An event based denotational semantics for natural language queries to data represented in triple stores

Richard A. Frost and Bryan St. Amour
School of Computer Science
University of Windsor
Windsor, Canada

Randy Fortier
Department of Computer Science
Thompson Rivers University
Kamloops, Canada

Abstract—Binary-relational and graph-based databases are more flexible than conventional relational databases because the type of data to be stored does not need to be known in advance. In many implementations of these systems the data is held in a “triple store”. This paper presents a new event-based denotational semantics which can be used as the basis for natural-language (NL) query interfaces to triple stores. This paper is also a first step towards an NL semantics which can be used for building NL query interfaces to RDF semantic web data.

Keywords—binary-relational database, triple store, natural-language query, denotational semantics, prepositional phrases, Montague semantics.

I. INTRODUCTION

In some applications the type of data to be stored is not known in advance (e.g. criminal investigations). Therefore, rather than store such data in a conventional relational database, which would require continuous changes to be made to the database schema and to queries and report definitions, some researchers, e.g. [1] and [2], have argued that the data should be represented by binary relations or graphs and stored in “triple” stores. For example:

\[
(EV 1000, \text{REL} \ "type", \ \text{TYPE} \ "born\_ev"),
\]

\[
(EV 1000, \text{REL} \ "subject", \ \text{ENT} \ "capone"),
\]

\[
(EV 1000, \text{REL} \ "date", \ \text{ENTNUM} \ 1899)
\]

We can now add the fact that Capone was born in Brooklyn with the triple: \( (EV 1000, \text{REL} \ "location", \ \text{ENT} \ "brooklyn") \).

Binary-relational databases and triple stores have been around since the ’80s and various attempts have been made to create user-friendly query interfaces to them. These include pseudo natural-language (NL) interfaces [3], Prolog interfaces [4], and graphical visual interfaces [2] and [5]. However, no one has yet created a wide-coverage natural-language query interface to triple stores which is based on a formal easily-implementable natural-language semantics.

In this paper, we present an event-based semantics for natural language which is denotational (every word and every phrase has a well-defined mathematical meaning), is compositional (the meaning of a composite expression is created through simple operations from the meaning of its components), is referentially transparent (the meaning of a word or phrase, after syntactic disambiguation, is the same no matter what context it appears in), and has a Montagovian correspondence [6] between the syntactic rules and the semantic rules (there is a one-to-one correspondence between the syntactic rules describing the structure of a phrase and the semantic rules describing how the meaning of the phrase is computed from the meanings of its components).

The four properties above are important because they ensure that the semantics can be used as a basis to easily build natural language query processors as highly modular syntax-directed interpreters, or as executable attribute grammars, which can answer a wide range of queries such as the following with respect to data stored in a triple-store:

“Which gangster who stole a car in 1899 or 1908 joined a gang which was joined by Torrio?”

“Was every gangster who joined the four points gang born in Brooklyn?”

Perhaps more importantly, the semantics that we present here is a first step towards a semantics that might be used as a basis for NL query interfaces to RDF semantic-web data which is also stored in the form of triples (but which is not always event based). There is insufficient space in this paper to discuss this further, but it is the focus of our current research.

The semantics described in this paper is based on NL semantics that we have developed previously for use with conventional databases [7]. That semantics was based on a modified version of Montague Semantics (MS) [6]. Our new semantics is also based on MS but differs in that is an event-based, rather than an entity-based, semantics. It also has wider coverage of English than our earlier semantics in that it can accommodate propositional phrases.

We describe the semantics using notation from set theory and recursive function theory. We have implemented a prototype of the semantics directly as function definitions in the programming language Miranda (details can be obtained by contacting the first author). However, the semantics can be easily implemented in any programming language that supports higher-order functions (e.g. Lisp, Haskell, Scheme, ML, and Python) and less easily in most other programming languages.
II. Retrieving Data From A Triple Store

We begin by defining a triple store called \textit{data}, which we use as an example throughout the rest of the paper. \textit{data} =

\begin{align*}
&\{EV\ 1000, \ REL \ "type", \ TYPE \ "born\ ev",\} \\
&(EV\ 1000, \ REL \ "subject", \ ENT \ "capone"), \\
&(EV\ 1000, \ REL \ "date", \ 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 \ "date", \ ENTNUM\ 1908), \\
&(EV\ 1003, \ REL \ "type", \ TYPE \ "join\ ev"), \\
&(EV\ 1003, \ REL \ "subject", \ ENT \ "capone"), \\
&(EV\ 1003, \ REL \ "object", \ ENT \ "torrio"), \\
&(EV\ 1004, \ REL \ "type", \ TYPE \ "steal\ ev"), \\
&(EV\ 1004, \ REL \ "subject", \ ENT \ "capone"), \\
&(EV\ 1004, \ REL \ "date", \ ENTNUM\ 1908), \\
&(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\ 1"), \\
&(EV\ 1007, \ REL \ "type", \ TYPE \ "membership"), \\
&(EV\ 1007, \ REL \ "subject", \ ENT \ "fpg"), \\
&(EV\ 1007, \ REL \ "object", \ ENT \ "gang\ set"), \\
&(EV\ 1008, \ REL \ "type", \ TYPE \ "membership"), \\
&(EV\ 1008, \ REL \ "subject", \ ENT \ "browey"), \\
&(EV\ 1008, \ REL \ "object", \ ENT \ "gang\ set"), \\
&(EV\ 1009, \ REL \ "type", \ TYPE \ "membership"), \\
&(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\ set"), \\
&(EV\ 1011, \ REL \ "type", \ TYPE \ "membership"), \\
&(EV\ 1011, \ REL \ "subject", \ ENT \ "torrio"), \\
&(EV\ 1011, \ REL \ "object", \ ENT \ "person\ set")
\end{align*}

Next, we define the basic retrieval function \textit{getts} which returns sets of triples matching given field value(s). Note that we use "set-builder" notation, e.g. \{\{x^2 \mid x \in \mathbb{R} \land x > 0\}\} is the set of squares of all positive real numbers. Note also that \(a \in s\) returns True if \(a\) is a member of \(s\) and False otherwise.

\textit{getts} (a, ANY, ANY) = \{(x,\ y,\ z) \mid \{(x,\ y,\ z) \in \textit{data} \land x = a\}\}

\textit{getts} (ANY, b, ANY) = \{(x,\ y,\ z) \mid \{(x,\ y,\ z) \in \textit{data} \land y = b\}\}

\textit{getts} (ANY, b, c) = \{(x,\ y,\ z) \mid \{(x,\ y,\ z) \in \textit{data} \land y = b \land z = c\}\}

e tc.

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

\begin{align*}
\text{first} & \quad (a,\ b,\ c) = a \\
\text{third} & \quad (a,\ b,\ c) = c \\
\text{thirdwithfirst} & \quad (a,\ b,\ c) = (c,\ a)
\end{align*}

We can now define more complex operators, using the set-theory function \textit{map}, where for example:

\text{map} (\{1,2,3\}) \Rightarrow \{2,4,6\}.

\textit{get_subjs_for_event} ev = \text{map} \ \text{third} \ \text{(getts} (ev, \ REL \ "subject", \ ANY))

\text{get_subjs_for_events} evs = \bigcup \text{(map \ \text{get_subjs_for_event} evs)}

Where \(\text{US}\) in this context returns the set obtained by uniting all of the sets which are elements of the set \(\text{S}\).

e g. \ \text{get_subjs_for_events} \{EV\ 1000,\ EV\ 1009\} \Rightarrow \{\text{ENT} \ "capone", \ \text{ENT} \ "torrio"\}

Another useful operation returns all entities which are members of a given set. We can think of this as retrieving all entities which are subjects of an event of type "membership" where the given set is the object of the event:

\text{get_members} \ \text{set} = \text{get_subjs_for_events} \ \text{events}

\text{events} = \text{events_of_type_membership} \ \cap \ \text{events_with_set_as_object}

\text{events_of_type_membership} = \text{map} \ \text{first} \ \text{(getts} (ANY, \ REL \ "type", \ TYPE \ "membership"))

\text{events_with_set_as_object} = \text{map} \ \text{first} \ \text{(getts} (ANY, \ REL \ "object", \ ENT \ set))

\ e g. \ \text{get_members} \ "gang\ set" \Rightarrow \{\text{ENT} \ "fpg", \ \text{ENT} \ "browey"\}

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

\text{get_subjs_of_event_type} ev_type = \text{get_subjs_for_events} evs

\text{evs} = \text{map} \ \text{first} \ \text{(getts} (ANY, \ REL \ "type", \ TYPE \ ev_type))

\ e g. \ \text{get_subjs_of_event_type} "smoke\ ev" \Rightarrow \{\text{ENT} \ "capone"\}

These definitions can be optimised depending on how the data is stored and retrieved from the triple store.

III. The Semantics

In our semantics, the denotations (meanings) of words and phrases are sets of entities, sets of events, Boolean values or functions defined over these values. Note that, as in MS, some of the denotations are higher-order functions which take functions as arguments and/or return a function as result.

In the following, we use bold italic font for the denotation of a word. That is, \(\chi\) is the meaning of the word "xyz". In some cases \(\chi\) includes the word and an indication of how the word is used, as determined by the parser, e.g. \textit{steal\_intrans}, and \textit{date\_1890}. 

143
The denotations of nouns are sets of entities:

\[
\begin{align*}
\text{person} &= \text{get\_members } \langle \text{person\_set} \rangle \\
\text{gang} &= \text{get\_members } \langle \text{gang\_set} \rangle \\
\text{car} &= \text{get\_members } \langle \text{car\_set} \rangle \\
\end{align*}
\]
e.g. \( \text{gang} \Rightarrow \{ \text{ENT } \text{"fpg"}, \text{ENT } \text{"bowery"} \} \)

The denotations of intransitive verbs (and the intransitive use of transitive verbs) are also sets of entities:

\[
\begin{align*}
\text{smokes} &= \text{get\_subs\_of\_event\_type } \langle \text{smoke\_ev} \rangle \\
\text{steal\_intrans} &= \text{get\_subs\_of\_event\_type } \langle \text{steal\_ev} \rangle \\
\end{align*}
\]
e.g. \( \text{smokes} \Rightarrow \{ \text{ENT } \text{"capone"} \} \)

Following Montague’s insight, proper nouns do not denote entities directly. Here, proper nouns denote functions which take a set of entities as argument and which return \( \text{True} \) if a particular entity is a member of that set, and \( \text{False} \) otherwise:

\[
\begin{align*}
\text{capone } \text{setofents} &= \langle \text{ENT } \text{"capone"} \rangle \\
\text{torrio } \text{setofents} &= \langle \text{ENT } \text{"torrio"} \rangle \\
\text{fpg } \text{setofents} &= \langle \text{ENT } \text{"fpg"} \rangle \\
\end{align*}
\]
e.g. \( \text{capone } \text{smokes} \Rightarrow \text{True} \)

Note the compositionality and the Montagovian correspondence: the meaning of the phrase “Capone smokes” is obtained by applying the function which is the denotation of “Capone” to the set of entities which is the denotation of the word “smokes”. The meaning of all phrases in our semantics is obtained by function application where the order of application is determined by the parser and is closely related to the syntactic structure of the phrase.

The denotations of quantifiers, “a”, “two”, “every” etc. are as follows, where \( \#s \) returns the size of the set \( s \):

\[
\begin{align*}
\text{a } \text{nph } \text{vbph} &= \#(\text{nph } \cap \text{vbph}) = 0 \\
\text{two } \text{nph } \text{vbph} &= \#(\text{nph } \cap \text{vbph}) = 2 \\
\text{every } \text{nph } \text{vbph} &= \text{nph } \subseteq \text{vbph} \\
\end{align*}
\]

The denotations of conjoiners for nouns are:

\[
\begin{align*}
\text{noun\_and } s \ t &= s \cap t \\
\text{noun\_or } s \ t &= s \cup t \\
\text{that} &= \text{noun\_or} \\
\end{align*}
\]

The denotations of conjoiners of termphrases are:

\[
\begin{align*}
\text{term\_and } \text{tmph1 } \text{tmph2} &= f \text{ where } \\
& \quad f \text{ setofevs} = (\text{tmph1 setofevs}) \& (\text{tmph2 setofevs}) \\
\text{term\_or } \text{tmph1 } \text{tmph2} &= f \text{ where } \\
& \quad f \text{ setofevs} = (\text{tmph1 setofevs}) \lor (\text{tmph2 setofevs}) \\
\end{align*}
\]

For example (where \( \text{Sf} \) indicates that \( f \) is used in infix mode):

\[
\langle \text{capone } \text{Sterm\_or } \text{torrio} \rangle \text{ smoke } \Rightarrow \text{True}
\]

The denotations of transitive verbs are complex and quite different from both MS and our previous semantics. They were not easy to derive, and will take some time to understand. We begin by creating “images” of an event type.

\[
\begin{align*}
\text{make\_image } e t &= \text{collect } (\cap \{\text{map thirdwithfirst} \\
& \quad (\text{getts } (\text{ev, REL } \text{"subject"}, \text{ANY}) ) \mid \text{ ev } \in \text{ events})\} \\
& \quad \text{ where } \\
& \quad \text{events } = \text{map } \text{first} (\text{getts } (\text{ANY, REL } \text{"type"}, \text{TYPE } e t)) \\
\end{align*}
\]

e.g. \( \text{make\_image } \langle \text{join\_ev} \rangle \Rightarrow \{ \langle \text{ENT } \text{"capone"}, \langle \text{EV } 1001, \langle \text{EV } 1003 \rangle \rangle, \langle \text{ENT } \text{"torrio"}, \langle \text{EV } 1009 \rangle \rangle \} \)

The \text{collect} function is defined such that when applied to a binary relation, it “collects” values from pairs:

\[
\begin{align*}
\text{collect rel} &= \{ (x, y) \mid (x, y) \in \text{ rel} \} \mid (x, z) \in \text{ rel} \\
\end{align*}
\]
e.g. \( \text{collect } \{ (a,2), (b,3), (a,1), (c,4), (a,7) \} \Rightarrow \{ (a, \{2,1,7\}), (b,\{3\}), (c,\{4\}) \} \)

Note that our program implementation of \text{collect} is significantly more efficient than the definition above might suggest.

We can now use \text{make\_image} to define the denotation of a transitive verb associated with an event of a given type. However, rather than define the denotation of every transitive verb through the explicit use of \text{make\_image}, we abstract the details of the definition into a higher order function \text{make\_trans} which can then be used to easily define the denotation of any transitive verb by giving \text{make\_trans} the event type.

\[
\begin{align*}
\text{make\_trans } \text{event\_type} &= f \text{ where } \\
& \quad f \text{ tmph} = \{ \text{subj} \mid (\text{subj, evs}) \in (\text{make\_image } \text{ event\_type}) \\
& \quad \text{& tmph } \cap (\text{map } \text{thirds} (\text{getts } (\text{ev, REL } \text{"object"}, \text{ANY}))) \mid (\text{ev } \in \text{ evs})\} \\
\end{align*}
\]

The definition of \text{make\_trans} is somewhat complex. It takes an event type \text{event\_type} as argument and returns a function \( f \) which is the denotation of the verb associated with \text{event\_type}. When \( f \) is applied to a termphrase \text{tmph} (which is itself a function) it returns a list of subjects each of which is associated with a set of events \text{evs} in the image of \text{event\_type}, such that when \text{tmph} is applied to the list of objects of the events \text{evs}, the result is \text{True}. See below for an explanation.
with respect to an application example of the denotation of the transitive verb “join”. Here is an example use of make_trans:

\[
\text{join} = \text{make_trans} \ "\text{join}_ev"
\]

e.g. \(\text{join (a gang)} \Rightarrow \{\text{ENT "capone"}, \text{ENT "torrio"}\}\)

\text{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 \text{ENT "capone"} in the image of the event type “join\_ev”. Similarly for \text{ENT "torrio"}.

We can define the passive form of transitive verbs by replacing subject by object in the definition of make_image and defining a new function make_passive_trans.

e.g. \(\text{joined}_by = \text{make_passive_trans} \ "\text{join}_ev"
\)

e.g. \(\text{joined}_by (\text{capone} \text{ Stem_and} \text{torrio}) \Rightarrow \{\text{ENT "fpg"}\}\)

Prepositional phrases, such as “in Brooklyn”, and “in 1908” have typically been somewhat difficult to integrate into a compositional NL semantics which allows arbitrarily-nested quantification (as our semantics does). However, we can easily accommodate prepositional phrases by providing different denotations of transitive verbs depending on the type of prepositional phrases they are used with. For example, to accommodate prepositional phrases involving dates with the transitive verb “steal”, we use the following definition:

\[
\text{steal\_with\_time} \ \text{tmph} \ \text{date} = \{\text{subj} \ (\text{subj}, \text{evs}) \in \text{image\_steal} \ & \ \text{tmph} \ (\bigcup \ \{\text{thirds} (\text{getts (ev,REL "object", ANY})) \ | \ \text{ev} \in \text{evs} \\
& \ \text{date}(\text{thirds} (\text{getts (ev,REL "date", ANY}))\})\}
\]

The \text{date} argument is used to “filter” the events.

e.g. \(\text{steal\_with\_time (a car) (date\_1908) } \Rightarrow \{\text{ENT "capone"}\}\)

This first attempt to accommodate prepositional phrases is somewhat clumsy and we are currently generalizing our semantics so that we have a single definition of each transitive verb which takes a list of prepositional arguments (created by the parser) which are used to filter events.

Query words are defined as follows ( #n is the length of n):

\[
\begin{align*}
\text{which nph vph} & = \text{intersect nph vph} \\
\text{how\_many nph vph} & = \#(\text{intersect nph vph}) \\
\text{who vph} & = \text{which person vph} \ etc.
\end{align*}
\]

The declarative nature of our semantics allows the meaning of words to be defined in terms of words and phrases whose meanings have already been defined.

e.g. \(\text{gangster} = \text{join (a gang)}\)

IV. CONCLUDING COMMENTS AND FUTURE WORK

We are creating an NL query processor by integrating the semantics with a parser using an executable attribute-grammar environment [8]. The parser will convert the query: “Which gangster who stole a car in 1899 or 1908 joined a gang which was joined by Torrio?” to the following semantic expression:

\[
\begin{align*}
\text{which (gangster)} & \ \text{Stest (steal\_with\_time (a car))} \\
& \ (\text{date\_1908} \ \text{Sterm_or} \ \text{date\_1899}) \\
& \ (\text{join (a (gang Stest (joined\_by torrio))))})
\end{align*}
\]

which evaluates to give: \{\text{ENT "capone"}\}.

The major contribution of our work is the creation of a computer-implementable Montague-like formal semantics of natural-language with an explicit denotation of transitive verbs which can accommodate arbitrary-nested quantification and prepositional phrases.

Although the semantics has been defined with respect to data stored in binary-relational event-based triple-stores, it has much more general use because, in our opinion, any data stored in any format can be converted to an event-based triple store. The intriguing question is whether our new semantics might find use in the development of natural-language query interfaces to RDF semantic-web data.

ACKNOWLEDGMENT

The authors acknowledge the support of the Natural Science and Engineering Council (NSERC) of Canada.

REFERENCES


