitive verbs such as "smokes" can be thought of as denoting predicates over the set of entities in the "universe of discourse", i.e. characteristic functions of type `entity` → `bool`, where `x` → `y` denotes the type of functions whose input is a value of type `x` and whose output is of type `y`. One of Montague's many insights is that proper nouns (i.e. names) do not denote entities directly. Rather, they denote functions defined in terms of entities. For example, the proper noun "Capone" denotes the function `λp p cap` where `cap` represents the entity associated with the name "Capone". (For readers not familiar with the lambda calculus, the expression `λx e` denotes a function which, when applied to an argument `y`, returns as result the expression `e` with all instances of `x` in it replaced by `y`.) According to the rules proposed by Montague, the phrase "Capone smokes" (ignoring temporal aspects) is interpreted as shown below, where `a => b` indicates that `b` is the result of evaluating `a`, ‖x‖ represents the denotation (meaning) of the word or phrase x, and `x_pred` is the predicate associated with the word x.

```
‖Capone smokes‖ => ‖Capone‖ ‖smokes‖
=> (λp p cap) smokes_pred
=> smokes_pred cap
```

Quantifiers such as "every", and "a" denote higher-order functions of type: `(entity→bool)→((entity→bool)→bool)`
For example:

```
‖every‖ = λpλq ∀x (p x) → (q x)
```

where → denotes logical implication in this context. Accordingly, the phrase "every thief smokes" is interpreted as:

```
(λpλq ∀x(p x)→(q x))thief_pred smokes_pred
=>(λq ∀x thief_pred (x) → q(x)) smokes_pred
=> ∀x thief_pred (x) → smokes_pred(x)
```

There are many advantages to MS including the fact that phrases of the same syntactic type, e.g. "Capone" and "every thief" have denotations of the same type i.e. `(entity → bool) → bool`, making the semantics highly compositional with respect to the syntactic structure of the phrase. Consequently, the semantics is easy to implement in a syntax directed interpreter as there is a one-to-one correspondence between the syntactic rules of the grammar defining the query language and the semantic rules defining how the meaning of a compound construct is computed from the meaning of its components). There are, however, two disadvantages of directly implementing MS as the basis for a database query processor, as discussed in the next two sub-sections.

2.2 An explicit denotation for transitive verbs

MS is not fully compositional as it does not have an explicit denotation for transitive verbs. Instead it leaves transitive verbs uninterpreted throughout the rewriting of the lambda expression denoting the sentence or phrase, and only deals with the transitive verb at the very end through a syntactic rewrite rule (see page 216 in [6] for the details). In earlier work [11] we developed a method for defining the denotation of transitive verbs explicitly. Accordingly, the denotation of "join" (as in "join a gang") is as follows:

```
‖join‖ = λz z(λxλy join_pred (y, x))
```

Where `join_pred` is the two place predicate corresponding to the word "join". Note that this is similar to, but not exactly the same as, that proposed by Hendricks [18], Main and Benson [22], Blackburn and Bos [2] (who attribute it to Robin Cooper at the University of Goteborg), and Clifford [4]. The following shows the use of this denotation:

```
‖Did Capone join the Five Points Gang?‖
by parsing
=> ‖Capone‖(‖join‖ ‖Five Points Gang‖)
=> (λp p cap) ((λz z(λxλy join_pred(y,x)))(λq q fpg))
=> (λp p cap) ((λq q fpg)(λxλy join_pred(y,x)))
=> (λp p cap) ((λxλy join_pred(y, x)) fpg)
=> (λp p cap) (λy join_pred(y, fpg))
=> (λy join_pred (y, fpg)) cap
=> join_pred(cap, fpg)
```

which returns True if Capone joined the Five Points Gang.

2.3 An efficient version of MS

Another disadvantage of MS as a basis for database query processors is that a direct implementation of the denotations

of phrases which include the word "every" is computationally intractable. This is due to the fact that the function which denotes the word "every" requires all entities in the universe of discourse to be examined. For example:

```
∀x thief_pred(x) → smokes_pred(x)
```

In order to overcome this problem, MS can be converted to a semantics called FLMS that is based on sets and relations rather than on their corresponding predicates. In this approach, which was first suggested and partially implemented by Frost and Launchbury [9] and further developed by Frost and Fortier, [15], sets and relations are used in the denotations of common nouns, intransitive verbs, and transitive verbs, rather than their corresponding predicates, and all other denotations are modified appropriately:

```
‖thief‖ = {capone, torrio, moran ..
‖gang‖ = {bowery, fpg ..
‖smoke‖ = {capone, torrio, moran ..
‖Capone‖ = λp capone ∈ p
‖Moran‖ = λp moran ∈ p
‖Torrio‖ = λp torrio ∈ p
‖Five Points Gang‖ = λp fpg ∈ p
‖every‖ = λsλt s → t
‖a‖ = λsλt s ∩ t ≠ {}
‖and‖ = λfλg λs ((f s) & (g s))
‖join‖ = λq {x|(x,image_x) ∈ collect(join_rel)
                                    & q(image_x)}
join_rel = {(capone, bowery),(capone, fpg),
                                (torrio, fpg),etc.}
```

The definition of ‖join‖ uses relative set notation: informally, `{a | b1 ∈ s1 etc., & c1 etc.}` is read as the set of all `a` such that `b1` is a member of the set `s1`, etc. and `c1` is a condition, etc. The function `collect` is defined such that it returns a new binary-relation, containing one binary tuple `(x, image_x)` for each member of the projection of the left-hand column of `join_rel`, where `image_x` is the mathematical image of `x` under the relation `join_rel`. For example:

```
collect join_rel
=> {(capone, {bowery,fpg}),(torrio, {fpg}), etc.
```

As example of the use of the denotation of the word "join", consider: ‖join‖ ‖Five Points Gang‖

```
=> λq{x|(x,image_x) ∈ collect(join_rel) & q(image_x)}
                                        (λp fpg ∈ p)
=> {x|(x,image_x) ∈ collect(join_rel) &
                                (λp fpg ∈ p)(image_x)}
=> {x|(x,image_x) ∈ collect(join_rel) &
                                 (fpg ∈ image_x)}
=> {capone, torrio}
```

The resulting semantics is highly compositional: denotations of compound phrases and sentences are created using function application, according to the syntactic structure of the query. It should be noted that syntactic ambiguity is accommodated by having the parser generate more than one syntax tree each of which determines an order of application of the functions which are denoted by the words and phrases in the query. For example: one of the two syntactic parses of the query "Did Capone and Torrio join a gang?" would result in the following expression, which has the same meaning as the query "Did Capone join a gang and did Torrio join a (not necessarily the same) gang?"

```
(‖and‖‖Capone‖‖Torrio‖) (‖join‖ (‖a‖‖gang‖) )
```

which evaluates to `True` w.r.t. the definitions given above. We discuss ambiguity further in section 7.

The set-based FLMS semantics has the first five of the six properties discussed earlier. It can be implemented directly as part of a syntax-directed query processor for conventional relational databases in any programming language, but most easily in languages such as LISP, Miranda, Haskell, Scheme, ML or Python which support higher-order functions. Denotations of common nouns such as "thief" and intransitive verbs such as "smokes" are defined directly in terms of unary relations in the database. Relations such as join_rel, which are used in the denotations of transitive verbs, are defined directly in terms of binary-relations.

3. EVENT-BASED TRIPLE STORES

Before we discuss how to convert FLMS to a form that can be used with event-based triple stores, it is helpful to consider further how knowledge can be represented in such stores. First, we consider how to represent facts associated with intransitive verbs. For example, the fact that Capone was known to smoke. This can be represented as:

```
{(EV 1005, REL "type",      TYPE "smoke_ev"),
 (EV 1005, REL "subject",   ENT  "capone")}
```

Next, set membership which is the result of an action, e.g. the fact that Capone became a thief, can be represented by treating set membership as an event:

```
{(EV 1002,  REL "type",      TYPE "membership"),
 (EV 1002,  REL "subject",   ENT  "capone"),
 (EV 1002,  REL "object",    ENT  "thief")}
```

Now we can add the fact that he became a thief in 1908:

```
(EV 1002, REL "year", ENTNUM 1908)
```

Finally, consider set membership which is a consequence of an intrinsic property of an entity, e.g. the triples representing the fact that "Capone stole a car in 1918 in Manhattan" are:

```
{(EV 1004, REL "type",     TYPE   "steal_ev"),
 (EV 1004, REL "subject",  ENT    "capone"),
 (EV 1004, REL "object",   ENT    "car1"),
 (EV 1004, REL "year",     ENTNUM 1908),
 (EV 1004, REL "location", ENT    "Manhattan")}
```

In the above, we have not represented the fact that car1 is a car. To be consistent, this fact should be represented in a way that is similar to the way in which event 1002 represents the fact that Capone was a thief:

```
{(EV 1006, REL "type",      TYPE "membership"),
 (EV 1006, REL "subject",   ENT  "car1"),
 (EV 1006, REL "object",    ENT  "car")}
```

It is somewhat burdensome to have to treat membership of a set (e.g. car) which results from the "core" essence of an entity (e.g. car1) in a similar way to membership of a set which is contingent on an action. However, this allows us define denotation of all common nouns in the same way.

From the examples given above, one can see that when set membership (e.g. the set thief) is contingent on an action (e.g. steal), there could be some redundancy in the triple store. For example, the data represented by event 1002 could be derived from event 1004 data. We do not address this concern in this paper, as it has to do with how data from other data structures is converted to triple store data, and what deductive machinery accompanies the triple store.

4. RETRIEVING DATA FROM AN EVENT-BASED TRIPLE STORE

Before we introduce the new event-based semantics, we define some basic triple store retrieval operators. Given the limitations of space, rather than define the retrieval operators and our new semantics using the notation of lambda calculus and set theory, and then show how they can be implemented in a programming language, we give the definitions directly using the notation of the programming language Miranda. We choose Miranda for four reasons: 1) It has built-in list operators and a list comprehension construct which corresponds to the "relative set notation" that we used in denotations in FLMS (section 2.3). 2) Similar to MS and FLMS, our new semantics uses higher-order functions which can be defined directly in Miranda. 3) Miranda has a simpler syntax than other higher-order functional languages. 4) Given the declarative nature of Miranda, the definitions are executable specifications which allow us to test our ideas.

In Miranda:

```
- [x1,..,xn] is a list of n elements of the same type.

- #s is the length of the list s.

- f a1..an  returns the result of applying f to  a1..an

- member s x returns True if x is in the list s.

- (x1,..,xn) is a tuple of n values of different type.

- Lists are created using  list-comprehensions which
  have the general form:[values|generators;conditions]
  For example: [(x^2 | x <- [1..10], odd x]
                                    => [1, 9, 25, 49, 81]

- f a1..an = e defines f to be a function of n arguments
                        whose value is the expression e.

- n $f m allows f  to be used as an infix operator.

- Functions can be composed with the . operator:
       (f . g) x = f (g x)

- map f s applies f  to every member of the list s.

- New types can be defined using type constructors,e.g.

   field ::= EV num       |  ENT [char]  |  ENTNUM num
             TYPE [char]  |  REL [char]  |  ANY

   then EV 1000 is a value of type field
```

Note in function application brackets are used to establish the order of application, not to enclose arguments, e.g. `sqrt 9 + sqrt (2 + 2) => 5`. We begin by defining a triple store called `data` which we use as an example throughout the rest of the paper. Note that we have used type constructors `EV, REL` etc. (which we earlier referred to as "tags") in the definition of the triple store. Note also that the definition of `data` is part of the Miranda program that we built to test our semantics.

```
data =
 [(EV 1000, REL "type",      TYPE "born_ev"),
  (EV 1000, REL "subject",   ENT  "capone"),
  (EV 1000, REL "year",      ENTNUM 1899),
  (EV 1000, REL "location",  ENT  "brooklyn"),
  (EV 1001, REL "type",      TYPE "join_ev"),
  (EV 1001, REL "subject",   ENT  "capone"),
  (EV 1001, REL "object",    ENT  "fpg"),
  (EV 1002, REL "type",      TYPE "membership"),
  (EV 1002, REL "subject",   ENT  "capone"),
  (EV 1002, REL "object",    ENT  "thief"),
  (EV 1002, REL "year",      ENTNUM 1908 ),
  (EV 1003, REL "type",      TYPE "join_ev"),
  (EV 1003, REL "subject",   ENT  "capone"),
  (EV 1003, REL "object",    ENT  "bowery"),
  (EV 1004, REL "type",      TYPE "steal_ev"),
  (EV 1004, REL "subject",   ENT  "capone"),
  (EV 1004, REL "object",    ENT  "car_1"),
  (EV 1004, REL "year",      ENTNUM 1918),
  (EV 1004, REL "location",  ENT  "manhattan"),
  (EV 1005, REL "type",      TYPE "smoke_ev"),
  (EV 1005, REL "subject",   ENT  "capone"),
  (EV 1006, REL "type",      TYPE "membership"),
  (EV 1006, REL "subject",   ENT  "car_1"),
  (EV 1006, REL "object",    ENT  "car"),
  (EV 1007, REL "type",      TYPE "membership"),
  (EV 1007, REL "subject",   ENT  "fpg"),
  (EV 1007, REL "object",    ENT  "gang"),
  (EV 1008, REL "type",      TYPE "membership"),
  (EV 1008, REL "subject",   ENT  "bowery"),
  (EV 1008, REL "object",    ENT  "gang"),
  (EV 1009, REL "type",      TYPE "join_ev"),
  (EV 1009, REL "subject",   ENT  "torrio"),
  (EV 1009, REL "object",    ENT  "fpg"),
  (EV 1010, REL "type",      TYPE "membership"),
  (EV 1010, REL "subject",   ENT  "capone"),
  (EV 1010, REL "object",    ENT  "person"),
  (EV 1011, REL "type",      TYPE "membership"),
  (EV 1011, REL "subject",   ENT  "torrio"),
  (EV 1011, REL "object",    ENT  "person")]
```

We now define a basic retrieval function `getts` which returns triples from `data` which match given field value(s):

```
getts (a,ANY,ANY) = [(x,y,z) | (x,y,z) <- data; x = a]
getts (ANY,ANY,c) = [(x,y,z) | (x,y,z) <- data; z = c]
etc.
Example uses are:

getts (ANY,  "subject",  "torrio")
                => [(1009, "subject",  "torrio"),
                    (1011, "subject",  "torrio"),
                     etc.]

getts (1009, "type", ANY) => [(1009, "type", join_ev)]
```

Operators to extract one or more fields from a triple include:

```
first                (a,b,c)  = a
second               (a,b,c)  = b
third                (a,b,c)  = c
thirdwithfirst       (a,b,c)  = (c, a)   etc.
```

Operators which return sets of fields from sets of triples can be defined using the functions above and the function map:

```
firsts trips           = map first trips
thirds trips           = map third trips
thirdswithfirsts trips = map thirdwithfirst trips  etc.
```

We can now define more complex operators, such as:

```
get_subj_for_event ev
            = thirds (getts (ev, REL "subject", ANY))


get_subjs_for_events evs
            = concat (map get_subj_for_event evs)
```

```
Such that:

get_subjs_for_events [EV 1000, EV 1009]
                        => [ENT "capone",ENT "torrio"]
```

The function `get_members` returns all entities which are members of a given set:

```
 get_members set = get_subjs_for_events events
 where
 events_for_type_membership
   = firsts (getts (ANY,REL "type",TYPE "membership"))
 events_for_set_as_object
   = firsts (getts (ANY,REL "object", ENT set))
 events
   = intersect  events_for_type_membership
                            events_for_set_as_object

An example use of this operator is:

get_members "person" => [ENT "capone",ENT "torrio"]
```

Another useful operator is one which returns all of the subjects of an event of a given type:

```
 get_subjs_of_event_type event_type
 = get_subjs_for_events events
   where
   events
     = firsts (getts (ANY, REL "type", TYPE event_type))

For example:

   get_subjs_of_event_type "smoke" => [ENT "capone"]
```

5. A NEW SEMANTICS BASED ON TRIPLES AND EVENTS

5.1 Denotations of words

We begin with nouns. As in FLMS, the denotation of a noun is the set of entities which are members of the set associated with that noun. The `get_members` function returns that set as a list. Note that in the Miranda program, sets are implemented as lists. We use the term "set" when discussing the semantics and "list" when discussing the implementation of the triple-store operators. Note also, that from now on, instead of representing denotaions as, for example: ‖`person`‖, we define denotations as functions with an appropriate name, e.g. `person`. These denotations can then be applied to each other in the program, as shown on the next page, to create the meanings of more complex phrases.

```
    person    = get_members "person"
    gang      = get_members "gang"
    car       = get_members "car"
    thief     = get_members "thief"

e.g.  gang    => [ENT "fpg", ENT "bowery"]
```

Next, we consider intransitive verbs. The denotation of an intransitive verb is the set of entities which are subjects of an event of the type associated with that verb:

```
smoke         = get_subjs_of_event_type  "smoke_ev"

e.g   smoke   => [ENT "capone"]
```

Intransitive use of transitive verbs are similar:

```
steal_intrans      = get_subjs_of_event_type  "steal_ev"
steal_intrans      => [ENT"capone"]
```

As in FLMS, proper nouns denote functions which take a set of entities as argument and which return `True` if a particular entity is a member of that set, and `False` otherwise:

```
capone    setofents  = member setofents (ENT "capone")
torrio    setofents  = member setofents (ENT "torrio")
car_1     setofents  = member setofents (ENT "car_1)
fpg       setofents  = member setofents (ENT "fpg")
year_1908 setofents  = member setofents (ENTNUM 1908)
etc.
An example application:       capone smoke  => True
```

The quantifiers, "a", "one" , "two", "every", etc. and the conjoiners are defined in the same way as in FLMS:

```
a     nph vbph     = #(intersect  nph vbph) ~= 0
one   nph vbph     = #(intersect  nph vbph)  = 1
two   nph vbph     = #(intersect  nph vbph)  = 2
every nph vbph     = subset  nph vbph

nounand s t  = intersect s t
nounor  s t  = mkset (s ++ t)
that         = nounor

termand tmph1 tmph2
 = f  where
       f setofevs = (tmph1 setofevs) & (tmph2 setofevs)
termor tmph1 tmph2
 = f  where
       f setofevs = (tmph1 setofevs)\/(tmph2 setofevs)

An example application of the above is:

(capone $termor torrio) thief => True
```

Transitive verbs are more complex. We need something similar to the `image` in the FLMS approach. We can create "images" for an event `et` using the following:

```
make_image   et
= collect
  (concat [(thirdswithfirsts . getts)
              (ev, REL "subject", ANY)| ev <- events])
where
events = (firsts . getts)  (ANY, REL "type", TYPE et)

An example application:

    make_image "join_ev"
           => [(ENT "capone", [EV 1001, EV 1003]),
               (ENT "torrio", [EV 1009])]
```

We can now use `make_image` to define the denotation of a transitive verb associated with an event of a given type:

```
join
= f where
   f tmph
    = [subj|(subj,evs)<- make_image "join_ev";
             tmph(concat[(thirds.getts)
                (ev, REL "object", ANY)| ev <- evs])]
```

This definition is somewhat complex. We begin by noting that a termphrase is a syntactic category that includes proper nouns and determiner phrases such as "fpg", "a gang", "a gang that was joined by torrio", etc. The denotation of "join" is a function `f` such that when `f` is applied to a termphrase `tmph` (which is itself a function) it returns a list

of subjects each of which is associated with a set of events evs in the image of the join_ev, such that when tmph is applied to the list of objects of the events evs, the result is True, e.g.:

```
join (a gang) => [ENT "capone",ENT "torrio"]
```

ENT capone is in the result owing to the fact that the denotation of the termphrase (a gang) is a function which returns True when applied to the list of objects of the set of events associated with ENT "capone" in the image of event type join_ev. Similarly for ENT "torrio".

We can define the passive form of transitive verbs by replacing subject by object in the definition of make_image and use it to create a function make_passive_trans. For example:

```
joined_by = make_passive_trans "join_ev"

Example use:
joined_by (capone $termand torrio)  => [ENT "fpg"]
```

We conclude this sub-section by defining some "query" words:

```
which    nph vph     = intersect nph vph
how_many nph vph     = intersect nph vph
did      tph vbph    = "yes",if tph vbph = True
                     = "no", otherwise
who      vph         = which person vph
```

5.2 Prepositional phrases

Complex prepositional phrases, such as "in 1908 or 1918 in a city in Illinois" have typically been somewhat difficult to integrate into a compositional NL query semantics which allows arbitrarily-nested quantification (which our semantics does). We do not have any problems and can easily accommodate multiple prepositional phrases by having the parser convert the list of prepositional phrases to a possibly empty list of "prepositional pairs". Each pair consists of a REL value and a termphrase. For example, the phrase "in 1908 or 1918, in Manhattan" which consists of two prepositional phrases is converted to:

```
[(REL "year", year_1908 $termor year_1918),
 (REL "location", "manhattan")]
```

The definition of each transitive verb is redefined to make use of this list to filter the events which are in the image of the event-type associated with that transitive verb before the termphrase which is the argument to the denotation of the transitve verb is applied to the set of objects associated with the event. A recursive function called filter_ev applies each prepositional phrase in turn as a filter to each event:

```
steal' tmph preps
 = [ subj | (subj, evs) <- image_steal;
      tmph (concat
       [(thirds.getts) (ev, REL "object", ANY)
         | ev <- evs; filter_ev ev preps])]

filter_ev event [] = True
filter_ev event (prep:list_of_preps)
   = ((snd (prep)) ((thirds.getts)
                           (event,fst (prep),ANY)))
     & filter_ev event list_of_preps

for example:

steal' (a car)
   [(REL "year", year_1908 $termor year_1918),
    (REL "location", "manhattan")]

=> [ENT "capone"]
```

6. DEFINING WORDS INDIRECTLY AND EXAMPLE QUERIES

The meaning of some words can be defined in terms of words and phrases whose meanings are known. For example:

```
gangster = join (a gang)
```

Our EV-FLMS semantics has the six properties mentioned earlier. The answers to complex queries can be obtained from the meanings of their components by simple function application. For example, the query "Which gangster who stole a car in 1908 or 1918 in Manhattan, joined a gang which was joined by Torrio?" would be converted to the following functional expression by the parser, and then evaluated directly by the programming language in the same way as the expression 3 + (2 * 4) would be evaluated:

```
which
   (person $that
    (steal' (a car)
          [(REL "year", year_1908 $termor year_1918),
           (REL "location", "manhattan")]))
   (join (a (gang $that (joined_by torrio))))

  => [ENT"capone"]
```

The conversion, by the parser, of the word "in" to (REL "year") and (REL "location") in the two different contexts is clumsy and contravenes Montague's notion that words do not denote entities directly. We will improve this approach in futue work.

In our semantics, queries can contain arbitrarily-nested quantification. Termphrases with quantifiers ("a", "every", "some", "one" "two", etc.) can also appear in prepositional phrases. For example, if the data store held the appropriate triples, the following query can be processed "Who broke into a bar using a jimmy or a brick in two cities located in Illinois?"

7. USING A PARSER TO DISAMBIGUATE

Our new semantics has a one-to-one correspondence between the syntax rules defining the syntactic structure of the queries, and the semantic rules determining the order of application of the functional denotations. All phrases and words of a syntactic category have denotations (meanings) of the same semantic type, simplifying integration of the semantics with a parser to create a syntax-directed interpreter. We have already done this for the FLMS semantics and we are currently doing this for the semantics presented here.

There is insufficient space in this paper to discuss ambiguity in detail. In summary, our parser generates more than one syntax tree for ambiguous queries. For example, the query "Did Capone and Torrio join a gang?" would be parsed in two ways, resulting in the two expressions:

```
(capone  $termand  torrio) (join (a gang))
(a gang) (joined_by  (capone  $termand  torrio))
```

The first returns `True` if Capone and Torrio both joined a, not necessarily the same, gang, and the latter would only return `True` if at least one gang was joined by both Capone and Torrio.

8. RELATED WORK

Triple stores have been used in binary-relational databases since the 70's. A comprehensive survey of research on binary relational databases and triplestores, up to and including that carried out in the early 1980's, is provided in [10]. That paper also includes a description of a triple-based query language called WAROUT which appears to be one of the first SPARQL like query languages to have been developed.

Since the 80's, various attempts have been made to create user-friendly query interfaces to binary-relational triplestores. Early attempts include WAROUT mentioned above, pseudo natural-language interfaces [26], Prolog interfaces [29], and graphical visual interfaces [28], [24] and [25].

More recent interfaces to triplestores include the system of Mandreoli et al [23] on flexible query answering which returns best approximations to a query; the four systems (Semantic Crystal, Ginseng, NLP-reduce and Querix) of Kaufmann and Bernstein [1], [20], [19]; the AquaLog system of Lopez et al. [21]; the ORAKEL system of Cimiano et al. [3] which also uses a Montague-like semantics; the NQ system of Ran and Lencevicius [27]; the Pythia system of Unger and Cimiano which converts the NL query to an FLogic query [30]; the SQUALL system of Ferre [8] which is also based on Montague's linguistic approach; the system of Yahya et al [32]; the system of Damova et al. [5] which is also based on a formal lagic and which which converts NL queries to SPARQL using the Grammatical Framework (GF); the system of Hakimov et al [17] and the Metafrastes system of Embregts et al. [7].

The approach that we have presented in this paper is different from the work mentioned above in that we regard bracketed NL (e.g. English) queries as functional expressions using a formal denotational semantics, and then evaluate those expressions through direct reference to the triplestore using basic triple retrieval operators. We do not translate the NL query to any intermediate language such as SPARQL or FLogic.

This paper describes work which is part of a research project that has extended over several years [9], [13], [15], [11], [16] and [12]. The major contributions of this paper include 1) a detailed explanation of the development of the new semantics, 2) the method for dealing with multiple and complex prepositional phrases, and 3) Miranda program code showing how the event-based semantics can be implemented.

9. CONCLUSION AND FUTURE WORK

We have argued that 1) using events rather than entities as the subject of triples, and 2) treating (bracketed) NL queries as expressions of the lambda calculus that can be evaluated directly with respect to the triplestore, allows the creation of a denotational semantics for a wide range of NL queries, and also the construction of query processors as modular syntax-directed interpreters.

The semantics described in this paper is only a proof of concept and much remains to be done.

We have already started work on interfacing our semantics to remote semantic-web event-based triplestores and have built an on-line query interface. That work is described in an unpublished paper [14].

Our research group is planning to do the following over the next year: 1) improve our approach to prepositional phrases, 2) extend the semantics to accommodate aggregation and negation, 3) integrate the semantics with a parser using the SAIGA attribute grammar programming environment [16], 4) investigate the use of our query processor with existing (conventional) entity-based triple stores in the semantic web. This will require converting, as needed, some of the entity-based triples to event-based triples, 5) investigate the integration of the method of Walter et al [31] for mapping query words to appropriate URIs and building the denotations of words in real-time when the query is parsed, and 6) create a denotational semantics for Japanese and investigate the use of event-based triple stores as an intermediate knowledge representation format for language translation between English and Japanese.

We hope that this paper prompts discussion of the relative advantages and disadvantages of "entity-based" and "event-based" triplestores, and also prompts discussion of the pros and cons of converting NL queries to SPARQL before they are evaluated.

A possible way forward might be to have a 2-stage approach to the development of a powerful NL query processor: Stage I: use the approach described in this paper to develop a formal denotational semantics for a wide range of NL constructs including nested quantifiers, complex chained prepositional phrases, aggregation, negation, modality (such as "who believes that ...", aggregates, and temporal phrases (such as for what period of time...) Stage II: after the NL semantics has been developed, a translater could be built, based the semantics, to convert NL queries to SPARQL queries. Stage I would facilitate the development of the complex denotational semantics necessary to accommodate a wide range of NL queries, and STAGE II would allow the query processor to make use of the many methods that have been developed to optimize SPARQL queries.

10. ACKNOWLEDGMENTS

The authors acknowledge the support of the Natural Science and Engineering Council (NSERC) of Canada, and the reviewers for their comprehensive review and comments on this paper.

11. REFERENCES

[1] A. Bernstein, E. Kaufmann, and C. Kaiser. Querying the semantic web with ginseng: A guided input natural language search engine. In *Proceedings of the 15th Workshop on Information Technology and Systems (WITS 2005)*, pages 45–50, 2005.

[2] P. Blackburn and J. Bos. *Representation and Inference in Natural Language*. CSLI Publications, 2005.

[3] P. Cimiano, P. Haase, and J. Heizmann. Porting natural language interfaces between domains: an experimental user study with the orakel system. In *Proceedings of the 12th international conference on Intelligent user interfaces*, pages 180–189. ACM, 2007.

[4] J. Clifford, S. Abramsky, and C. van Rijsbergen. *Formal Semantics and Pragmatics for Natural Language Querying. Cambridge Tracts in Theoretical*

Computer Science 8. Cambridge University Press, Cambridge, 1990.

[5] M. Damova, D. Dannelles, R. Enache, M. Mateva, and A. Ranta. Natural language interaction with semantic web knowledge bases and lod. In *Towards the Multilingual Semantic Web*. Springer, 2013.

[6] D. Dowty, R. Wall, and S. Peters. *Introduction to Montague Semantics*. D. Reidel Publishing Company, Dordrecht, Boston, Lancaster, Yokyo, 1981.

[7] H. Embregts, V. Milea, and F. Frasincar. Metafrastes: A news ontology-based information querying using natural language processing. In *The 8th International Conference on Knowledge Management in Organizations*, pages 313–324. Springer, 2014.

[8] S. Ferre. Squall: A controlled natural language for querying and updating rdf graphs. In *Proceedings of CNL 2012*, pages 11–25. LNCS 7427, 2012.

[9] R. Frost and J. Launchbury. Constructing natural language processors in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.

[10] R. A. Frost. Binary-relational storage structures. *The Computer Journal*, 25(3):358–367, 1982.

[11] R. A. Frost. Realization of natural language interfaces using lazy functional programming. *ACM Comput. Surv.*, 38(4):1–54, 2006.

[12] R. A. Frost, B. S. Amour, and R. Fortier. An event based denotational semantics for natural language queries to data represented in triple stores. In *Proceedings of ICSC 2013*. IEEE, Sept. 2013.

[13] R. A. Frost and P. Boulos. An efficient compositional semantics for natural-language database queries with arbitrarily-nested quantification and negation. In *Conference Proceedings of Advances in Artificial Intelligence, the 15th Conference of the Canadian Society for Computational Studies of Intelligence, AI 2002*, pages 252–267. LNCS 2338, 2002.

[14] R. A. Frost, J. Donais, E. Matthews, and R. Stewart. A denotational semantics for natural langauge query interfaces to semantic web triplestores. In *Submitted for publication*, 2014.

[15] R. A. Frost and R. Fortier. An efficient denotational semantics for natural language database queries. In *Proceedings of Natural Language Processing and Information Systems, 12th International Conference of Applications of Natural Langauge to Information Systems, NLDB 2007*, pages 12–24. LNCS 4592, 2007.

[16] R. Hafiz and R. Frost. Lazy combinators for executable specifications of general attribute grammars. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 167–182. ACM-SIGPLAN, Jan. 2010.

[17] S. Hakimov, H. Tunc, M. Akimaliev, and E. Dogdu. Semantic question answering system over linked data using relational patterns. In *Proc. of the Joint EDBT/ICDT 2013 Workshops*, pages 83–88. ACM, 2013.

[18] H. Hendricks. *Studied Flexibility: catgeories and types in syntax and semantics*. Doctoral Dissertation, Universiteit van Amsterdam, 1993.

[19] E. Kaufmann and A. Bernstein. Evaluating the usibility of natural language query languages and interfaces to semantic web knowledge bases. *Web Semantics - Science, Services and Agents on the World Wide Web*, 8(4):377–393, Nov 2009.

[20] E. Kaufmann, A. Bernstein, and R. Zumstein. Querix: A natural language interface to query ontologies based on clarification dialogs. In *Proceedings of the 5th International Semantic Web Conference*, Nov 2006.

[21] V. Lopez, V. Uren, E. Motta, and M. Pasin. Aqualog: An ontology-driven question answering system for organizational semantic intranets. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):72–105, 2007.

[22] M. G. Main and D. Benson. Denotational semantics for a natural language question answering program. *Computational Linguistics*, 9(1):11–21, 1983.

[23] F. Mandreoli, R. Martoglia, W. Penzo, and G. Villani. Flexible query answering on graph-modeled data. In *Proceedings of the 12th International Conference on Extending Database Technology*, pages 216–227. EDBT, March 2009.

[24] J. A. Mariani and R. Lougher. Triplespace an experiment in 3d graphical interface to a binary-relational database. *Interacting with Computers*, 4:147–162, 1992.

[25] N. Memon and H. Larson. Investigative data mining toolkit: a software prototype for visualizing, analyzing and destabilizing terrorist networks. In *Proceedings Visualizing Network Information, RTO-MP-IST*, pages 1–24, 2006.

[26] N.Nicholson. *The design of a user-interface to a deductive database: a sentence based approach*. PhD thesis Dept. of Computer Science - Birkbeck College, University of London, 1988.

[27] A. Ran and R. Lencevicius. Natural language query system for rdf repositories. In *Proceedings of the 7th International Symposium on Natural Language processing*, pages 1–6. SNLP, 2007.

[28] Smith and King. Incrementally visualizing criminal networks. In *Proceedings of the Sixth International Conference on Information Visualisation*, 2002.

[29] S. Todd. An interface from prolog to a binary relational database. *Prolog and databases - implementations and new directions*, pages 108–117, 1989.

[30] C. Unger and P. Cimiano. Pythia: Compositional meaning construction for ontology-based question answering on the semantic web. In *NLDB 2011, LNCS 6716*, pages 153–160, 2011.

[31] S. Walter, C. Unger, P. Cimiano, and D. Bär. Evaluation of a layered approach to question answering over linked data. In *The Semantic Web–ISWC 2012*, pages 362–374. Springer, 2012.

[32] M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, , and G. Weikum. Natural language questions for the web of data. In *The 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 379–390. ACL, July 2012.