High Accuracy Question Answering via Hybrid Controlled Natural Language

Tiantian Gao, Paul Fodor, Michael Kifer
Department of Computer Science
Stony Brook University
Stony Brook, NY, USA
tiagao@cs.stonybrook.edu, pfodor@cs.stonybrook.edu, kifer@cs.stonybrook.edu

***Abstract*—Knowledge representation and reasoning (KRR) is key to the vision of the intelligent Web. Unfortunately, wide deployment of KRR is hindered by the difficulty in specifying the requisite knowledge, which requires skills that most domain experts lack. A way around this problem could be to acquire knowledge automatically from documents. The difficulty is that, KRR requires high-precision knowledge and is sensitive even to small amounts of errors. Although most automatic information extraction systems developed for general text understandings have achieved remarkable results, their accuracy is still woefully inadequate for logical reasoning. A promising alternative is to ask the domain experts to author knowledge in *Controlled Natural Language* (*CNL*). Nonetheless, the quality of knowledge construction even through CNL is still grossly inadequate, the main obstacle being the multiplicity of ways the same information can be described even in a controlled language. Our previous work addressed the problem of high accuracy knowledge authoring for KRR from CNL documents by introducing the Knowledge Authoring Logic Machine (KALM). This paper develops the *query* aspect of KALM with the aim of getting high precision answers to CNL questions against previously authored knowledge and is tolerant to linguistic variations in the queries. To make queries more expressive and easier to formulate, we propose a *hybrid* CNL, i.e., a CNL with elements borrowed from formal query languages. We show that KALM achieves superior accuracy in semantic parsing of such queries.**

***Index Terms*—Controlled Natural Language, Question Answering**

I. Introduction

The vision of Web intelligence is that complex knowledge will be available in an open format, such as RDF/OWL [1], [2] or RIF [3], [4], and will be queriable via standard protocols. The most popular Web query language is SPARQL [5], designed for the simplest form of Web knowledge, RDF. In contrast, RIF-style rules are largely absent from the Web. Despite the existence of some RIF-aware systems, a major obstacle is that specifying knowledge via rules requires skilled knowledge engineers, who are in short supply.

A promising idea, but one that is hard to realize, is to extract the requisite knowledge from text. Despite the impressive advances in this area (e.g., various Open IE systems [6], [7]), the technology is still far from being usable: the accuracy of the extracted knowledge is too low for logic knowledge bases, which are notoriously sensitive to errors.

A more feasible approach at present is to *author* knowledge via *controlled natural languages* (CNLs) [8], such as Attempto Controlled English (ACE) [9] or Processable English (PENG) [10]. These CNLs are fairly rich, yet restricted languages, and algorithms exist to accurately convert the sentences they accept into sets of logical facts that can be queried. Unfortunately, CNL systems do very limited semantic analysis of sentences, and they do not recognize when sentences that have the same meaning are expressed via different syntactical forms or using different language idioms. For instance, they would translate *Mary buys a car*, *Mary is the purchaser of a car*, *Mary makes a purchase of a car*, and many other equivalent sentences into different sets of facts that are not logically related. Thus, if any of these sentences is entered into the knowledge base, the reasoner would fail to answer questions like *Who purchases a car?* or *Who is the buyer of a car?* Clearly, this is a serious obstacle to using CNL as input to logical reasoning systems.

Aim of this work. Our previous work developed a high-accuracy knowledge authoring framework called *Knowledge Authoring Logic Machine* (KALM) [11], which standardizes the meaning of CNL sentences that represent logical facts and solves the above-mentioned semantic problems, achieving the accuracy of 96% — far exceeding other systems. The present work develops the query aspect of KALM to provide high accuracy query service for the previously developed high accuracy knowledge authoring service. A forthcoming rule authoring component will make KALM a fairly comprehensive framework for reasoning about human-authored knowledge.

KALM extends CNL queries with elements of formal query languages, which we call a *hybrid* syntax. KALM uses a touch of hybrid syntax because CNL cannot express many concepts in query languages, while full natural language is too ambiguous and cumbersome to be used for querying (cf. variables, aggregation, many types of sub-queries).

Contributions. The contributions of this paper are four-fold:

(a) A hybrid CNL-based language for authoring queries.
(b) A semantic parser that disambiguates CNL queries by mapping semantically equivalent queries into *unique logical representation for queries* (ULRQ).
(c) *Explainability*: the approach makes it possible to explain both why particular meanings are assigned to queries and also why certain answers are chosen (or not).
(d) A KALM prototype that supports authoring of facts and

querying—both with very high accuracy.[1]

Organization. Section II provides the required background material, including elements of the KALM framework. Section III introduces the ACE-based query language of KALM. Section IV describes query processing in KALM. In Section V, we report the experiments in support of our claim of high accuracy of query processing in KALM. Section VI presents related work and Section VII concludes the paper.

II. Preliminaries

A. Attempto Controlled English

Attempto Controlled English (ACE) is a CNL for knowledge representation. It is designed as a subset of English with restricted grammar and a set of interpretation rules that determine the unique meaning of each sentence. Despite the restrictions, ACE is quite general and expressive, and requires little training to learn how to paraphrase natural language sentences into CNL. ACE enables domain experts who lack the experience in logic to write logical statements in controlled English and the Attempto Parsing Engine (APE) translates CNL sentences into a logical form called *Discourse Representation Structure* (*DRS*) [12]. DRS has 7 predicates: `object`, `predicate`, `property`, `modifier_adv`, `modifier_pp`, `relation`, and `has_part`. For instance, an `object`-fact represents an entity—a noun-word with some properties (such as countable or uncountable quantity). A `predicate`-fact represents an event—a verb-word and its participating entities (e.g., *subject* and *object*). Each fact in DRS has a word index that refers to the relevant word in the original sentence. Consider the sentence *A customer buys a watch*[2] and its DRS:

```
object(A,customer,countable,na,eq,1)-1/2.
object(B,watch,countable,na,eq,1)-1/5.
predicate(C,buy,A,B)-1/3.
```

where the identifiers `A` and `B` represent the *customer*- and *watch*-entities, respectively, and `C` refers to the *buy*-event. In an `object`-fact, the second argument represents the stem form of the word the fact represents. The rest of the arguments are properties of this entity. In a `predicate`-fact, the third argument is the subject of the event, and the fourth argument represents the object of the event. In our example, the identifier `A` (resp. `B`) of the third fact indicates that the *customer* (resp. *watch*) is the subject (resp. object) of the event. An appendage like `-1/5` in the second `object`-fact says that `watch` is the fifth word is the first sentence.

B. BabelNet

BabelNet [13] is a multilingual knowledge base of words and concepts that integrates multiple sources of linguistic and general knowledge, including WordNet [14], DBPedia [15], Wikidata [16], and others. Like WordNet, BabelNet groups the words that express the same meaning into *synset*s and *gloss*es describe the meaning of these synsets. Synsets are linked by *semantic relations*, such as *hypernym*, *hyponym*, *holonym*, etc. Syntactic connections may also have weights, denoting the degrees of relevance between the synsets. As a result, BabelNet is a huge knowledge graph, where nodes are BabelNet synsets and edges represent the semantic relationships and weights.

BabelNet can be used for many tasks, including word sense disambiguation, measuring semantic similarity between synsets, and so on. At the same time, this knowledge graph contains a fair amount of noise, faulty and missing information, which makes many of the semantic analysis tasks quite challenging.

C. Frame-based Semantic Model of English Sentences

KALM introduced a FrameNet-inspired [17] ontology, *FrameOnt*, which endows English sentences with a frame-based semantic model. *FrameOnt* uses *frames*, described by *frame roles* and other components, to capture the meaning of English sentences. *FrameOnt* frames are captured formally as Prolog-facts (called *logical frames*), which specify the different roles of the frames. For instance, the `Commerce_Buy` frame is shown below:

```
fp(Commerce_Buy,[
   role(Buyer,[bn:00014332n],[]),
   role(Seller,[bn:00053479n],[]),
   role(Goods,[bn:00006126n,bn:00021045n],[]),
   role(Recipient,[bn:00066495n],[]),
   role(Money,[bn:00017803n],[currency])]).
```

In each `role`-term, the first argument is the name of the role and the second is a list of *role meanings* represented via BabelNet synset IDs. There can be several such meanings. For instance, the role `Goods` above can mean *an article of commerce* or *an article of sale*. These two concepts have different synsets in BabelNet. The third argument of a `role`-term is a list of constraints on that role. In the above frame, `Money` has a type constraint. Based on the above description, the meaning of sentences like *Mary buys a car*, *Mary makes a purchase of a car*, *Mary is a buyer of a car* is represented by the `Commerce_Buy` frame, where *Mary* fills the `Buyer` role and *car* fills the `Goods` role of the frame.

Each frame also has a set of *lexical units* and *logical valence patterns* (or *lvps*) that are used to extract instances of frames from CNL sentences. We call the entities extracted from sentences that correspond to the respective frame roles *role-filler* words. A lexical unit is an English word, with its part-of-speech, which represents a situation in which the frame can be "triggered." An lvp is represented as a Prolog-fact that specifies a syntactic context in which role-filler words and the lexical unit can occur in a sentence. All these lvps are constructed automatically, via *learning* linguistic structures from annotated training sentences that are marked with the frame type, lexical unit, and relevant roles. For instance, the `Commerce_Buy` frame has this lvp among others:

```
lvp(buy,v,Commerce_Buy, [
  pattern(Buyer,verb->subject,required),
  pattern(Goods,verb->object,required),
  pattern(Recipient,verb->pp(for)->dep,optnl),
  pattern(Money,verb->pp(for)->dep,optnl),
  pattern(Seller,verb->pp(from)->dep,optnl)]).
```

The first three arguments of an `lvp`-fact identify the lexical unit, its part of speech, and the frame. The fourth argument is a set of `pattern`-terms, each having three parts: the name of a role, a grammatical pattern, and the required/optional flag. The *grammatical pattern* determines the grammatical context in which the lexical unit, a role, and a role-filler word can

[1] Available at https://github.com/tiantiangao7/kalm under Apache license. The queries used for testing are found at https://datahub.csail.mit.edu/browse/pfodor/kalm/files

[2] Currently, ACE supports no past or future tenses, but they can be added.

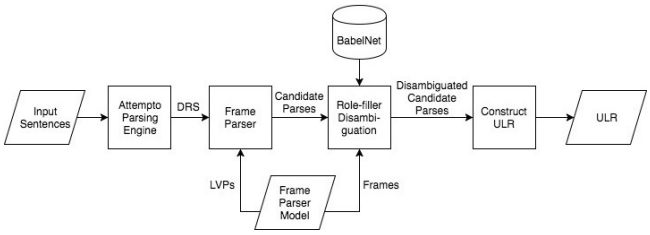

Fig. 1. The KALM pipeline for acquiring knowledge from CNL sentences

appear in that frame. Each grammatical pattern is captured by a *parsing rule* (a Prolog rule) that can be used to extract appropriate role-filler words based on the APE parses.

D. Knowledge Authoring in the KALM Framework

KALM is a high-accuracy knowledge authoring framework for translating CNL sentences into actionable logic. Its first version was limited to extraction of facts from *factual* sentences [11]. The result of parsing is translated into a logical form, called *unique logical representations* (*ULR*), which was used for reasoning. The pipeline of the KALM framework for factual sentences is shown in Figure 1. This process has several non-trivial parts.

Syntactic parsing. KALM uses APE to perform syntactic parsing of CNL sentences, yielding DRS logic facts as output.

Frame-based parsing. The DRS facts are fed to the frame-based parser, which generates a set of *candidate parses*. This includes the semantic frames and the lvps that the sentences might possibly belong to. For example, *Mary buys a car* has one candidate parse: `Frame(Commerce_Buy, Roles: Buyer =` *Mary*`, Goods =` *car*`)`, where *Mary* is the `Buyer` and *car* is the `Goods` of the `Commerce_Buy` frame. For each word in a sentence, the frame-based parser checks whether there is an lvp whose lexical unit matches the word and its part-of-speech. If so, the lvp is applied to the sentence and extracts the words that fill the various roles of the frame. This is not sufficient, however, as candidate parses may be wrong and thus yield wrong frames or extract wrong role-filler words.

Example 1. Consider the following sentences:

- *A laborer makes a bridge.*
- *Mary makes a cake.*
- *John makes a start-up.*

These three sentences have exactly the same syntactic structure, but obviously talk about very different things. Three different lvps, belonging to different frames, apply to each of these sentences:

```
lvp(make,n,Building, [
    pattern(Agent,verb->subject,required),
    pattern(Created_Entity,verb->object,
            required)]).
lvp(make,v,Cooking,[
    pattern(Cook,verb->subject,required),
    pattern(Food,verb->object,required)]).
lvp(make,v,Create_Organization,[
    pattern(Creator,verb->subject,required),
    pattern(Org,verb->object,required)]).
```

Thus, the sentence *A laborer makes a new bridge* has three candidate parses extracted due to these disparate lvps, and similarly for the other two sentences:

```
Frame(Building, Roles: Agent = laborer,
      Created_Entity = bridge)
Frame(Cooking, Roles: Cook=laborer, Food=bridge)
Frame(Create_Organization,
      Roles: Creator = laborer, Org = bridge)
```

For the first sentence, we would like to eliminate the last two parses; for the second, only the second lvp is right; and for the third sentence we want only the parse based on the third lvp. This is done via *role-filler disambiguation*.

Role-filler disambiguation. Given a role and a role-filler word, disambiguation is done by first finding all semantically-meaningful BabelNet paths between all the synsets of the filler-word and the synsets that represent the meanings of the role. The paths are scored for the strength of the connection they represent, and the highest-scored connection is chosen. The score takes into account the length of the path, the popularity of the intermediate nodes on the path, the types of the links (e.g., hypernym, holonym), and other factors. The role-filler synset that achieves the highest score is chosen as the *most probable* synset for the role-filler in question. Once each role-filler word is disambiguated, the entire *disambiguated candidate parse* is scored and the parses that fall below a threshold are removed.

Role-filler disambiguation is related to word-sense disambiguation but it does not try to disambiguate entire sentences. Instead, the goal is to disambiguate different senses of the extracted role-fillers and find the best sense for each.

Translation to unique logical representation (ULR). The disambiguated candidate parses are translated into ULR, which represents the true meaning of the original CNL sentence and is suitable for querying. ULR uses the predicates `frame` and `role` to represent instances of the frames and the roles. The predicates `synset` and `text` are used for the synset and the textual information. This is illustrated next.

Example 2. Consider the following information:

- *Mary buys a Camry for 15000 dollars.*
- *Mary pays 10000 dollars for a Jetta.*
- *Mary makes a purchase of a pen at a price of 2 dollars.*
- *Mary purchases a diamond with 30000 dollars.*

Although these sentences have very different syntactic structures, they all trigger the same `Commerce_Buy` frame and provide role-fillers for `Buyer`, `Goods`, and `Money`. For instance, the first sentence matches the following lvp:

```
lvp(buy, v, Commerce_Buy, [
 pattern(Buyer, verb->subject, required),
 pattern(Goods, verb->object, required),
 pattern(Recipient, verb->pp(for)->dep, optnl),
 pattern(Money, verb->pp(for)->dep, optnl),
 pattern(Seller, verb->pp(from)->dep, optnl)]).
```

Since the `Recipient` and `Money` roles share the same grammatical pattern, the first sentence above has two candidate parses. They differ in the `Money` and `Recipient` roles, which come from different `patterns` of the above lvp:

Frame(Commerce_Buy, Roles: Buyer = *Mary*, Goods = *Camry*, Money = *15000 dollars*)
Frame(Commerce_Buy, Roles: Buyer = *Mary*, Goods = *Camry*, Recipient = *15000 dollars*)

The second parse is ruled out by role-filler disambiguation since the role-filler *15000 dollars* does not match the Recipient role. After the disambiguation, the parse is translated into the following ULR:

```
frame(id_1, Commerce_buy).
role(id_1, Buyer, id_2).
role(id_1, Goods, id_3).
role(id_1, Money, id_4).
synset(id_2, bn:00046516n). % Person synset
text(id_2, Mary).
synset(id_3, bn:03606178n). % Camry synset
text(id_3, Camry).
synset(id_4, bn:00024507n). % Currency synset
text(id_4, '15000 dollars').
```

Translations of sentences 2-4 have similar ULR structure except that the frame- and role-facts will have different IDs, and synset and text-facts will use different BabelNet synsets and role-filler words.

III. The KALM Query Language

The KALM query language is a hybrid CNL based on ACE. A query can be either an *interrogative* or an *affirmative* sentence. Interrogative queries are sentences like

- *Does Mary buy a car?*
- *What does Mary buy?*

The first is a true/false query, which yields no output. The second query has an *output variable*, represented by *What*—a placeholder for entities to be shown in the result. Examples of affirmative queries are

- *Mary buys a $car.*
- *A $person buys a Toyota.*

The first query has an output variable, *$car*, which says that the output entities are expected to be of type car. The last query has an output variable *$person*, telling that the output entities are to be of type person. In general, output variables are represented in one of the following ways: as *wh-variables* or as *explicitly typed variables*.

Wh-variables include *who*, *where*, *when*, *which*, *whose*, *what*. Each *wh*-variable has a *type*, as explained next. For some *wh*-variables, the types are predefined: the type of *who* or *whose* is fixed to be the BabelNet synsets of *person* or *organization*. The type of *where* is set to BabelNet synsets for *place*. For *when*, the type is set to be synsets of *date and time*, and the type of *how much/how many* is the quantity of an entity. The types of *which* and *what* are determined by the semantic parser via a more complicated process, described in Section IV-A. Queries that use wh-variables include sentences like:

1) *Who buys a car?*
2) *Where does Mary live?*
3) *What is Mary's job?*
4) *Mary lives in which city?*
5) *Mary buys what?*

In sentences (1) and (2), the types of *Who* and *Where* are fixed, as described above. In sentence (3), *What* is semantically associated with the BabelNet synset representing a *job* entity; it is done by the semantic parser. In sentence (4), *which* is set to the BabelNet synset representing a *city* entity; again, this is determined semantically by the parser. In sentence (5), the type of *What* is Thing, which means that the answer can be an entity of any type.

Explicitly typed variables are represented by noun words prefixed with the *$*-sign. This is one instance of the *hybrid syntax* in KALM. In parsing, each such variable will be matched to a role in a frame. Queries containing explicitly typed variables are sentences like:

- *Mary buys a $car.*
- *A $person buys which car*?

Note that the first example is an affirmative sentence, while the second is an interrogative sentence. As we shall see, the types of the explicitly typed variables (*$car* and *$person* here) are determined via semantic parsing.

Aggregation. Basic aggregation is supported via *how many/much* variables as in

How many apples does Mary buy?

More complex aggregation can be supported via additional hybrid syntax. For instance,

How many products are sold per company, sorted?

This is a hybrid syntax because ACE does not support the idioms *per*, and *sorted* and not all sentences may be grammatically correct English.

Coordinated conjunction is expressed in KALM by joining independent clauses with the word *and* or relative pronouns (e.g., *that*, *which*). Examples of coordinated conjunctions are sentences like:

- *Who graduates from UC Berkeley and founds Apple Inc*?
- *Which person who graduates from UC Berkeley founds Apple Inc*?

For the above cases, the entities that the *who*/*which*-variables hold must satisfy the following conditions: (1) graduating from UC Berkeley, and (2) founding of Apple Inc. More complex coordination can be achieved via additional hybrid syntax.

IV. Query Processing

This section describes query processing and answer filtering.

A. Query Parsing

Query parsing has these main parts: syntactic parsing, DRS adaptation, and frame-based semantic parsing. *Syntactic parsing* simply uses APE to create DRS representations for CNL queries. For instance, the query *Who buys what?* is represented in DRS as

```
query(A,who)-1/1
query(B,what)-1/3
predicate(C,buy,A,B)-1/2
```

The variables *who* and *what* appear in the query-predicate.

Phase two, *DRS adaptation*, serves two purposes: rewriting DRS to prepare for frame-based semantic parsing and lexical typing of output variables in a query. DRS rewriting is needed because queries have different DRS structure than sentences

specifying factual knowledge, and parsing queries directly requires additional lvps and parsing rules. DRS rewriting into affirmative sentences obviates the need for these additions.

Typing of *what* and *which* variables is done via a syntactic analysis of the associated words. For instance, in *Mary buys which car?*, the variable *which* is typed as a *car* entity, by a typing word appearing in the sentence (this kind of words are called *lexical answer types* [18]). Similarly, in *What is Mary's job?*, the type of *What* will be identified to be the noun *job*. Technical details are omitted due to space limitation. In other cases, like in *Who buys a car?*, the variable *Who* is fixed to be a person or organization entity, as explained previously. Similarly, for explicitly typed variables, the type is given explicitly.

DRS adaptation is done by Prolog rules that have this structure:

```
change_query_DRS(OldDRS, NewDRS):-
  find_output_variable(OldDRS,Var),
  find_variable_type(OldDRS,Var, VarType),
  to_affirmative_drs(OldDRS,Var,VarType,NewDRS).
```

The third phase of query parsing performs frame-based parsing of affirmative sentences, as described in Section II-D, which yields *candidate parses for queries*. The difference in case of queries is that the parses may contain output variables. For example, the query *Mary buys a $car* has one candidate parse: `Frame(Commerce_Buy, Roles: Buyer =` *Mary*`, Goods =` *$car*`)` in which *Mary* is a named entity and *$car* is a typed output variable.

B. Role-filler Disambiguation for Queries

As shown in Section II-D, frame-based parsing of factual sentences may yield multiple candidate parses, which requires disambiguation. This problem occurs for queries also. Consider these queries with exactly the same syntactic structure:

- *Who makes a cake?*
- *Who makes a start-up?*
- *Who makes a bridge?*

These queries can be parsed using any of the three lvps shown in Example 1 of Section II-D, but only one of the parses for each of these sentences is semantically correct. For instance, the query *Who makes a cake?* has these candidate parses:

`Frame(Cooking, Roles: Cook=`*who*`, Food=`*cake*`)`
`Frame(Building, Roles: Agent=`*who*`,`
`  Created_Entity=`*cake*`)`
`Frame(Create_Organization, Roles:`
`  Creator=`*who*`, Org=`*cake*`)`

In this example, the second and the third candidate parses are wrong and should be eliminated at the role-filler disambiguation phase. This is similar to disambiguation for factual sentences, as described in Section II-D. The process disambiguates each role-filler word, scores the entire candidate parse, and removes the parses that score low. The main difference is that instead of words representing entities, queries have output variables. For variables that are bound to types by the DRS adaptor, the lexical type-words are used as role-fillers in the scoring process. For untyped variables (e.g., the untyped *what* in *What does Mary buy?*), a predefined score is used. We call the synsets that represent the type of the output variables *answer type synsets*.

C. Translating Queries into ULRQ

Disambiguated query parses are translated into a logical form called *unique logical representation for queries* (*ULRQ*), which is then used to query the acquired knowledge base. ULRQ is similar to ULR for factual sentences (described in Section II-D) except that ULRQ is for queries and therefore it uses logical variables to represent instances of frames and role-fillers. We should note that KALM queries can also be translated into SPARQL [5] or RIF [3], [4], but ULRQ is directly executable in our implementation platform (XSB [19]). One can query the sentences in Example 2 as follows:

- *Who is a buyer of a $car?*
- *Who makes a purchase of which car?*
- *A $person purchases a $car.*
- *Who buys which car?*

Although these questions have different syntactic forms, the frame-based parser generates the following unique candidate parse for *all of them*, since they all have the same meaning: `Frame(Commerce_Buy, Roles: Buyer =` *Who*`, Goods =` *$car*`)`. Note that no other approach does this kind of systematic query standardization, which is absolutely essential for getting correct answers. This parse is translated into this ULRQ (in Prolog syntax):

```
?-frame(FrameV,'Commerce_Buy'),
  role(FrameV,'Buyer',BuyV),
  synset(BuyV,BuyerRoleFillerOutV),
  role(FrameV,'Goods',GoodV),
  synset(GoodV,GoodsRoleFillerOutV),
  check_type(BuyerRoleFillerOutV,bn:00046516n),
  check_type(GoodsRoleFillerOutV,bn:00007309n).
```

where *Who* is associated with the *person*-synset (BabelNet's `bn:00046516n`) and *$car* is associated with the *car*-synset (`bn:00007309n`).

Lines 1-3 in the query find all the known instances of the `Commerce_Buy` frame in the knowledge base along with the `Buyer` and `Goods` role-fillers and their synsets. This returns *Mary* as the `Buyer`, but for the `Goods` role we get *Camry*, *Jetta*, *pen* and *watch* as the role-fillers. These are *candidate answers* to the query and their synsets are *candidate answer synsets*. Not all of them are real answers, however, since the query asks about cars and persons, while not all the purchased goods are cars. The extraneous answers are eliminated by type filtering on lines 4 and 5. Details of the algorithms for type checking are discussed in the next subsection.

Next, consider a more complicated example of coordinated sentences in a query: *Who graduates from UC Berkeley and founds Apple Inc?* The parser generates two candidate parses whose conjunction represents the meaning of the entire query: `Frame(Education, Roles: Student =` *Who*`, Institution =` *UC Berkeley*`)` and `Frame(Create_Organization, Roles: Creator =` *Who*`, Org =` *Apple Inc*`)`. Here, the output variable *Who* (represented as `WhoVar` in ULRQ) fills the roles of `Creator` and `Student`, and it denotes the same entity:

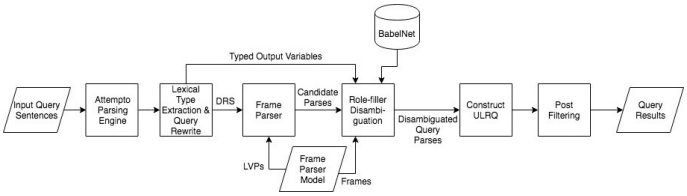

Fig. 2. The KALM pipeline for query processing and answer retrieval and filtering

```
?-frame(FrameV1,'Education'),
  role(FrameV1,'Student',WhoVar),
  synset(WhoVar,WhoVarSynset),
  role(FrameV1,'Institution',InstitutionV),
  synset(InstitutionV,bn:02547986n),
  frame(FrameV2,'Create_Organization'),
  role(FrameV2,'Creator',WhoVar),
  synset(WhoVar,WhoVarSynset),
  role(FrameV2,'Org',OrgV),
  synset(OrgV,bn:03739345n),
  check_type(WhoVarSynset,bn:00046516n).
```

D. Explaining Query Parses

As discussed in Section IV-B, role-filler disambiguation assigns the most appropriate BabelNet synset to each role-filler word by finding the highest-scored semantic path that connects the candidate synset for the role-filler word and a synset for the corresponding role in the selected *FrameOnt* frame. These paths can be used to explain the chosen meanings for queries.

Interestingly, this same mechanism can be used to explain why a *wrong* meaning was assigned to a query. A wrong meaning may get chosen because BabelNet includes fair amount of noise due to the fact that this knowledge graph was created automatically by merging information from various sources. The noise is typically manifested via wrong synset assignments to words, incorrect semantic links, and missing links. Explanations to disambiguation errors help domain experts pinpoint the exact instances of BabelNet noise that contributed to these errors and KALM provides the means to neutralize that noise and correct the errors. Due to space limitation, we omit examples or error correction and the reader is referred to [11].

E. Type Filtering of Query Results

Type filtering checks whether the type of each candidate answer synset matches the corresponding answer-type synset by measuring their affinity in BabelNet's network. For each candidate answer and answer-type synset, the type filtering process does breadth-first search to find all semantic paths from the candidate answer synset to the answer-type synset. A heuristic scoring function assigns a score to each path, prunes the unpromising paths, and selects the path with the highest score. The scoring function is chosen to encourage the paths going through popular nodes and edges with higher weight, and to penalize longer paths. Different types of edges in a path also have different *relevance factors*. For instance, hypernym edges have the highest relevance factor, glossary-based links and derivational relationships have lower relevance, and hyponyms even lower. Formally, let n_1 be a candidate answer synset node, n_l be an answer-type synset node and $L = \{n_1, e_{12}, n_2, e_{23}, \cdots, e_{l-1,l}, n_l\}$ be a semantic path from n_1 to n_l, where the n_i's are BabelNet synset nodes and $e_{i,i+1}$ is an edge between n_i and n_{i+1}. The semantic score of the path is

$$score = \frac{\sum_{i=1}^{n-1} \sqrt{f_n(n_i)} \times f_w(e_{i,i+1})}{5^{\sum_{i=1}^{n-1} f_p(e_{i,i+1})}} \quad (1)$$

where $f_n(n_i)$ is n_i's popularity score, $f_w(e_{i,i+1})$ is the sum of $e_{i,i+1}$'s edge weight and its relevance factor, and $f_p(e_{i,i+1})$ is a penalty for following the edge $e_{i,i+1}$, defined based on the edge type. The exponent base of 5 in the denominator implies serious penalty for longer paths. The candidate answer synsets with higher scores are retained in the output and the synsets with scores that fall below a threshold are removed.

Unfortunately, scoring paths using a brute-force search takes hours because BabelNet is very large and the number of required BabelNet queries is in millions, while even a *local* query takes at least 10 ms. To deal with this issue, we developed a heuristic search algorithm, which reduces the computation time from hours to seconds. The algorithm is based on the optimizations described below.

Parallel computation. The first obvious observation is that the brute-force algorithm is easily parallelizable, since finding the semantic paths connecting candidate answers and answer-type synsets can be done independently. A thread pool is created where each thread finds paths from a specific candidate answer synset (*cand_ans_syn*$_{ki}$) to an answer type synset (*ans_type_syn*$_k$). If one such path is found, the highest known score (*max_score*$_k$) for the found paths is shared with all the parallel threads that have the same answer-type synset; it is used to compute a cut-off value (*cutoff_val*$_k$) for pruning the computation of any low-scoring path that is in progress. Here, variable *cutoff_val*$_k$ is computed as $\alpha \cdot$*max_score*$_k$ ($0 < \alpha \leq 1$), where α is a learned parameter. With more CPU cores, more threads could be created, which would speed up the computation even further.

Caching BabelNet queries. As mentioned, BabelNet queries are relatively expensive, even when no network is involved, and collectively they take most of the computing time. Profiling shows, however, that about 70% of such queries are repeated more than once. To improve the search for paths between pairs of synsets, KALM caches the results of BabelNet queries internally, which yields a speedup of 3-5 times.

Priority-based BabelNet path search. Given the complexity of BabelNet, any pair of synsets can have many connecting paths and even more paths may wonder astray without connecting the target nodes. Although we prune away paths that score low, naive breadth-first search can get bogged down exploring wrong parts of the graph. To avoid this problem, we use priority-based search, which downgrades and eventually prunes unpromising paths.

The pseudo-code of the parallel priority-based path search is shown in Algorithm 1. The setup phase initializes the

Algorithm 1: priority_based_path_search_thread

Input: *cand_ans_syn*$_{ki}$, *ans_type_syn*$_k$, BabelNet semantic network
Output: *score*$_{ki}$

```
1  global max_score_k, cutoff_val_k, protected_path_len,
     path_len_limit;
2  initialize queue_ki;    // priority queue for synset nodes
3  initialize ht_ki<synset_id, score>; // hash table for scores
4  initialize score_ki := 0; // max score for input synset pair
5  queue_ki.add(cand_ans_syn_ki);
6  while not empty(queue_ki) do
7      cur_node := queue_ki.top();
8      cur_len := path_len(cand_ans_syn_ki, cur_node);
9      next_len := cur_len + 1;
10     if cur_node.score > max_score ∨
11       cur_len ≤ protected_path_len then
12         for nbr_node ∈ get_neighbors(cur_node) do
13             nbr_node.score:=score(cand_ans_syn_ki,nbr_node);
14             if nbr_node.sid == ans_type_syn_k then
15                 if nbr_node.score > max_score then
16                     max_score_k := nbr_node.score;
17                     cutoff_val_k := α·max_score_k;
18                 if nbr_node.score > score_ki then
19                     score_ki := nbr_node.score;
20             else
21                 if (nbr_node.score > cutoff_val_ki ∧
22                       next_len < path_len_limit)
23                   ∨ (next_len ≤ protected_path_len) then
24                     if not ht_i.hasKey(nbr_node.sid) then
25                         queue_ki.add(nbr_node);
26                         ht_ki[nbr_node.sid] := nbr_node.score;
27                     else
28                         if ht_i[nbr_node.sid]<nbr_node.score
29                         then
30                             queue_ki.add(nbr_node);
31                             ht_ki[nbr_node.sid]=nbr_node.score;
```

global variables *max_score*$_k$ and *cutoff_val*$_k$, which are shared with all threads that have the same answer-type synset *ans_type_syn*$_k$. A priority queue, q_i, for each candidate answer synset *cand_ans_syn*$_{ki}$ is also initialized.

The algorithm starts a thread per each candidate answer synset *cand_ans_syn*$_{ki}$, trying to reach the answer-type synset *ans_type_syn*$_k$ via a path with as high a score as possible. During the breadth-first search, each time a synset is encountered, we create a new *synset node* that records the synset ID, total number of semantic links, semantic score, and the parent information. In each round of the computation, we pick the top element (cur_node) from the priority queue and compare its semantic score with *cutoff_val*$_k$. If this score is below *cutoff_val*$_k$, we discard the current node and choose the next element from the priority queue. Otherwise, we inspect each neighbor nbr_node of *cand_ans_syn*$_{ki}$ and compute the score of the path from *cand_ans_syn*$_{ki}$ to nbr_node. If the neighbor is the requisite answer-type synset, a path from *cand_ans_syn*$_{ki}$ to *ans_type_syn*$_k$ has been found and we update *max_score*$_k$ if the score of the newly found path exceeds *max_score*$_k$; we also update *cutoff_val*$_k$ accordingly. If the node's neighbor is not the requisite answer-type synset, we push the node onto the priority queue, if it satisfies certain conditions. First, the length of the path from *cand_ans_syn*$_{ki}$ to nbr_node should not exceed a certain limit (e.g., 7). Second, the semantic score for the path from *cand_ans_syn*$_{ki}$ to nbr_node must be greater than the global *cutoff_val*$_k$. Third, if nbr_node was visited before, it means we have seen a different path from *cand_ans_syn*$_{ki}$ to nbr_node and the score for the current path must be greater than that for the previous path. Finally, paths that are too short (e.g., shorter than 3) are never pruned on a theory that they may turn out to be "late bloomers." The process continues until the priority queue is empty.

Note that, similarly to query parsing, type filtering can be explained to the user.

V. Experiments

Data. At present, KALM contains 50 logical frames with 213 logical valence patterns. It was shown in [11] that this achieves an accuracy of 95.6% for factual sentences. This section presents an evaluation of the question answering aspect of KALM. We used 179 queries[3] to check whether the system returns the expected frames, disambiguates role-filler words correctly, and identifies the types of the *output variables*. Note that since these queries are against the databases authored using KALM, correct queries are *guaranteed* to return correct results. Since our approach is based on CNL, public standardized data sets are not available and so our queries are based on factual CNL sentences with appropriate modifications according to English grammar.

Results. The results are summarized below. In a nutshell, 170 out of 179 sentences are parsed correctly, 5 partially correctly, and 4 incorrectly. Note that KALM's 95% accuracy compares favorably to about 62% accuracy of query systems for open domain, such as [20].

Sentences	**Explanation**
170 (94.97%)	all frames, roles & output variables are identified correctly; all role-filler words & variable types are disambiguated correctly
5 (2.79%)	all frames, roles and output variables are identified correctly, but some have disambiguation mistakes
4 (2.23%)	some frames, roles or variables are misidentified

An example of a misidentified frame is the sentence *Who releases an article on waltz?* which should have been placed in the `Publishing` frame, but finds itself in the `Releasing_from_custody` frame instead. The reason is that BabelNet does not think "waltz" is a likely topic for an article. An example where the frame is identified correctly, but not all role-fillers are disambiguated correctly is the sentence *Which country does Shinzo Abe visit?* Here, "country" was misinterpreted as "countryside."

VI. Related work

In our previous work [11] on high precision knowledge authoring, we compared KALM with the state-of-the-art infor-

[3] https://datahub.csail.mit.edu/browse/pfodor/kalm/files

mation extraction systems such as SEMAFOR [21], SLING [22], and Stanford CoreNLP [23]. We showed that other systems lack sufficient accuracy even for simple and common sentences. These systems are not designed for query processing and so are ill-suited for comparison with the present work.

Related work includes natural language interfaces for database and linked data query systems, such as ATHENA [24], Quadri [25], NaLIR [26], and PRECISE [27], PowerAqua [28], and Pythia [29], which are tuned to specific ontologies. However, these systems do not support knowledge authoring and they query *existing* databases via ontologies constructed for each case separately. In contrast, KALM reuses general linguistic resources and knowledge bases in an incremental fashion, and is suitable both for knowledge authoring and querying—with superior accuracy.

Another related field is textual entailment [30]. These works have achieved impressive results, but the corresponding systems are not incremental and their accuracy tends to be below 80%—too low for reasoning about knowledge.

VII. Conclusion

Although considerable work has been done on question answering, we are unaware of a query system that can respond to questions about knowledge that was automatically acquired from text and has very high accuracy. KALM, which combines CNL-based knowledge authoring and question answering, is unique in that respect. In this paper, we described the question answering aspect of KALM, including a CNL-based query language, semantic parsing of queries, disambiguation and answer filtering. We show that this approach is within 5% of perfect accuracy. This superior accuracy was achieved through a unique combination of logic, linguistic knowledge bases FrameNet and BabelNet, and sophisticated parsing, disambiguation, and answer filtering algorithms. For future work, we plan to extend KALM to acquire rules with near perfect accuracy, and support common sense and other complex types of reasoning.

Acknowledgements. We would like to thank Niranjan Balasubramanian and H. Andrew Schwartz for the helpful discussions on knowledge authoring and question answering. This work was partially supported by NSF grant 1814457.

References

[1] O. Lassila and R. S. (editors), "Resource description framework (RDF) model and syntax specification," W3C, Tech. Rep., February 1999, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[2] OWL Working Group, "Web ontology language (OWL)," WWW Consortium, Tech. Rep., December 2012, https://www.w3.org/OWL/.

[3] H. Boley and M. Kifer, "RIF Basic logic dialect," July 2013, w3C Recommendation. [Online]. Available: http://www.w3.org/TR/rif-bld/

[4] ——, "RIF Framework for logic dialects," July 2013, w3C Recommendation. [Online]. Available: http://www.w3.org/TR/rif-fld/

[5] E. PrudHommeaux, A. Seaborne *et al.*, "SPARQL query language for RDF," *W3C recommendation*, vol. 15, 2008.

[6] Mausam, M. Schmitz, S. Soderland, R. Bart, and O. Etzioni, "Open language learning for information extraction," in *Joint Conf. on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL*, 2012, pp. 523–534.

[7] G. Angeli, M. J. J. Premkumar, and C. D. Manning, "Leveraging linguistic structure for open domain information extraction," in *53rd Annual Meeting of the Association for Computational Linguistics*, Beijing, China, 2015, pp. 344–354.

[8] T. Kuhn, "A survey and classification of controlled natural languages," *Comp. Linguistics*, vol. 40, no. 1, pp. 121–170, 2014.

[9] N. E. Fuchs, K. Kaljurand, and T. Kuhn, "Attempto controlled english for knowledge representation," in *Reasoning Web, 4th Intl. Summer School, Sept. 7-11*, Venice, Italy, 2008, pp. 104–124.

[10] R. Schwitter, "English as a formal specification language," in *13th Intl. Workshop on Database and Expert Systems Appl.*, 2002, pp. 228–232.

[11] T. Gao, P. Fodor, and M. Kifer, "Knowledge authoring for rule-based reasoning," in *17th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE), Malta*, October 2018.

[12] N. E. Fuchs, K. Kaljurand, and T. Kuhn, "Discourse Representation Structures for ACE 6.6," Department of Informatics, University of Zurich, Switzerland, Tech. Rep. 2010.0010, 2010.

[13] R. Navigli and S. P. Ponzetto, "BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network," *Artificial Intelligence*, vol. 193, pp. 217–250, 2012.

[14] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[15] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives, "DBpedia: A nucleus for a web of open data," in *6th Intl. Semantic Web Conf.*, 2007, pp. 722–735.

[16] D. Vrandečić and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Comm. of the ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[17] C. R. Johnson, C. J. Fillmore, M. R. Petruck, C. F. Baker, M. J. Ellsworth, J. Ruppenhofer, and E. J. Wood, "FrameNet: Theory and Practice," 2002.

[18] A. Lally, J. M. Prager, M. C. McCord, B. K. Boguraev, S. Patwardhan, J. Fan, P. Fodor, and J. Chu-Carroll, "Question analysis: How watson reads a clue," *IBM Journal of Research and Development*, vol. 56, no. 3.4, pp. 2–1, 2012.

[19] T. Swift and D. Warren, "XSB: Extending the power of prolog using tabling," *Theory and Practice of Logic Programming*, 2011.

[20] C. Unger, L. Bühmann, J. Lehmann, A. N. Ngomo, D. Gerber, and P. Cimiano, "Template-based question answering over RDF data," in *21st World Wide Web Conference, WWW*, 2012, pp. 639–648.

[21] D. Das, D. Chen, A. F. T. Martins, N. Schneider, and N. A. Smith, "Frame-semantic parsing," *Comp. Linguistics*, vol. 40, no. 1, pp. 9–56, 2014.

[22] M. Ringgaard, R. Gupta, and F. C. N. Pereira, "SLING: A framework for frame semantic parsing," *CoRR*, vol. 1710.07032, pp. 1–9, 2017. [Online]. Available: http://arxiv.org/abs/1710.07032

[23] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford CoreNLP natural language processing toolkit," in *52nd Annual Meeting of the Assoc. for Computational Linguistics, ACL, System Demonstrations*, 2014, pp. 55–60.

[24] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan, "ATHENA: an ontology-driven system for natural language querying over relational data stores," *PVLDB*, vol. 9, no. 12, pp. 1209–1220, 2016.

[25] K. Richardson, D. G. Bobrow, C. Condoravdi, R. J. Waldinger, and A. Das, "English access to structured data," in *5th IEEE International Conference on Semantic Computing (ICSC)*, 2011, pp. 13–20.

[26] F. Li and H. V. Jagadish, "Constructing an interactive natural language interface for relational databases," *PVLDB*, vol. 8, no. 1, pp. 73–84, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol8/p73-li.pdf

[27] A. Popescu, O. Etzioni, and H. A. Kautz, "Towards a theory of natural language interfaces to databases," in *8th International Conference on Intelligent User Interfaces, IUI, Miami, FL*, 2003, pp. 149–157.

[28] V. López, M. Fernández, E. Motta, and N. Stieler, "Poweraqua: Supporting users in querying and exploring the semantic web," *Semantic Web*, vol. 3, no. 3, pp. 249–265, 2012.

[29] C. Unger and P. Cimiano, "Pythia: Compositional meaning construction for ontology-based question answering on the semantic web," in *16th International Conference on Applications of Natural Language to Information Systems, NLDB, Alicante, Spain*, 2011, pp. 153–160.

[30] I. Androutsopoulos and P. Malakasiotis, "A survey of paraphrasing and textual entailment methods," *CoRR*, vol. abs/0912.3747, 2009.