Handbook of Knowledge Representation
Edited by F. van Harmelen, V. Lifschitz and B. Porter

DOI: 10.1016/S1574-6526(07)03020-9

Chapter 20

Knowledge Representation and Question Answering

Marcello Balduccini, Chitta Baral, Yuliya Lierler

20.1 Introduction

Consider an intelligence analyst who has a large body of documents of various kinds. He would like answers to some of his questions based on the information in these documents, general knowledge available in compilations such as fact books, and commonsense. A search engine or a typical information retrieval (IR) system like Google does not go far enough as it takes keywords and only gives a ranked list of documents which may contain those keywords. Often this list is very long and the analyst still has to read the documents in the list. Other reasons behind the unsuitability of an IR system (for an analyst) are that the nuances of a question in a natural language cannot be adequately expressed through keywords, most IR systems ignore synonyms, and *most IR systems cannot reason*. What the intelligence analyst would like is a system that can take the documents and the analyst's question as input, that can access the data in fact books, and that can do commonsense reasoning based on them to provide answers to questions. Such a system is referred to as a question answering system or a QA system. Systems of this type are useful in many domains besides intelligence analysis. Examples include a Biologist who needs answers to his questions, say about a particular set of genes and what is known about their functions and interactions, based on the published literature; a lawyer looking for answers from a body of past law cases; and a patent attorney looking for answers from a patent database.

A precursor to question answering is database querying where one queries a database using a database query language. Question Answering takes this to a whole other dimension where the system has increasing body of documents (in natural languages, possibly including multimedia objects and possibly situated in the web and described in a web language) and it is asked a query in natural language. It is expected to give an answer to the question, not only using the documents, but also using appropriate commonsense knowledge. Moreover, the system needs to be able to accommodate new additions to the body of documents. The interaction with a question answering system

can also go beyond a single query to a back and forth exchange where the system may ask questions back to the user so as to better understand and answer the user's original question. Moreover, many questions that can be asked in English can be proven to be inexpressible in most existing database query languages.

The response expected from a QA system could also be more general than the answers expected from standard database systems. Besides yes/no answers and factual answers, one may expect a QA system to give co-operative answers, give relaxed answers based on user modeling and come back with clarifying questions leading to a dialogue. An example of co-operative answering [31] is that when one asks the question "Does John teach AI at ASU in Fall'06", the answer "the course is not offered at ASU in Fall'06", if appropriate, is a co-operative answer as opposed to the answer "no". Similarly, an example of relaxed answering [30] is that when one asks for a Southwest connection from Phoenix to Washington DC National airport, the system realizing that Baltimore is close to DC, and Southwest does not fly to DC, offers the flight schedules of Southwest from Phoenix to Baltimore.

QA has a long history and [53] contains an overview of that as well as various papers on the topic. Its history ranges from early attempts on natural language queries for databases [39], deductive question answering [40], story understanding [19], web based QA systems [4], to recent QA tracks in TREC [72], ARDA supported QA projects and Project Halo [29]. QA involves many aspects of Artificial Intelligence ranging from natural language processing, knowledge representation and reasoning, information integration and machine learning. Recent progress and successes in all of these areas and easy availability of software modules and resources in each of these areas now make it possible to build better QA systems. Some of the modules and resources that can be used in building a QA system include natural language parsers, WordNet [54, 26], document classifiers, text extraction systems, IR systems, digital fact books, and reasoning and model enumeration systems. However, most QA systems built to date are not strong in knowledge representation and reasoning, although there has been some recent progress in that direction. In this chapter we will discuss the role of knowledge representation and reasoning in developing a QA system, discuss some of the issues and describe some of the current attempts in this direction.

20.1.1 Role of Knowledge Representation and Reasoning in QA

To understand the role of knowledge representation and reasoning in a QA system let us consider several pairs of texts and questions. We assume that the text has been identified by a component of the QA system from among the documents given to it, as relevant to the given query.

1. *Text*: John and Mike took a plane from Paris to Baghdad. On the way, the plane stopped in Rome, where John was arrested.

   Questions: Where is Mike at the end of this trip? Where is John at the end of this trip? Where is the plane at the end of this trip? Where would John be if he was not arrested?

   Analysis: The commonsense answers to the above questions are Baghdad, Rome, Baghdad and Baghdad respectively. To answer the first and the third

question the QA system has to reason about the effect of the action of taking a plane from Paris to Baghdad. It has to reason that at the end of the action the plane and its occupants will be in Baghdad. It has to reason that the action of John getting arrested changes his status as an occupant of the plane. To reason about John's status if he was not arrested, the QA system has to do counterfactual reasoning.

2. *Text*: John, who always carries his laptop with him, took a flight from Boston to Paris on the morning of Dec 11th.

   Questions: In which city is John's laptop on the evening of Dec 10th? In which city is John's laptop on the evening of Dec 12th?

   Analysis: The commonsense answers to the above questions are Boston and Paris respectively. Here, as in the previous case, one can reason about the effect of John taking a flight from Boston to Paris, and conclude that at the end of the flight, John will be in Paris. However, to reason about the location of John's laptop one has to reason about the causal connection between John's location and his laptop's location. Finally, the QA system needs to have an idea about the normal time it takes for a flight from Boston to Paris, and the time difference between them.

3. *Text*: John took the plane from Paris to Baghdad. He planned to meet his friend Mike, who was waiting for him there.

   Question: Did John meet Mike?

   Analysis: To answer the above question, the QA systems needs to reason about agent's intentions. From commonsense theory of intentions [18, 22, 74], agents normally execute their intentions. Using that one can conclude that indeed John met Mike.

4. *Text*: John, who travels abroad often, is at home in Boston and receives a call that he must immediately go to Paris.

   Questions: Can he just get on a plane and fly to Paris? What does he need to do to be in Paris?

   Analysis: The commonsense answer to the first question is 'no'. In this case the QA system reasons about the precondition necessary to perform the action of flying and realizes that for one to fly one needs a ticket first. Thus John cannot just get on a plane and fly. To answer the second question, one needs to construct a plan. In this case, a possible plan is to buy a ticket, get to the airport and then to get on the plane.

5. *Text*: John is in Boston on Dec 1. He has no passport.

   Question: Can he go to Paris on Dec 4?

   Analysis: With the general knowledge that it takes more than 3 days to get a passport the commonsense answer to the above is 'no'.

6. *Text*: On Dec 10th John is at home in Boston. He made a plan to get to Paris by Dec 11th. He then bought a ticket. But on his way to the airport he got stuck in the traffic. He did not make it to the flight.

   Query: Would John be in Paris on Dec 11th, if he had not gotten stuck in the traffic?

   Analysis: This is a counterfactual query whose answer would be "yes". The reasoning behind it would be that if John had not been stuck in the traffic, then he would have made the flight to Paris and would have been in Paris on Dec 11th.

The above examples show the need for commonsense knowledge and domain knowledge; and the role of commonsense reasoning, predictive reasoning, counterfactual reasoning, planning and reasoning about intentions in question answering. All these are aspects of knowledge representation and reasoning. The examples are not arbitrarily contrived examples, but rather are representative examples from some of the application domains of QA systems. For example, an intelligence analyst tracking a particular person's movement would have text like the above. The analyst would often need to find answers for what if, counterfactual and intention related questions. Thus, knowledge representation and reasoning ability are very important for QA systems. In the next section we briefly describe attempts to build such QA systems and their architecture.

20.1.2 Architectural Overview of QA Systems Using Knowledge Representation and Reasoning

We start with a high level description of approaches that are used in the few QA systems [1, 57, 71, 62] or QA-like systems that incorporate knowledge representation and reasoning.

1. Logic Form based approach:
   In this approach an information retrieval system is used to select the relevant documents and relevant texts from those documents. Then the relevant text is converted to a logical theory. The logical theory is then added to domain knowledge and commonsense knowledge resulting in a Knowledge Base KB. (Domain knowledge and common-sense knowledge will be together referred to as "background knowledge" and sometimes as "background knowledge base".) The question is converted to a logic form and is posed against KB and a theorem prover is then used. This approach is used in the QA systems [1, 20] from *Language Computer/LCC*.[1]

2. Information extraction based approach:
   Here also, first an information retrieval system is used to select the relevant documents and relevant texts from those documents. Then with a goal to extract relevant facts from these text, a classifier is used to determine the correct script and the correct information extractor for the text. The extracted relevant facts are added to domain knowledge and commonsense knowledge resulting in

[1] http://www.languagecomputer.com.

the Knowledge Base KB. The question is translated to the logical language of KB and is then posed against it. An approach close to this is used in the story understanding system reported in [62].

3. Using logic forms in information extraction:
   A mixed approach of the above two involves processing the logic forms to obtain the relevant facts from them and then proceed as in (2) above.

We now describe the above approaches in greater detail. We start by examining various techniques to translate English to logical theories. Next, we describe COGEX and DD, two systems that perform inference starting from the logic form of English sentences. Section 20.5 presents an approach where the output of a semantic parser is used directly in obtaining the relevant facts, and background knowledge is employed to reduce semantic ambiguity. In Section 20.6, we describe Nutcracker, a system for recognizing textual entailment based on first-order representation of sentences and first-order inference tools. Section 20.7 examines an approach based on the use of Event Calculus for the semantic representation of the text. Finally, in Section 20.8 we draw conclusions.

20.2 From English to Logical Theories

An ambitious and bold approach of doing reasoning in a question answering system is to convert English (or any other natural language for that matter) text to a logical representation and then use a reasoning system to reason with the resulting logical theory. Here, we discuss some of the attempts [1, 20] in this direction.

The most popular approach for the translation from English to a logical representation is based on the identification of the *syntactic structure* of the sentence, usually represented as a tree (the "parse tree") that systematically combines the phrases in which the English text can be divided and whose leaves are associated with the lexical items. As an example, the parse tree of the sentence "John takes a plane" is shown in Fig. 20.1. Once the syntactic structure is found, it is used to derive a logical representation of the discourse.

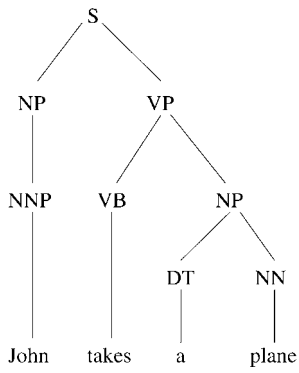

Figure 20.1: Parse tree of "*John takes a plane*".

The derivation of the logical representation typically consists of:

- Assigning a logic encoding to the lexical items of the text.
- Describing how logical representations of sub-parts of the discourse are to be combined in the representation of larger parts of it.

Consider the parse tree in Fig. 20.1 (for the sake of simplicity, let us ignore the determiner "a"). We can begin by stating that lexical items "John" and "plane" are represented by constants *john* and *plane*. Next, we need to specify how the verb phrase is encoded from its sub-parts. A possible approach is to use an atom $p(x, y)$, where p is the verb and y is the constant representing the syntactic direct object of the verb phrase. Thus, we obtain an atom $take(x, plane)$, where x is an unbound variable. Finally, we can decide to encode the sentence by replacing the unbound variable in the atom for the verb phrase with the constant denoting the syntactic subject of the sentence. Hence, we get to $take(john, plane)$.

Describing formally how the logical representation of the text is obtained is in general a nontrivial task that requires a suitable way of specifying how substitutions are to be carried out in the expressions.

Starting with theoretical attempts in [59] to a system implementation in [7], attempts have been made to use *lambda calculus* to tackle this problem. In fact, lambda calculus provides a simple and elegant way to mark explicitly where the logical representation of smaller parts of the discourse is to be inserted in the representation of the more complex parts. Here we describe the approach from [14].

Lambda calculus can be seen as a notational extension of first-order logic containing a new *binding operator* λ. Occurrences of variables bound by λ intuitively specify where each substitution has to occur. For example, an expression

$$\lambda x.plane(x)$$

says that, once x is bound to a value, that value will be used as the argument of relation *plane*. The application of a lambda expression is denoted by symbol @. Hence, the expression

$$\lambda x.plane(x) \ @ \ boeing767$$

is equivalent to $plane(boeing767)$. Notice that, in natural language, nouns such as plane are preceded by "a", "the", etc. In the lambda calculus based encoding, *the representation of nouns is connected to that of the rest of the sentence by the encoding of the article*.

In order to provide the connection mechanism, the lambda expressions for articles are more complex than the ones shown above. Let us consider, for example, the encoding of "a" from [14]. There, "a" is intuitively viewed as describing a situation in which an element of a class has a particular property. For example, "a woman walks" says that an element of class "woman" "walks". Hence, the representation of "a" is parameterized by the class, w, and the property, z, of the object, y:

$$\lambda w.\lambda z.\exists y.(w \ @ \ y \wedge z \ @ \ y).$$

In the expression, w is a placeholder for the lambda expression describing the class that the object belongs to. Similarly, z is a placeholder for the lambda expression denoting

the property of the object. Notice the implicit assumption that *the lambda expressions substituted to w and z are of the form $\lambda x.f(x)$*—that is, they lack the "$@\ p$" part. This assumption is critical for the proper merging of the various components of a sentence: when w, in $w\ @\ y$ above, is replaced with the actual property of the object, say $\lambda x.plane(x)$, we obtain $\lambda x.plane(x)\ @\ y$. Because of the use of parentheses, it is *only at this point* that the $@\ y$ part of the expression above can be used to perform a substitution. Hence, $\lambda x.plane(x)\ @\ y$ is simplified into $plane(y)$, as one would expect.

To see how the mechanism works on the complete representation of "a", let us look at how the representation of the phrase "a plane" is obtained by combining the encoding of "a" with the one of "plane" (which provides the class information for "a"):

$$\lambda w.\lambda z.\exists y.(w\ @\ y \wedge z\ @\ y)\ @\ \lambda x.plane(x) =$$
$$\lambda z.\exists y.(\lambda x.plane(x)\ @\ y \wedge z\ @\ y) =$$
$$\lambda z.\exists y.(plane(y) \wedge z\ @\ y).$$

Note that this lambda expression encodes the assumption that the noun phrase is followed by a verb. This is achieved by introducing z as a placeholder for the verb.

The representation of proper names is designed, as well, to allow the combination of the name with the other parts of the sentence. For instance, "John" is represented by:

$$\lambda u.(u\ @\ john),$$

where u is a placeholder for a lambda expression of the form $\lambda x.f(x)$, which can be intuitively read (if $f(\cdot)$ is an action) "an unnamed actor x performed action f". So, for example, the sentence "John did f" is represented as:

$$\lambda u.(u\ @\ john)\ @\ \lambda x.f(x).$$

As usual, the right part of the expression can be substituted to u, which leads us to:

$$\lambda x.f(x)\ @\ john.$$

The expression can be immediately simplified into:

$$f(john).$$

The encoding of (transitive) verb phrases is based on a relation with both subject and direct object as arguments. The subject and direct object are introduced in the expression as placeholders, similarly to what we saw above. For example, the verb "take" is encoded as:

$$\lambda w.\lambda z.(w\ @\ \lambda x.take(z, x)),$$

where z and x are the placeholders for subject and direct object respectively. The assumption, here, is that *the lambda expression of the direct object contains a placeholder for the verb*, such as z in $\lambda z.\exists y.(plane(y) \wedge z\ @\ y)$ above. Hence, when the representation of the direct object is substituted to w, the placeholder for the verb can be replaced by $\lambda x.take(z, x)$. Consider how this mechanism works on the phrase "takes a plane". The lambda expressions of the two parts of the phrase are directly

combined into:

$$\lambda w.\lambda z.(w \ @ \ \lambda x.take(z, x)) \ @ \ \lambda w.\exists y.(plane(y) \wedge w \ @ \ y).$$

As we said, the expression for the direct object is substituted to w, giving:

$$\lambda z.(\lambda w.\exists y.(plane(y) \wedge w \ @ \ y) \ @ \ \lambda x.take(z, x)).$$

Now, the placeholder for the verb, w, in the encoding of the direct object is replaced by (the remaining part of) the expression for the verb.

$$\lambda z.(\exists y.(plane(y) \wedge \lambda x.take(z, x) \ @ \ y) =$$
$$\lambda z.(\exists y.(plane(y) \wedge take(z, y))).$$

At this point we are ready to find the representation of the whole sentence, "John takes a plane". "John" and "takes a plane" are directly combined into:

$$\lambda u.(u \ @ \ john) \ @ \ \lambda z.(\exists y.(plane(y) \wedge take(z, y)))$$

which simplifies to:

$$\lambda z.(\exists y(plane(y) \wedge take(z, y))) \ @ \ john$$

and finally becomes:

$$\exists y(plane(y) \wedge take(john, y)).$$

It is worth stressing that the correctness of the encoding depends on the proper identification of subject, verb, and objects of the sentences. If, in the example above, "John" were to be identified as direct object of the verb, the resulting encoding would be quite different.

As this example shows, lambda calculus offers a simple and elegant way to determine the logical representation of the discourse, in terms of first-order logic formulas encoding the meaning of the text. Notice, however, that the lambda calculus specification alone does not help in dealing with some of the complexities of natural language, and in particular with *ambiguities*. Consider the sentence "John took a flower". A possible first-order representation of its meaning is:

$$\exists y(flower(y) \wedge take(john, y)).$$

Although in this sentence verb "take" has a quite different meaning from the one of "take a plane", the logical representations of the two sentences are virtually identical. We describe now a different approach that is aimed at providing information to help disambiguate the meaning of sentences.

This alternative approach translates the discourse into logical statements that we will call *LCC-style Logic Forms* (LLF for short). Logic forms of this type were originally introduced in [44, 45], and later substantially extended in, e.g., [42, 21]. (Note that as mentioned in Chapter 8 of [6], there have been many other logic form proposals, such as [73, 60, 66].) Here, by LLF, we refer to the extended type of logical representation of [42, 21]. In the LLF approach, a triple $\langle base, pos, sense \rangle$ is associated with every noun, verb, adjective, adverb, conjunction and preposition, where *base* is the base form of the word, *pos* is its part-of-speech, and *sense* is the word's sense

in the classification found in the WordNet database [54, 26]. Notice that such tuples provide richer information than the lambda calculus based approach, as they contain sense information about the lexical items (which helps understand their semantic use).

In the LLF approach, logic constants are (roughly) associated with the words that introduce relevant parts of the sentence (sometimes called *heads of the phrases*). The association is obtained by atoms of the form:

$$base\_pos\_sense(c, a_0, \ldots, a_n),$$

where *base*, *pos*, *sense* are the elements of the triple describing the head word, c is the constant that denotes the phrase, and $a_0, \ldots, a_n$ are constants denoting the sub-parts of the phrase. For example, "John takes a plane" is represented by the collection of atoms:

$$John\_NN(x1), take\_VB\_11(e1, x1, x2), plane\_NN\_1(x2).$$

The first atom says that $x1$ denotes the noun (NN) "John" (the sense number is omitted when the word has only one possible meaning). The second atom describes the action performed by John. The word "take" is described as a verb (VB), used with meaning number 11 from the WordNet 2.1 classification (i.e., "travel or go by means of a certain kind of transportation, or a certain route"). The corresponding part of the discourse is denoted by $e1$. The second argument of relation *take_VB_11* denotes the syntactic subject of the action, while the third is the syntactic direct object.

The relations of the form *base_pos_sense* can be classified based on the type of phrase they describe. More precisely, there are six different types of predicates:

1. verb predicates
2. noun predicates
3. complement predicates
4. conjunction predicates
5. preposition predicates
6. complex nominal predicates

In recent papers [56], verb predicates have been used with variable number of arguments, but no less than two. The first required argument is called *action/eventuality*. The second required argument denotes the subject of the verb. Practical applications of logic forms [1] appear to use the older *fixed slot allocation* schema [58], in which verbs always have three arguments, and dummy constants are used when some parts of the text are missing. For sake of simplicity, in the rest of the discussion, we consider only the fixed slot allocation schema.

Noun predicates always have arity one. The argument of the relation is the constant that denotes the noun.

Complement relations have as argument the constant denoting the part of text that they modify. For example, "run quickly" is encoded as (the tag RB denotes an adverb):

$$run\_VB\_1(e1, x1, x2), quickly\_RB(e1).$$

Conjunctions are encoded with relations that have a variable number of arguments, where the first argument represents the "result" of the logical operation induced by the conjunction [65, 58]. The other arguments encode the parts of the text that are connected by the conjunction. For example, "consider and reconsider carefully" is represented as:

$$and\_CC(e1, e2, e3), consider\_VB\_2(e2, x1, x2),$$
$$reconsider\_VB\_2(e3, x3, x4), carefully\_RB(e1).$$

One preposition atom is generated for each preposition in the text. Preposition relations have two arguments: the part of text that the prepositional phrase is attached to, and the prepositional object. For example, "play the position of pitcher" is encoded as:

$$play\_VB\_1(e1, x1, x2), position\_NN\_9(x2),$$
$$of\_IN(x2, x3), pitcher\_NN\_4(x3).$$

Finally, complex nominals are encoded by connecting the composing nouns by means of the *nn_NNC* relation. The *nn_NNC* predicate has a variable number of arguments, which depends on the number of nouns that have to be connected. For example, "an organization created for business ventures" is encoded as:

$$organization\_NN\_1(x2), create\_VB\_2(e1, x1, x2),$$
$$for\_IN(e1, x3),$$
$$nn\_NNC(x3, x4, x5), business\_NN\_1(x4), venture\_NN\_3(x5).$$

An important feature of the LLF approach is that the logic forms are also augmented with *named-entity tags*, based on *lexical chains among concepts* [43]. Lexical chains are sequences of concepts such that adjacent concepts are connected by an hypernymy relation.[2] Lexical chains allow to add to the logic forms information *implied by the text*, but not explicitly stated. For example, the logic form of "John takes a plane" contains a named-entity tag:

$$human\_NE(x1),$$

stating that John (the part of the sentence denoted by $x1$) is a human being. The named-entity tag is derived from the lexical chain connecting name "John" to concept "human (being)".

A recent extension of this approach consists in further augmenting the logic forms by means of *semantic relations*—relations between two words or concepts that provide a somewhat deeper description of the meaning of the text.[3] More than 30 different types of semantic relations have been identified, including:

[2] Recall that a word is a hypernym of another if the former is more generic or has broader meaning than the latter.

[3] Further information can be found at:
http://www.hlt.utdallas.edu/~moldovan/CS6373.06/IS_Knowledge_Representation_from_Text.pdf,
http://www.hlt.utdallas.edu/~moldovan/CS6373.06/IS_SC.pdf, and
http://www5.languagecomputer.com/demo/polaris/PolarisDefinitions.pdf.

- Possession ($POS\_SR(X, Y)$): X is a possession of Y.
- Agent ($AGT\_SR(X, Y)$): X performs or causes the occurrence of Y.
- Location, Space, Direction ($LOC\_SR(X, Y)$): X is the location of Y.
- Manner ($MNR\_SR(X, Y)$): X is the way in which event Y takes place.

For example, the agent in the sentence "John takes a plane" is identified by:

$$AGT\_SR(x1, e1).$$

Notice that the entity specified by *AGT_SR* does not always coincide with the subject of the verb.

The key step in the automation of the generation of logic forms is the construction of a parse tree of the text by a syntactic parser. The parser begins by performing word-sense disambiguation with respect to WordNet senses [54, 26] and determines the parts of speech of the words. Next, grammar rules are used to identify the syntactic structure of the discourse. Finally, the parse tree is augmented with the word sense numbers from WordNet and with named-entity tags.

The logic form is then obtained from the parse tree by associating atoms to the nodes of the tree. For each atom, the relation is determined from the triple $\langle base, pos, sense \rangle$ that identifies the node. For nouns, verbs, compound nouns and coordinating conjunction, a fresh constant is used as first argument (*independent argument*) of the atom and denotes the corresponding phrase. Next, the other arguments (*secondary arguments*) of the atoms are assigned according to the arcs in the parse tree. For example, in the parse tree for "John takes a plane", the second argument of *take_VB_*11 is filled with the constant denoting the sub-phrase "John", and the third with the constant denoting "plane".

Named-entity tagging substantially contributes to the generation of the logic form when the parse tree contains ambiguities. Consider the sentences [56]:

1. They gave the visiting team a heavy loss.
2. They played football every evening.

Both sentences contain a verb followed by two noun phrases. In (1), the direct object of the verb is represented by the second noun phrase. This is the typical interpretation used for sentences of this kind. However, it is easy to see that (2) is an exception to the general rule, because there the direct object is given by the *first* noun phrase.

Named-entity tagging allows the detection of the exception. In fact, the phrase "every evening" is tagged as an indicator of time. The tagging is taken into account in the assignment of secondary arguments, which allows to exclude the second noun phrase as a direct object and correctly assign the first noun phrase to that role.

Finally, semantic relations are extracted from text with a pattern identification process:

1. Syntactic patterns are identified in the parse tree.
2. The features of each syntactic pattern are identified.
3. The features are used to select the applicable semantic relations.

Although the extraction of semantic relations appears to be at an early stage of development (the process has not yet been described in detail by the LCC research group), preliminary results are very encouraging (see Section 20.4 for an example of the use of semantic relations).

The approach for the mapping of English text into LLF has been used, for example, in the LCC QA system *PowerAnswer* [1, 20].

In the next section, we turn our attention to the reasoning task, and briefly describe the reasoning component of the LCC QA system.

20.3 The COGEX Logic Prover of the LCC QA System

The approach used in many recent QA systems is roughly based on detecting matching patterns between the question and the textual sources provided, to determine which ones are answers to the question. We call the textual sources available to the system *candidate answers*. Because of the ambiguity of natural language and of the large amount of synonyms, however, these systems have difficulties reaching high success rates (see, e.g., [20]). In fact, although it is relatively easy to find fragments of text that possibly contain the answer to the question, it is typically difficult to associate to them some kind of measure allowing to select one or more *best answers*. Since the candidate answers can be conflicting, the inability to rank them is a substantial shortcoming.

To overcome these limitations, the LCC QA system has been recently extended with a prover called COGEX [20]. In high-level terms, COGEX is used to analyze the connection between the question in input and the candidate answers obtained using traditional QA techniques. Consider the question "Did John visit New York City on Dec, 1?" and assume that the QA system has access to data sources containing the fragments "John flew to the City on Dec, 1" and "In the morning of Dec, 1, John went down memory lane to his trip to Australia". COGEX is capable of identifying that the connection between question and candidate answer requires the knowledge that "New York City" and "City" denote the same location, and that "flying *to* a location" implies that the location will be visited. The type and number of these differences is used as a measure of how close a question and candidate answer are—in our example, we would expect that the first answer will be considered the closest to the question (as the second does not describe an *actual* travel on Dec, 1). This measure gives an ordering of the candidate answers, and ultimately allows the selection of the best matches.

The analysis carried out by COGEX is based on world knowledge extracted from WordNet (e.g., the description of the meaning of "fly (to a location)") as well as knowledge about natural language (allowing to link "New York City" and "City"). In this context, the descriptions of the meaning of words are often called *glosses*.

To be used in the QA system, glosses from WordNet have been collected and mapped into logic forms. The resulting pairs $\langle word, gloss\_LLF \rangle$ provide definitions of *word*. Part of the associations needed to link "fly" and "visit" in the example above are encoded in COGEX by axioms (encoding complete definitions, from WordNet, of those verbs with the meanings used in the example) such as[4]:

[4] To complete the connection, axioms for "travel" and "go" are also needed.

$$\exists x_3, x_4 \forall e_1, x_1, x_2$$
$$\quad \textit{fly\_VB\_9}(e_1, x_1, x_2) \equiv$$
$$\quad\quad \textit{travel\_VB\_1}(e_1, x_1, x_4) \wedge \textit{in\_IN}(e_1, x_3) \wedge \textit{airplane\_NN}(x_3),$$

$$\exists x_3, x_4, x_9 \forall e_1, x_1, x_2$$
$$\quad \textit{visit\_VB\_2}(e_1, x_1, x_2) \equiv$$
$$\quad\quad \textit{go\_VB\_1}(e_1, x_1, x_9) \wedge \textit{to\_IN}(e_1, x_3) \wedge \textit{certain\_JJ}(x_3) \wedge \textit{place\_NN}(x_3) \wedge$$
$$\quad\quad \textit{as\_for\_IN}(e_1, x_4) \wedge \textit{sightseeing\_NN}(x_4).$$

(As discussed above, variables x_2, x_4 in the first formula and x_9 in the second are placeholders, used because verbs "fly", "travel", and "go" are intransitive.)

The linguistic knowledge is aimed at linking different logic forms that denote the same entity. Consider for instance the complex nominal "New York City" and the name "City". The corresponding logic forms are

$$\textit{New\_NN}(x_1), \textit{York\_NN}(x_2), \textit{City\_NN}(x_3), \textit{nn\_NNC}(x_4, x_1, x_2, x_3)$$

and

$$\textit{City\_NN}(x_5).$$

As the reader can see, although in English the two names sometimes denote the same entity, their logic forms alone do not allow to conclude that $x5$ and $x4$ denote the same object. This is an instance of a known linguistic phenomenon, in which an object denoted by a sequence of nouns can also be denoted by one element of the sequence. In order to find a match between question and candidate answer, COGEX automatically generates and uses axioms encoding instances of this and other pieces of linguistic knowledge. The following axiom, for example, allows to connect "New York City" and "City".

$$\forall x_1, x_2, x_3, x_4$$
$$\quad \textit{New\_NN}(x_1) \wedge \textit{York\_NN}(x_2) \wedge$$
$$\quad\quad \textit{City\_NN}(x_3) \wedge \textit{nn\_NNC}(x_4, x_1, x_2, x_3) \rightarrow \textit{City\_NN}(x_4).$$

Another example of linguistic knowledge used by COGEX is about equivalence classes of prepositions. Consider prepositions "in" and "into", which are often interchangeable. Also usually interchangeable are the pairs "at, in" and "from, of". It is often important for the prover to know about the similarities between these prepositions. Linguistic knowledge about it is encoded by axioms such as:

$$\forall x_1, x_2 \ (\textit{in\_IN}(x_1, x_2) \leftrightarrow \textit{into\_IN}(x_1, x_2)).$$

Other axioms are included with knowledge about appositions, possessives, etc.

From a technical point of view, for each candidate answer, the task of the prover is that of refuting the negation of the (logic form of the) question using the candidate answer and the knowledge provided. If the prover is successful, a correct answer has been identified. If the proof fails, further attempts are made by iteratively *relaxing* the question and finding a new proof. The introduction of the two axioms above, allowing

the matching of "New York City" with "City" and of "in" with "into", provides two examples of relaxation. Other forms of relaxation consist of uncoupling arguments in the predicates of the logic form, or removing prepositions or modifiers (when they are not essential to the meaning of the discourse). The system keeps track of how many relaxation steps are needed to find a proof. *This number is the measure of how close an answer and a question are*—the higher the value, the farther apart they are. If no proof is found after relaxing the question beyond a given threshold, the procedure is assumed to have failed. This indicates that the candidate is *not* an answer to the question.

Empirical evaluations of COGEX have given encouraging results. [20] reports on experiments in which the LCC QA system was tested, with and without COGEX, on the questions from the 2002 Text REtrieval Conference (TREC). According to the authors, the addition of COGEX caused a 30.9% performance increase.

Notice that, while the use of the prover increased performance, it did not bring any significant addition to the *class of questions* that can be answered. These systems can do a reasonable job at matching parts of the question with other text to find candidate answers, but they are not designed to perform inference (e.g., prediction) *on the story that the question contains*.

That is why the type of reasoning carried out by these QA systems is sometimes called *shallow reasoning*. Systems that can reason on the domain described by the question are instead said to perform *deep reasoning*. Although the above mentioned systems do not use domain knowledge and common-sense knowledge (recall that together they are referred to as background knowledge) that is needed for deep reasoning, they could do so. However it is not clear whether the 'iterative relaxing' approach would work in this case.

In the following two sections we describe two QA systems capable of deep reasoning, which use extraction of relevant facts from natural language text as a first step. We start with the DD system that takes as input a logical theory obtained from natural language text, as was described in this section.

20.4 Extracting Relevant Facts from Logical Theories and its Use in the DD QA System about Dynamic Domains and Trips

The DD system focuses on answering questions in natural language about the evolution of dynamic domains and is able to answer the kind of questions (such as reasoning about narratives, predictive reasoning, planning, counterfactual reasoning, and reasoning about intentions) we presented in Section 20.1.1. Its particular focus is on travel and trips. For example, given a paragraph stating "*John is in Paris. He packs the laptop in the carry-on luggage and takes a plane to Baghdad*", and a query *"Where is the laptop now?"*, DD will answer "Baghdad".

Notice that the task of answering questions of this kind requires fairly deep reasoning, involving not only logical inference, but also the ability to *represent and reason about dynamic domains and defaults*.

To answer the above question, the system has to know, for instance, that whatever is packed in the luggage normally stays there (unless moved), and that one's carry-on luggage normally follows him during trips. An important piece of knowledge is also that the action of taking a plane has the effect of changing the traveler's location to the destination.

In DD, the behavior of dynamic domains is modeled by *transition diagrams* [37, 38], directed graphs whose nodes denote states of the domain and whose arcs, labeled by actions, denote state transitions caused by the execution of those actions. The theory encoding a domain's transition diagram is called here *model of the domain*.

The language of choice for reasoning in DD is AnsProlog [33, 9] (also called A-Prolog [35, 36, 32]) because of its ability to both model dynamic domains and encode commonsense knowledge, which is essential for the type of QA task discussed here. As usual, problem solving tasks are reduced to computing models, called answer sets, of suitable AnsProlog programs. Various inference engines exist that automate the computation of answer sets.

20.4.1 The Overall Architecture of the DD System

The approach followed in the DD system for understanding natural language consists of translating the natural language discourse, in various steps, into its *semantic representation* (a similar approach can also be found in [14]), a collection of facts describing the semantic content of the discourse and a few linking rules. The task of answering queries is then reduced to performing inference on the theory consisting of the semantic representation and model of the domain.

More precisely, given a discourse H in natural language, describing a particular history of the domain, and a question Q, as well in natural language, the DD system:

1. obtains logic forms for H and Q;
2. translates the logic forms for H and Q into a *Quasi-Semantic Representation* (*QSR*), consisting of AnsProlog facts describing properties of the objects of the domain and occurrences of events that alter such properties. The representation cannot be considered fully semantic, because some of the properties are still described using syntactic elements of the discourse (hence the attribute *quasi*). The encoding of the facts is independent of the particular relations chosen to encode the model of the domain;
3. maps the QSR into an *Object Semantic Representation* (*OSR*), a set of AnsProlog atoms which describe the contents of H and Q using the relations with which the domain model is encoded. The mapping is obtained by means of AnsProlog rules, called *OSR rules*;
4. computes the answer sets of the AnsProlog program consisting of the OSR and the model of the domain and extracts the answer(s) to the question from such answer sets.

Although, in principle, steps 2 and 3 can be combined in a single mapping from H and Q into the OSR, their separation offers important advantages. First of all, separation of concerns: step 2 is mainly concerned with mapping H and Q into AnsProlog facts, while 3 deals with producing a semantic representation. Combining them would significantly complicate the translation. Moreover, the division between the two steps allows for a greater modularity of the approach: in order to use different logic form generators, only the translation at step 2 needs to be modified; conversely, we only need to act on step 3 to add to the system the support for new domains (assuming the

vocabulary of H and Q does not change). Interestingly, this multi-layered approach is also similar to one of the most widely accepted text comprehension models from cognitive psychology [48].

20.4.2 From Logic Forms to QSR Facts: An Illustration

Consider

- a history H consisting of the sentences "*John is in Paris. He packs the laptop in the carry-on luggage and takes a plane to Baghdad*",
- a query, Q, "*Where is the laptop at the end of the trip*?"

The first step consists in obtaining logic forms for H and Q. This task is performed by the logic form generator described in Section 20.2, that here we call LLF generator. Recall that LLFs consist of a list of atoms encoding the syntactic structure of the discourse augmented with some semantic annotations. For H, the LLF generator returns the following logic form, H_{lf}:

```
John_NN(x1)   & _human_NE(x1)  & be_VB_3(e1,x1,x27) &
in_IN(e1,x2)  & Paris_NN(x2)   & _town_NE(x2) &
AGT_SR(x1,e1) & LOC_SR(x1,x2)  &

pack_VB_1(e2,x1,x9)    &
laptop_NN_1(x9)        & in_IN(e2,x11)          &
carry-on_JJ_1(x12,x11) &
luggage_NN_1(x11)      & and_CC(e15,e2,e3)      &
take_VB_11(e3,x1,x13)  & plane_NN_1(x13)
& to_TO(e3,x14)        & Baghdad_NN(x14)        &
_town_NE(x14)          &

TMP_SR(x5,e2)  & AGT_SR(x1,e2)  & THM_SR(x9,e2) &
PAH_SR(x12,x11) & AGT_SR(x1,e3) &
THM_SR(x13,e3) & LOC_SR(x14,e3)
```

Here, *John_NN*($x1$) says that constant $x1$ will be used in the logic form to denote noun (NN) "John". Atom *be_VB*_3($e1, x1, x27$) says that constant $e1$ denotes a verb phrase formed by "to be", whose subject is denoted by $x1$. Hence, the two atoms correspond to "John is".[5]

One feature of the LLF generator that is important for the DD system is its ability to insert in the logic form simple semantic annotations and ontological information, most of which are extracted from the WordNet database [54, 26]. Recall that, for example, the suffix _3 in *be_VB*_3($e1, x1, x27$) says that the third meaning of the verb from the WordNet classification is used in the phrase (refer to Section 20.2 for more details). The availability of such annotations helps to identify the semantic contents of sentences, thus substantially simplifying the generation of the semantic representation in the following steps. For instance, the logic form of verb "take" above, *take_VB*_11($e3, x1, x13$) makes it clear that John did not actually *grasp* the plane.

[5] As this sense of verb "to be" does not admit a predicative complement, constant $x27$ is unused.

The logic form, Q_{lf}, for Q is:

```
laptop_NN_1(x5) & LOC_SR(x1,x5)
```

It can be noticed that the LLF generator does not generate atoms representing the verb. This is the feature that distinguishes the history from *where is/was/...* and *when is/was/...* queries at the level of logic form.[6] In the interpretation of the logic form of such queries, an important role is played by the *semantic relations* introduced by the LLF generator. Semantic relations are intended to give a rough description of the semantic role of various phrases in the discourse. For example, $LOC\_SR(x1, x5)$ says that the location of the object denoted by $x5$ is $x1$. Notice, though, that $x1$ is not used anywhere else in Q_{lf}: $x1$ is in fact a placeholder for the entity that must be identified to answer the question. *In general, in the LCC Logic Forms of this type of questions, the object of the query is identified by the constant that is not associated with any lexical item*. In the example above, $x2$ is associated to John by $John\_NN(x2)$, while $x1$ is not associated with any lexical item, as it only occurs in $LOC\_SR(x1, x5)$.

The second step of the process consists in deriving the QSR from H_{lf} and Q_{lf}. The steps in the evolution of the domain described by the QSR are called *moments*. Atoms of the form $true\_at(FL, M)$ are used in the QSR to state that property FL is true at moment M of the evolution. For example, the phrase corresponding to $be\_VB\_3(e1, x1, x27)$ (and associated atoms) is encoded in the QSR as:

```
true_at(at(john,paris), m(e1)).
```

where $at(john, paris)$ ("John is in Paris") is the property that holds at moment $m(e1)$. In fact, the third meaning of verb "to be" in the WordNet database is "occupy a certain position or area; be somewhere". Property $at(john, paris)$ is obtained from the atom $in\_IN(e1, x2)$ as follows:

- *in_IN* is mapped into property at;
- the first argument of the property is obtained by extracting from the LLF the actor of $e1$: first, the constant denoting the actor is selected from $be\_VB\_3(e1, x1, x27)$; next, the constant is replaced by the lexical item it denotes, using the LLF $John\_NN(x1)$.

Events that cause a change of state are denoted by atoms of the form *event* (*EVENT_NAME*, *EVENT_WORD*, *MEANING*, *M*), stating that the event denoted by *EVENT_NAME* and corresponding to *EVENT_WORD* occurred at moment *M* (with *MEANING* being the index of the meaning of the word in WordNet's classification). For instance, the QSR of the phrase associated with $take\_VB\_11(e3, x1, x13)$ is:

```
event(e3,take,11,m(e3)). actor(e3,john). object(e3,plane).
parameter(e3,to,baghdad).
```

[6] Yes/no questions have a simpler structure and are not discussed here to save space. The translation of the LLFs of *Where-* and *When*-queries that do not rely on verb "to be" (e.g., "where did John pack the laptop") has not yet been fully investigated.

The first fact states that the event of type "take" occurred at moment $m(e3)$ (with the meaning "travel or go by means of a certain kind of transportation, or a certain route") and is denoted by $e3$. The second and third fact specify the actor and the object of the event. Atom *parameter*($e3$, *to*, *baghdad*) states that the parameter of type *to* of the event is Baghdad.

A default temporal sequence of the moments in the evolution of the domain is extracted from H_{lf} by observing the order in which the corresponding verbs are listed in the logic form. Hence, the QSR for H_{lf} contains facts:

```
next(m(e1),m(e2)). next (m(e2),m(e3)).
```

The first fact states that the moment in which John is said to be in Paris precedes the one in which he packs. Notice that the actual order of events may be modified by words such as "after", "before", "on his way", etc. Although the issues involved in adjusting the order of events have not been investigated in detail, we believe that the default reasoning capabilities of AnsProlog provide a powerful way to accomplish the task.

Finally, the QSR of Q_{lf} is obtained by analyzing the logic form to identify the property that is being queried. Atom *LOC_SR*($x1$, $x5$) tells us that the query is about the location of the object denoted by $x5$. The corresponding property is *at*(*laptop*, C), where variable C needs to be instantiated with the location of the laptop as a result of the QA task. All the information is condensed in the QSR:

```
answer_true(C) :- eventually_true(at(laptop,C)).
```

The statement says that the answer to the query is C if *at*(*laptop*, C) is predicted to be true at the end of the story.

20.4.3 OSR: From QSR Relations to Domain Relations

The next step consists in mapping the QSR relations to the domain relations. Since the translation depends on the formalism used to encode the transition diagram, the task is accomplished by an *interface module* associated with the domain model. The rules of the interface module are called Object Semantic Representation rules (OSR rules for short).

The domain model used in our example is the *travel domain* [11, 34], a commonsense formalization of actions involving travel. The two main relations used in the formalization are h—which stands for holds and states which fluents[7] hold at each time point—and o—which stands for occurs and states which actions occur at each time point.

The key object of the formalization is the *trip*. Properties of a trip are its origin, destination, participants, and means of transportation. Action *go_on*(*Actor*, *Trip*) is a compound action that consists in embarking in the trip and departing.

Hence, the mapping from the QSR of event "take", shown above, is obtained by the following OSR rules (some rules have been omitted to save space):

[7] Fluents are relevant properties of the domain whose truth value may change over time [37, 38].

```
o(go_on(ACTOR,trip(Obj)), T) :- event(E,take,11,M),
                                actor(E,ACTOR),
                                object(E,Obj),
                                time_point(M,T).

h(trip_by(trip(Obj),Obj),T) :- event(E,take,11,M),
                               object(E,Obj),
                               time_point(M,T).

dest(trip(Obj),DEST) :- event(E,take,11,M),
                        parameter(E,to,DEST),
                        object(E,Obj).
```

The first rule states that, if the QSR mentions event "take" with sense 11 (in the WordNet database, this sense refers to travel), the actor of the event is *ACTOR* and the object is *Obj*, then the reasoner can conclude that action *go_on*(*ACTOR*, *trip*(*Obj*)) occurs at time point *T*. In this example, the time point is computed in a straightforward way from the sequence of moments encoded by relation *next* described in the previous section.[8] Notice that the name of the trip is for simplicity obtained by applying a function *trip* to the means of transportation used, but in more realistic cases this need not be.

Explicit information on the means of transportation used for the trip is derived by the second rule. The rule states that the object of event "take" semantically denotes the means of transportation. Because, in general, the means of transportation can change as the trip evolves, *trip_by* is a fluent.

The last rule defines the destination of the trip. A similar rule is used to define the origin.[9]

Atoms of the form *true_at*(*FL*, *M*) from the QSR are mapped into domain atoms by the rule:

```
h(FL,T) :- true_at(FL,M),
           time_point(M,T).
```

The mapping of relation *eventually_true*, used in the QSR for the definition of relation *answer_true*, is symmetrical:

```
eventually_true(FL) :- h(FL,n).
```

where *n* is the constant denoting the time point associated with the end of the evolution of the domain.

Since the OSR rules are written in AnsProlog, the computation of the OSR can be combined with the task of finding the answer given the OSR: in our approach, the answer to *Q* is found by computing, in a single step, the answer sets of the AnsProlog program consisting of the QSR, the OSR rules, and the model of the travel domain.

[8] Recall that, in more complex situations, the definition of relation *time_point* can involve the use of defaults, to allow the assignment of time points to be refined during the mapping.

[9] Since in the travel domain the origin and destination of trips do not change over time, the formalization is designed to allow to specify the origin using a static relation rather than a fluent. This simplification is not essential and can be easily lifted.

A convenient way of extracting the answer when SMODELS[10] is used as inference engine, is to add the following two directives to the AnsProlog program:

```
#hide. #show answer_true(C).
```

As expected, for our example SMODELS returns[11]:

```
answer_true(baghdad).
```

20.4.4 An Early Travel Module of the DD System

As mentioned earlier, and as is necessary in any QA system performing deep reasoning, the DD system combines domain knowledge and common-sense knowledge together with information specific to the instance, extracted from text, questions, and the mapping rules (of the previous subsection). As a start the DD system focused on domain knowledge about travels and trips (which we briefly mention in the previous subsection) and contained rules for commonsense reasoning about dynamic domains. In this section we briefly describe various parts of an early version of this background knowledge base, which is small enough to be presented in its entirety, but yet shows various important aspects of representation and reasoning.

Facts and basic relations in the travel module

The main objects in the travel modules are *actions*, *fluents* and *trips*. In addition there are various domain predicates and a Geography module.

1. Domain predicates: The predicates include predicates such as $person(X)$, meaning X is a person; $l(Y)$, meaning Y is a possible location of a trip; $time\_point(X)$, meaning X is a time point; $travel\_documents(X)$, meaning X is a travel document such as passports and tickets; $belongings(X)$, meaning X is a belonging such as a laptop or a book; $luggage(carry\_on(X))$, meaning X is a carry-on luggage; $luggage(lugg(X))$, meaning X is a regular (non-carry-on) luggage; $possession(X)$, meaning X is a possession; $type\_of\_transp(X)$, meaning X is a type of transportation; $action(X)$ meaning X is an action; $fluent(X)$ meaning X is a fluent; and $day(X)$ meaning X is a day.

2. The Geography module and related facts: The DD system has a simple geography module with predicates $city(X)$ denoting X is a city; $country(X)$ denoting X is a country; $union(X)$ denoting X is a union of countries such as the European Union; and $in(XCity, Y)$ denoting $XCity$ is in the country or union Y. In addition it has facts such as $owns(P, X)$, meaning person P owns luggage X; $vehicle(X, T)$ meaning X is a vehicle of type T; $h(X, T)$ meaning fluent X holds at time point T; and $time(T, day, D)$ meaning the day corresponding to time point T is D.

3. The Trips: The DD system has the specification of an activity "trip". Origins and destinations of trips are explicitly stated by the facts $origin(j, C1)$ and $dest(j, C2)$.

4. Actions and actors: The DD system has various actions such as $depart(J)$, meaning trip J departs from its origin; $stop(J, C)$, meaning trip J stops at city C; $go\_on(P, J)$, meaning person P goes on trip J; $embark(P, J)$, meaning person P

[10] http://www.tcs.hut.fi/Software/smodels/.

[11] The issue of translating the answer back into natural language will be addressed in future versions of the system.

embarks on trip J; and *disembark*(P, J), meaning person P disembarks from trip J. In each of these actions J refers to a trip. Other actions include *get*(P, PP), meaning person P gets possession PP; *pack*(P, PP, C), meaning person P packs possession PP in container C; *unpack*(P, PP, C), meaning person P packs possession PP in container C; and *change_to*(J, T), meaning trip J changes to the type of transportation T. The domain contains facts about actions and actors. For example, the fact *action*(*depart*(j)) means that *depart*(j) is an action; and the fact *actor*(*depart*(j), j) means that j is the actor of the action *depart*(j).

5. *Fluents:* The DD system has various fluents such as *at*(P, D), meaning the person P is at location D; *participant*(P, J), meaning the person P is a participant of trip J; *has_with_him*(P, PP), meaning person P has possession PP with him; *inside*(B, C), meaning B is inside the container C; and *trip_by*(J, T), meaning the trip J is using the transportation type T.

The rules in the travel module

We now present various rules of the travel module. We arrange these rules in groups that have a common focus on a particular aspect.

6. *Inertia:* The following two rules express the commonsense law of inertia that normally fluents do not change their value.

```
h(Fl,T+1)  :- T < n,   h(Fl,T),   not -h(Fl,T+1).
-h(Fl,T+1) :- T < n,  -h(Fl,T),   not  h(Fl,T+1).
```

7. *Default values of some fluents:* The following two rules say that, normally, people have their passport and their luggage with them at the beginning of the story.[12] Here, 0 denotes initial time point. (A different number could have been used with minor changes in few other rules.)

```
h(has_with_him(P,passport(P)),0) :-
                  not -h(has_with_him(P,passport(P)),0).
h(has_with_him(P,Luggage),0) :-
                  owns(P,Luggage),
                  not -h(has_with_him(P,Luggage),0).
```

8. *Agent starting a journey:* The following two rules specify that normally people start their journey at the origin of the journey.

```
h(at(J,C),0) :- o(go_on(P,J),0), origin(J,C),
                not -h(at(J,C),0).
h(at(P,C),0) :- o(go_on(P,J),0), origin(J,C),
                not -h(at(P,C),0).
```

9. *Direct and Indirect effect of the action embark:* The effects of the action *embark* and its executability conditions are expressed by the rules given below.

The following rule expresses that a person after embarking on a journey on a plane no longer has his luggage with him.

[12] Obviously these defaults are meaningful only in the context of travel-related stories, and can be suitably qualified in AnsProlog . We omit the qualification to simplify the presentation.

```
-h(has_with_him(P,lugg(P)),T+1) :- o(embark(P,J),T),
                                   h(trip_by(J,plane),T).
```

The following three rules express conditions under which a person can embark on a journey: he must be a participant; he must be at the location of the journey and he must have all that he needs to embark on that journey.

```
-o(embark(P,J),T) :- -h(participant(P,J),T).
-o(embark(P,J),T) :-  h(at(P,D1),T), h(at(J,D2),T),
                      neq(D1,D2).
-o(embark(P,J),T) :-  need(P,TD,J),
                     -h(has_with_him(P,TD),T).
```

The following rules define what person needs to go embark on a trip. The first rule says he normally needs a passport if he is traveling between two different countries. The third rule states an exception that one traveling between two European Union countries does not need a passport. The fourth rule states that one normally needs a ticket for a journey. The fifth rule states an exception that for a car trip one does not need a ticket. The last two rules define a car trip as a trip which started as a car trip and which has not changed its mode of transportation.

```
need(P,passport(P),J)  :- place(embark(P,J),C1),
                          dest(J,C2), diff_countries(C1,C2),
                          not -need(P,passport(P),J).
diff_countries(C1,C2)  :- in(C1,Country1), in(C2,Country2),
                          neq(Country1,Country2).
-need(P,passport(P),J) :- citizen(P,eu),
                          place(embark(P,J),C1),
                          dest(J,C2), in(C1,eu), in(C2,eu).

need(P,tickets(J),J)  :- not -need(P,tickets(J),J).
-need(P,tickets(J),J) :- car_trip(J).
-car_trip(J)          :- h(trip_by(J,TypeOfTransp),T),
                         neq(TypeOfTransp,car).
car_trip(J)           :- h(trip_by(J,car),0),
                         not -car_trip(J).
```

10. Direct and Indirect effect of the action disembark: The direct and indirect effects of the action *disembark* and its executability conditions are expressed by the rules given below.

The first two rules express that by disembarking a person is no longer a participant of a trip and unless his luggage is lost, he has his luggage with him. The third and fourth rules specify that one cannot disembark from a trip at a particular time if he is not a participant at that time, or if the journey is en route at that time.

```
-h(participant(P,J),T+1) :- o(disembark(P,J),T).
 h(has_with_him(P,lugg(P)),T+1) :-
                           o(disembark(P,J),T),
                           o(embark(P,J),T1),
                           h(has_with_him(P,lugg(P)),T1),
                           not h(lost(lugg(P)),T+1).
```

```
-o(disembark(P,J),T) :- -h(participant(P,J),T).
-o(disembark(P,J),T) :-  h(at(J,en_route),T).
```

11. Rules about the action go_on: The action *go_on* is viewed as a composite action consisting of first embarking and then departing. This is expressed by the first two rules below. The third rule states that a plane trip takes at most a day.

```
o(embark(P,J),T) :-  o(go_on(P,J),T).
o(depart(J),T+1) :-  o(go_on(P,J),T).

time(T2,day,D) | time(T2,day,D + 1) :- o(go_on(P,J),T1),
                                       o(disembark(P,J),T2),
                                       time(T1,day,D),
                                       h(trip_by(J,plane),T1).
```

12. Effect of the action get: The first rule below states that if one gets something then he has it. The second rule states that getting a passport could take at least three days. Rules that compute the duration of an action are discussed later in item 16.

```
h(has_with_him(P,PP),T+1) :- o(get(P,PP),T).
:- duration(get(P,passport(P)),Day), Day < 3.
```

13. Effect axioms and executability conditions of the actions pack and unpack:
The first two rules below state the effect of packing and unpacking a possession inside a container. The third and fourth rule state when one can pack a possession and the fifth and sixth rules state when one can unpack a possession.

```
h(inside(PP,Container),T+1)  :- o(pack(P,PP,Container),T).
-h(inside(PP,Container),T+1) :- o(unpack(P,PP,Container),T).

-o(pack(P,PP,Container),T)   :- -h(has_with_him(P,PP),T).
-o(pack(P,PP,Container),T)   :- -h(has_with_him(P,Container),T).
-o(unpack(P,PP,Container),T) :- -h(has_with_him(P,Container),T).
-o(unpack(P,PP,Container),T) :- -h(inside(P,Container),T).
```

14. Direct and Indirect effects (including triggers) of the actions depart and stop:
The first two rules below express the impact of departing and stopping. The third rule says that a stop at the destination of a journey is followed by disembarking of the participants of that journey. The fourth rule says that a stop in a non-destination is normally followed by a depart action. The fifth and sixth rules give conditions when departing and stopping is not possible. The seventh rule says that normally a trip goes to its destination. The eighth rule says that after departing one stops at the next stop. The last rule states that one can stop at only one place at a time.

```
h(at(J,en_route),T+1) :- o(depart(J),T).
h(at(J,C),T+1)        :- o(stop(J,C),T).

o(disembark(P,J),T+1) :- h(participant(P,J),T),
                         o(stop(J,D),T), dest(J,D).
```

```
o(depart(J),T+1)       :- o(stop(J,C),T), not dest(J,C),
                          not -o(depart(J),T+1).

-o(depart(J),T)        :- h(at(J,en_route),T).
-o(stop(J,C),T)        :- -h(at(J,en_route),T).
o(stop(J,C),T)         :- h(at(J,en_route),T), dest(J,C),
                          not -o(stop(J,C),T).

o(stop(J,C2),T+1)      :- leg_of(J,C1,C2), h(at(J,C1),T),
                          o(depart(J),T).
-o(stop(J,C),T)        :- o(stop(J,C1),T), neq(C,C1).
```

15. Effect of changing the type of transportation:

```
h(trip_by(J,Transp),T+1) :- o(change_to(J,Transp),T).
```

16. State constraints about the dynamic domain: The following are rules that encode constraints about the dynamic domain. The first rule states that an object can only be in one place at a particular time. The second rule states that a trip can only have one type of transportation at a particular time. The third rule states that if a person is at a location then his possessions are also at the same location. The fourth rules states that a participant of a trip is at the same location as the trip. The fifth rules states that if a person has a container then he also has all that is inside the container. The last rule defines the duration of an action based on the mapping between time points and days. (It assumes that all actions occurring at a time point have the same duration.)

```
-h(at(O,D1),T)            :- h(at(O,D2),T), neq(D1,D2).
-h(trip_by(J,Transp2),T) :- h(trip_by(J,Transp1),T),
                            neq(Transp1,Transp2).

h(at(PP,D),T)   :- h(has_with_him(P,PP),T), h(at(P,D),T).
h(at(P,D),T)    :- h(participant(P,J),T), h(at(J,D),T).

h(has_with_him(P,PP),T) :- h(inside(PP,Container),T),
                           h(has_with_him(P,Container),T).
duration(A,D) :- action(A), o(A,T), time(T,day,D1),
                 time(T+1,day,D2), D = D2 - D1.
```

20.4.5 Other Enhancements to the Travel Module

The module in the previous section is only sufficient with respect to some of the text question pairs of Section 20.1.1. For others we need additional modules, such as planning modules, modules for reasoning about intentions, and modules that can map time points to a calender.

Planning

Planning with respect to a goal can be done by writing rules about whether a goal is satisfied at the desired time points; writing rules that eliminate models where the goal is not satisfied and then writing rules that enumerate possible action occurrences. With respect to the example in Section 20.1.1 (fifth item), the following rules suffice.

```
answer_true :-   o(go_on(john,j,T)), origin(j,boston),
                   dest(j,paris), time(T,day,4).
yes :-answer_true.

:- not yes.

{o(Act,T) : action(Act) : actor(Act,P)}1 :- T < n-1.
```

The first rule states that the answer to query q is "true" if John performs the action of going to Paris on day 4. The next two rules say that it is impossible for the answer not to be "true". Finally, the last rule states that any action can occur at any time step.

Reasoning about intentions

To reason about intentions one needs to formalize commonsense rules about intentions [10]. One such rule is that an agent after forming an intention will normally attempt to achieve it. Another rules is that an agent will not usually give up on its intentions without good reason; i.e., intentions persist. We now give a simple formalization of these. We assume that intentions are a sequence of distinct actions.

In the following *intended_seq*(S, I) means that the sequence of actions S is intended starting from time point I. Similarly, *intended_action*(A, I) means that the action A is intended (for execution) at time point I.

```
intended_action(A,I)   :- intended_seq(S,I), seq(S,1,A).

intended_action(B,K+1) :- intended_seq(S,I), seq(S,J,A),
                          occurs(A,K), time_point(K),
                          seq(S,J+1,B).

occurs(A,I) :- action(A), intended_action(A,I),
               time_point(I), not -occurs(A,I).

intended_action(A,I+1) :- action(A), time_point(I),
                          intended_action(A,I),
                          not occurs(A,I).
```

The first rule above encodes that an individual action A is intended for execution at time point I, if, A is the first action of a sequence which is intended to be executed starting from time point I. The second rule encodes that an individual action B is intended for execution at time point $K + 1$, if B is the $(J + 1)$th action of a sequence intended to be executed at an earlier time point and the Jth action of that sequence is A which is executed at time point K. The third rule encodes the notion that intended actions occur unless they are prevented. The last rule encodes the notion that if an intended action does not occur as planned then the intention persists.

20.5 From Natural Language to Relevant Facts in the ASU QA System

In the previous section relevant facts and some question-related rules were obtained from natural language by processing a logic form of the natural language. In this

section we briefly mention an alternative approach from [71] where the output of a semantic parser is used directly in obtaining the relevant facts. In addition we illustrate the use of knowledge in reducing semantic ambiguities. Thus knowledge and reasoning is not only useful in obtaining answers but also in understanding natural language.

In the ASU QA system to extract the relevant facts from sentences, Link Grammar [70] is used to parse the sentences so that the dependent relations between pairs of words are obtained. Such dependent relations are known as *links*. The Link Grammar parser outputs labeled links between pairs of words for a given input sentence. For instance, if word a is associated with word b through the link "S", a is identified as the subject of the sentence while b is the finite verb related to the subject a. From the links between pairs of words, a simple algorithm is then used to generate AnsProlog facts. A simplified subset of the algorithm is presented as follows:

Input: Pairs of words with their corresponding links produced by the Link Grammar parser.

Output: AnsProlog facts.

Suppose e_i is the current event number[13] and the event is described in the jth sentence of the story.

1. Form the facts $in\_sentence(e_i, j)$ and $event\_num(e_i)$.

2. If word a is associated with word b through the link "S" (indicating a is a subject noun related to the finite verb b), then form the facts $event\_actor(e_i, a)$ and $event\_nosense(e_i, b)$. If a appears in the name database, then form the fact $person(a)$.

3. If word a is associated with word b through the link "MV" (indicating a is a verb related to modifying phrase b), and b is also associated with word c through the link "J" (indicating b is a preposition related to object c), then form the fact $parameter(e_i, b, c)$. If c appears in the city database, then form the fact $city(c)$.

4. If word a is associated with word b through the link "O" (indicating a is a transitive verb related to object b), then form the facts $noun(b)$ and $object(e_i, b)$.

5. If word a is associated with word b through the link "ON" (indicating a is the preposition "on" related to certain time expression b) and b is also associated with word c through the link "TM" (indicating b is a month name related to day number c), then form the fact $occurs(e_i, b, c)$.

6. If word a is associated with word b through the link "Dmcn" (indicating a is the clock time and b is AM or PM), then form the fact $clock\_time(a)$. (Here a is a time as one reads in a clock and hence is more fine grained than the information in the earlier used predicate *time_point*.)

[13] We use a complex sentence processer that processes complex sentences to a set of simple sentences. Thus we assume that there is one event in each sentence. We assign event numbers sequentially from the start of the text. This is a simplistic view and there have been some recent work on more sophisticated event analysis, such as in [47].

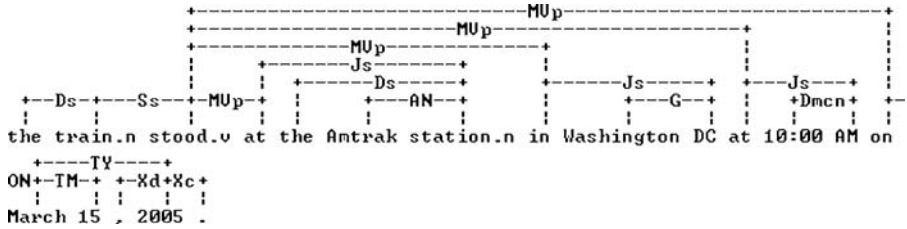

Figure 20.2: Output of the Link Grammar Parser for "*The train stood at the Amtrak station in Washington DC at 10:00 AM on March 15, 2005*".

7. If word a is associated with word b through the link "TY" (indicating b is a year number related to date a), then form the fact $occurs\_year(e_i, b)$.

8. If word a is associated with word b through the link "D" (indicating a is a determiner related to noun b), then form the fact $noun(b)$.

To illustrate the algorithm, the Link Grammar output for the sentence "The train stood at the Amtrak station in Washington DC at 10:00 AM on March 15, 2005." is shown below in Fig. 20.2.

The following facts are extracted based on the Link Grammar output:

```
event_num(e1).                          in_sentence(e1,1).
event_actor(e1,train).                  event_nosense(e1,stood).
parameter(e1,at,amtrak_station).
parameter(e1,in,washington_dc).
parameter(e1,at,t10_00am).              occurs(e1,march,15).
occurs_year(e1,2005).                   person(\mathit{john}).
city(washington_dc).                    verb(stood).
noun(train).                            noun(amtrak_station).
clock_time(t10_00am).
```

In the above extracted facts, the constant $e1$ is an identifier that identifies related facts extracted from the same sentence. Atoms such as $noun(train)$, $verb(stood)$ are event independent and thus no event number is assigned to such facts. The atom $event\_nosense(e1, stood)$ indicates that word sense has yet to be assigned to the word *stood*.

After extracting the facts from the sentences, it is necessary to assign the correct meanings of nouns and verbs with respect to the sentence. The process of identifying the types utilizes WordNet hypernyms. Word a is a hypernym of word b if a has a "is-a" relation with b. In the travel domain, it is essential to identify nouns that are of the types transportation (denoted as *tran*) or person (denoted as *person*). Such identification is performed using predefined sets of hypernyms for both transportation and person. Let H_t be a set of hypernyms for type t. Noun a belongs to type t if a is a hypernym of $h \in H_t$, and a AnsProlog fact $t(a)$ is formed. The predefined sets of hypernyms of transportation and person are: $H_{tran} = \{travel, public\ transport, conveyance\}$ and $H_{person} = \{person\}$. For instance, the hypernym of the noun *train* is *conveyance*. So we assign a AnsProlog fact $transportation(train)$.

A similar process is performed for each extracted verb by using the hypernyms of WordNet. The component returns all possible senses of a given verb. Given the verb v and v has hypernym v', then the component returns the fact $is\_a(v, v')$. From the various possible senses of verbs, the correct senses are matched by utilizing the extracted facts related to the same event. AnsProlog rules are written to match the correct senses of verbs. The following rule is used to match the correct senses of a verb that has the meaning of *be*:

```
event(E,be) :- event_actor(E,TR),
               is_a(V,be), event_nosense(E,V),
               parameter(E,at,C), parameter(E,at,T).
```

The intuition of the above AnsProlog rule is that verb V has the meaning of *be* if event E has transportation TR as the actor and E involves city C, clock time T and V has the hypernym *be*. With the extracted facts, we can assign the meaning of *stood* to have the meaning of *be* in our example sentence.

Using the extracted facts together with verbs and nouns with their correct senses, reasoning is then done with an AnsProlog background knowledge base similar to the one in the DD system described in the previous section.

20.6 Nutcracker—System for Recognizing Textual Entailment

In the problem of recognizing textual entailment, the goal is to decide, given a text *Text* and a hypothesis *Hypothesis* expressed in a natural language, whether a human reasoner would call the hypothesis *Hypothesis* a consequence of the text. The following example is part of *Text\Hypothesis* pair No. 633 in the collection of problems proposed as the Second PASCAL Recognizing Textual Entailment Challenge [8]:

> *Text*: Yoko Ono unveiled a statue of her late husband, John Lennon.
>
> *Hypothesis*: Yoko Ono is John Lennon's widow.
>
> *Expected entailment:* Yes

We can see recognizing textual entailment (RTE) as a special case of the question answering problem. It is a textual answering task that covers only some aspects of general QA problem. Most of the systems that are designed to solve this problem [24, 8] reason directly on a natural language input by applying various statistical methods. These methods generally encounter problems when reasoning involves background knowledge. To recognize the fact that *Hypothesis* is "entailed" by *Text*, we often need to use some background commonsense knowledge. For instance, in the example above it is essential that "being a late wife" is a the same as "being a widow".

One approach to the RTE problem is to use first-order reasoning tools to check whether the hypothesis can be derived from the text conjoined with relevant background knowledge, after expressing all of them by first-order formulas. Bos and Markert employ this method in [17] and implemented in the system Nutcracker.[14] Related work is described in [5, 28].

[14] http://www.cogsci.ed.ac.uk/~jbos/RTE/.

We can summarize the approach to recognizing textual entailment employed by Bos and Markert as follows:

1. *Text* and *Hypothesis* are represented first by discourse representation structures [46] and then by first-order formulas T and C, respectively,
2. potentially relevant background knowledge is identified and expressed by a first-order formula BK,
3. an automated reasoning system, first-order logic theorem prover or model builder, is used to check whether the implication

$$T \wedge BK \rightarrow C$$

is logically valid.

Step 1 of this approach employs similar ideas as described in Section 20.2 where lambda calculus is used to build semantic representation of a text in the form of first-order logic formula. Instead, lambda calculus is used to build semantic representation of a text in the form of discourse representation structure (DRS) [16]. Next, discourse representation structure is translated into first-order logic formula as described in [15]. The intermediate step of building DRS for the text, for instance, allows the Nutcracker system to use the anaphora resolution mechanism that discourse representation theory [46] about DRSs provides. Consider

Text: Yoko Ono unveiled a statue of her late husband, John Lennon.

It has the following first-order logic representation produced by Nutcracker

$$\begin{aligned}\exists x\ y\ z\ e\ (&p\_ono(x) \wedge p\_yoko(x) \wedge r\_of(z, x) \wedge \\ &n\_statue(y) \wedge r\_of(y, z) \wedge \\ &a\_late(z) \wedge n\_husband(z) \wedge p\_lennon(z) \wedge p\_john(z) \wedge \\ &n\_event(e) \wedge v\_unveil(e) \wedge r\_agent(e, x) \wedge r\_patient(e, y)).\end{aligned}$$

It is interesting to note different prefixes $a\_$, $n\_$, $v\_$, $r\_$, $p\_$ that intuitively stand for adjective, noun, verb, relation, and person. The fact that Yoko Ono is a person or statue is a noun is available to Nutcracker from a syntax parse tree of a sentence produced by Combinatorial Categorial Grammar (CCG) parser[15] employed by the system. On the other hand unary predicates *n_event*, *r_agent* and *r_patient* are fixed symbols that are generated during the semantic analysis of the sentence by associating the transitive verb *unveil* with the event whose agent is Yoko Ono and patient is the statue.

Nutcracker approach benefits by choosing first-order logic as the formal language for representing semantic meaning of the sentence. First-order logic allows occurrence of negation, disjunction, implication, universal and existential quantifiers in the formula with arbitrary nesting. This provides a possibility to formally express various natural language phenomena. For example, for sentence "John has all documents"., Nutcracker produces the following first-order logic formula

[15] http://svn.ask.it.usyd.edu.au/trac/candc/wiki/.

$$\exists x(p\_john(x) \wedge$$
$$\forall y\ (n\_document(y) \rightarrow$$
$$\exists e\ (n\_event(e) \wedge v\_have(e) \wedge r\_agent(e, x) \wedge r\_patient(e, y)))).$$

To the best of our knowledge logic form employed by the LCC method described in Section 20.2 is not capable of properly representing the sentences of such type. I.e., the information about generalized quantifier *all* used in the sentence will be lost.

Unlike the LCC method that performs word sense disambiguation while producing logic form of the sentence, Nutcracker disregards this issue.

Step 2 of Nutcracker system that identifies potentially relevant background knowledge is based on the following principles. Words occurring in *Text* and *Hypothesis* are used as triggers for finding necessary background knowledge that is represented as a set of first-order logic axioms *BK*. Nutcracker generates the formula *BK* using hand coded database of background knowledge and automatically generated axioms.

Hand coded knowledge is of two types. One is domain specific, as for example, first-order logic formula

$$\forall x\ y\ (n\_husband(x) \wedge a\_late(x) \wedge r\_of(x, y) \rightarrow$$
$$(n\_widow(y) \wedge r\_of(y, x)))$$

that encodes the fact that if x is a late husband of y then y is a widow of x.[16] Other hand coded axioms represent the generic knowledge that cover the semantics of possessives, active-passive alternation, and spatial knowledge. Bos and Markert in [17] present the axiom

$$\forall e\ x\ y\ (n\_event(e) \wedge r\_agent(e, x) \wedge f\_in(e, y) \rightarrow f\_in(x, y))$$

as an example. It states that if an event occurs in some location then the agent of this event is at the same location. Note that restating this axiom as "*normally* if an event occurs in some location then the agent of this event is at the same location" is a nontrivial task for the first-order logic formalism. On the other hand, the approach described in Sections 20.4 and 20.5 where nonmonotonic AnsProlog language is used to represent the background knowledge suits well for representing such axioms.

Automatically generated knowledge is created by two means. One uses hypernym relations of WordNet to create an ontology for the nouns and verbs occurring in the text that corresponds to some snapshot of the general WordNet database. Such ontology is called MiniWordnet and its construction mechanism is described in [16]. Its general structure is a tree whose nodes represent the words and the edges stand for the hypernym relations between the words. For example, MiniWordnet will, among others, contain the following hypernym relation for the sentence "Yoko Ono is John Lennon's widow.": *n_widow* is a hypernym of *n_person*. Nutcracker produces two kinds of first-order logic formulas that encode the knowledge represented by the MiniWordnet. First, it creates the implication for each hypernym relation that occurs in

[16] In fact such an axiom has a flaw. Consider a following pair *Text*: "Abraham is the husband of Sarah. Abraham is the father of Isaac. Isaac is the husband of Rebecca." and *Hypothesis*: "Abraham is the husband of Rebecca." Given a first-order logic representation of the pair and this axiom, *Text* entails *Hypothesis*. Resolving such issues is the problem of farther investigation.

the ontology. If MiniWordnet contains information that *n_widow* is a hypernym of *n_person* then the corresponding first-order formula is generated

$$\forall x\ (n\_widow(x) \rightarrow n\_person(x)).$$

It naturally can happen that one of the nodes in MiniWordnet has several children, i.e., several words are in hypernym relation with the node. Linguistic evidence suggests that the concepts (nonsynonyms) that are in hypernym relation with the same word are mutually exclusive. For instance, node that contains *n_person* might have two children that stand for *n_widow* and *n_husband*. In such case, Nutcracker generates the following two implications for *BK*

$$\forall x\ (n\_widow(x) \rightarrow \neg n\_husband(x)),$$
$$\forall x\ (n\_husband(x) \rightarrow \neg n\_widow(x)).$$

The second type of background knowledge automatically generate by the Nutcracker uses the syntax and lexical information provided by the parser. For instance, when the parser recognizes that *Yoko* is a person, the system will generate the following first-order logic formula

$$\forall x\ (p\_yoko(x) \rightarrow n\_person(x)).$$

The last step of the Nutcracker approach involves the use of an automated reasoning system, first-order logic theorem prover or model builder, to check whether the implication

$$T \wedge BK \rightarrow C \tag{20.1}$$

is logically valid. The formulas T and C are created during the Step 1 and correspond to *Text* and *Hypothesis* respectively. Formula BK, on the other hand, is the conjunction of the first-order formulas construction of which is described above.

Bos and Markert [17] propose the use of first-order logic tools in the following manner:

1. if a theorem prover finds a proof for the formula (20.1), Nutcracker concludes that *Text* entails *Hypothesis*.

2. if a theorem prover finds a proof for the formula

   $$\neg(T \wedge BK) \wedge C,$$

   then Nutcracker concludes that *Text* does not entail the *Hypothesis* due to the fact that they are inconsistent.

3. if a model builder finds a model for the negation of the formula (20.1)

   $$T \wedge BK \wedge \neg C \tag{20.2}$$

   then the system concludes that there is no entailment.

It is interesting to note that if the formula (20.2) belongs to the class of "effectively propositional", or "near-propositional" formulas [67] then it would be sufficient

to only use, so-called, effectively propositional reasoning (EPR) solvers to find an entailment. Effectively propositional formula is the universal closure of a quantifier-free formula in conjunctive normal form. On the class of such formulas the above three invocations of first-order tools can be reduced to one. For instance, model builder PARADOX[17] can also be seen as an EPR-solver, as it always recognizes a formula that can be converted into effectively propositional formula and is able to either find its models or state that the formula has no model. Furthermore, for effectively propositional formulas logic programming under stable model semantics can be used to verify the entailment.

This approach to RTE is related to QA approach described in Sections 20.4 and 20.5. First, Bos and Markert also consider the step of acquiring the related background knowledge as a vital element of a successful system for solving the RTE problem. Second, this method uses the first-order logic as the semantic representation language for the texts and background knowledge. Similarly, the systems described in Sections 20.4, 20.5 translate the natural language input and background knowledge into the AnsProlog rules. In both cases the representations have a formal model-theoretic semantics. Afterwards the approaches use general-purpose inference mechanisms designed for first-order logic and answer set programming inference, respectively.

20.7 Mueller's Story Understanding System

A different technique for obtaining a semantic representation of the discourse is described by Mueller in [62]. The technique uses *Event Calculus* [69, 55, 61] (which originated from [49] and evolved through [68]) for the semantic representation of the text. There, the discourse is initially mapped into a collection of *templates*—descriptions of the events consisting of frames with slots and slot fillers. Consider the text (this example is taken from [62]):

> *Bogota, 15 Jan 90—In an action that is unprecedented in Colombia's history of violence, unidentified persons kidnapped* 31 *people in the strife-torn banana-growing region of Uraba, the Antiouqia governor's office reported today. The incident took place in Puerto Bello, a village in Turbo municipality, 460 Km northwest of Bogota [. . .].*

Information extraction systems [2, 3] can be used to generate a template such as:

0. MESSAGE:ID DEV-MUC3-0040 (NNCOSC)
1. MESSAGE:TEMPLATE 1
2. INCIDENT:DATE – 15 JAN 90

[17] http://www.math.chalmers.se/~koen/paradox/.

3. INCIDENT: LOCATION COLOMBIA: URABA (REGION):
TURBO (MUNICIPALITY):PUERTO BELLO (VILLAGE)
4. INCIDENT: TYPE KIDNAPPING
5. INCIDENT: STAGE OF EXECUTION ACCOMPLISHED
[...]
8. PERP: INCIDENT CATEGORY TERRORIST ACT
9. PERP: INDIVIDUAL ID "UNIDENTIFIED PERSONS"/[...]
[...]
19: HUM TGT:NAME –
20. HUM TGT:DESCRIPTION: "VILLAGERS"
21. HUM TGT:NUMBER 31: "VILLAGERS"
22. HUM TGT:FOREIGN NATION –
23. HUM TGT:EFFECT OF INCIDENT –
24. HUM TGT:TOTAL NUMBER –

Next, each template is analyzed to find the *script* active in the template. The script determines the type of commonsense knowledge that the reasoner will use to understand the discourse. The above template is classified as matching the *kidnapping* script.

The pair consisting of the template and the script is then mapped into a *commonsense reasoning problem* encoding the initial state and narrative of events that take place in the story. Differently from what happens in the DD system, the commonsense reasoning problems for a particular script have a rather rigid structure: events listed in the script are *always* assumed to occur (apparently, even in the presence of contrary evidence from the text), while events mentioned in the story but not in the script are disregarded.

For the kidnapping script, the initial state and sequence of events are:

1. Initially the human targets are at a first location and the perpetrator is at a second location.
2. Initially the human targets are alive, calm, and uninjured.
3. The perpetrator loads a gun.
4. The perpetrator walks to the first location.
5. The perpetrator threatens the human targets with the gun.
6. The perpetrator grabs the human targets.
7. The perpetrator walks to the second location with the human targets.
8. The perpetrator walks inside a building.
9. The perpetrator lets go of the human targets.
10. For each human target:

(a) If the effect on the human target (from the template) is *death*, the perpetrator shoots the human target resulting in death.

(b) Otherwise, if the effect on the human target is *injury*, the perpetrator shoots the human target resulting in injury.

(c) Otherwise if the effect on the human target is *regained freedom*, the human target leaves the building and walks back to the first location.

Finally, reasoning is reduced to performing inferences on the theory formed by the commonsense reasoning problem and the commonsense knowledge selected based on the active script. The commonsense knowledge consists of Event Calculus axioms such as:

% An object can be only in one location at a time.

$HoldsAt(At(object, location1), time) \wedge$

$HoldsAt(At(object, location2), time) \Rightarrow$

$location1 = location2.$

% For an actor to activate a bomb, he must be holding it.

$Happens(BombActivate(actor, bomb), time) \Rightarrow$

$HoldsAt(Holding(actor, bomb), time).$

Next, we describe how Event Calculus theories can be used for question answering. Notice that the approach described in [62] does not explain how the questions are to be mapped into their logical representation.

For yes–no question answering about space:

Was actor "a" present when event "e" occurred?

- If for every time point t at which e occurs, the locations of a and that of the actor of e coincide, *the answer is "yes"*.
- If for every time point t at which e occurs, the two locations differ, *the answer is "no"*.
- Otherwise, *the answer is "some of the times"*.

For yes–no question answering about time:

Was fluent f true before event e occurred?

- If f is true for all time points less than or equal to t, *the answer is "yes"*.
- If f is false for all time points less than or equal to t, *the answer is "no"*.

It is also possible to deal with more complex questions whose answer is a phrase, such as "Where is the laptop?" Given an event or a fluent g whose ith argument is the one being asked, one can return an answer consisting of the conjunction of the ith arguments of all the events of fluents in the model that match g in all the arguments

except the ith. To answer the question about John's laptop, for example, the reasoner will return a conjunction of all the fluents of the form $at(laptop, L)$ that occur in the model of the theory.

20.8 Conclusion

To answer natural language questions posed with respect to natural language text, one either needs to develop a reasoning engine directly in natural language [52, 24, 41, 25] or needs a way to translate natural language to a formal language for which reasoning engines are available. While the first approach is commonly used for textual answering tasks such as in PASCAL [24] where the system needs to determine if a certain text H follows from a text T, at this point it is not developed enough to be used for answering the questions of the kind in Section 20.1.1. For questions of this kind there is an additional issue besides translating natural language to formal language; the need for commonsense knowledge, domain knowledge and specific reasoning modules. These are needed because often to answer a question with respect to a given text one needs to go beyond the text. The only exception is when the answer is a fact that is directly present or contradicted by the text.

In this paper we discussed two approaches to go from natural language to a formal representation. The first approach converts natural language to particular representations in classical logic. We discussed two such attempts: one does a syntactic parsing of the text, disambiguates the meaning of sentences using WordNet, creates a logic form, and uses a specialized reasoning engine; the second uses parsing but does not disambiguate, constructs first-order representations of knowledge and then uses first-order reasoning tools.

The second approach extracts relevant facts from the natural language. We discussed three such attempts: one that obtains relevant facts from the logic form mentioned earlier; the second that uses the semantic parser Link Grammar, the WordNet database and background knowledge to obtain relevant facts; and the third that uses an information extraction system to fill slots in templates.

In regards to background knowledge (domain knowledge plus commonsense knowledge) and specific reasoning modules, we illustrated their use in the DD QA system. In that system the knowledge representation language AnsProlog [32] is used for the most part. Recently, [63] also uses AnsProlog for natural language question answering. Mueller in [62] uses event calculus while LCC uses LLF and COGEX-based inference in their various QA systems. In this regard, one system that we did not cover so far is the CYC QA system. We are told that they use Link Grammar for understanding natural language and the CYC knowledge base [50, 23] for expressing domain knowledge. Since details of the CYC language, especially its semantics, are not available to us, we were not able to discuss the CYC system in more detail. However secondary sources such as [64] mention that the CYC system did not have axioms for reasoning about action and change, a very important component of commonsense reasoning. (It did have a rich ontology of actions and events.)

In the DD QA system and in general, by domain knowledge we refer to knowledge about specific topics such as the calendar, and world geography. By commonsense knowledge we refer to axioms such as the rule of inertia. By reasoning modules we refer to modules such as planning module, and reasoning about intentions module. The

DD QA system is a prototype and at present focuses only on a few types of domain knowledge, commonsense knowledge and reasoning modules.

To develop a broad QA system one needs a much larger background knowledge base than is in the DD system. In this regard CYC and its founders could be considered as pioneers. However by limiting its development to be within the company and by using a proprietary unvetted (outside CYC) language its usefulness to the general research community has become limited. This is despite CYC's effort to release ResearchCYC and other subsets of CYC. Thus what is needed is a community wide effort to build a knowledge repository that is open and to which anyone can contribute. To do that several sociological and technical issues still remain. Some of these issues are:

1. Which formal language(s) should be used by the community?

   While many are more comfortable with propositional and first-order logic, others prefer nonmonotonic logics that are more appropriate for knowledge representation. In this regard a recent development [51], whereby algorithms have been developed to translate theories in nonmonotonic knowledge representation languages such as AnsProlog and circumscriptive theories to propositional theories, is useful. It allows one to write knowledge in the more suitable and compact nonmonotonic logics, while the models can be enumerated using the efficient and ever improving propositional solvers.

2. How do we organize knowledge modules and how do we figure out which modules (say from among the travel module, calendar module, etc.) are needed to answer a particular question with respect to a particular text collection? For example in languages like JAVA there exists a large library of classes and methods. A programmer can include (i.e., reuse) these classes and methods in their program and needs to write much less code than if she had to write everything from scratch. Currently most knowledge bases outside CYC are written from scratch.

   A start in this regard has been made in the AAAI06 Spring Symposium on Knowledge repositories. It includes several papers on modular knowledge representation. We hope the community pursues this effort and similar to linguistic resources such as the WordNet [54, 26], FrameNet [27], the various large scale biological databases, and the large libraries of various programming languages, it develops an open knowledge base about everything in the world. A step in this direction would be to combine existing open source knowledge bases. Several of them are listed in http://www.cs.utexas.edu/users/mfkb/related.html.

3. If more than one logic needs to be used how do modules in different logics interact seamlessly?

   It seems to us that no single logic or formalization will be appropriate for different kinds of reasoning or for representing different kinds of knowledge. For example, while it is easier to express inertia axioms in AnsProlog, to deal with large numbers and constraints between them it is at present more efficient to use constraint logic programming. Thus there is a need to develop methodologies that would allow knowledge modules to be written in multiple logics and yet one will be able to use them together in a seamless manner. An initial attempt

in this direction, with respect to AnsProlog and Constraint logic programming is made in [13].

Finally, two other large research issues loom. First, to answer questions about calculating probabilities, one needs to be able to integrate probabilistic reasoning with logical reasoning without limiting the power and expressiveness of one or the other. Most existing approaches, except [12], limit the power of one or the other. Second, one needs to be able to develop ways to automatically learn some of the domain knowledge, commonsense knowledge and reasoning modules. While there has been some success in learning domain knowledge (and ontologies), learning commonsense knowledge and reasoning modules is still in its infancy.

Acknowledgements

We would like to thank Michael Gelfond, Richard Scherl, Luis Tari, Steve Maiorano, Jean-Michel Pomarede and Vladimir Lifschitz for their feedback on drafts of this paper. Section 20.5 was mostly written by Luis. The second reader Erik Mueller's comments were extremely insightful and improved the paper substantially. This research was supported by DTO contract ASU-06-C-0143 and NSF grant 0412000.

Bibliography

[1] The Language Computer Corporation Web Site, http://www.languagecomputer.com/.

[2] *Proceedings of the Third Message Understanding Conference (MUC-3)*. Morgan Kaufmann, 1991.

[3] *Proceedings of the Fourth Message Understanding Conference (MUC-4)*. Morgan Kaufmann, 1992.

[4] http://www.askjeeves.com, 1996.

[5] E. Akhmatova. Textual entailment resolution via atomic propositions. In *Proceedings of the PASCAL Challenges Workshop on Recognising Textual Entailment*, 2005.

[6] J. Allen. *Natural Language Understanding*. Benjamin Cummings, 1995.

[7] H. Alshawi, editor. *The Core Language Engine*. MIT Press, Cambridge, MA, 1992.

[8] R. Bar-Haim, I. Dagan, B. Dolan, L. Ferro, D. Giampiccolo, B. Magnini, and I. Szpektor. The second PASCAL recognising textual entailment challenge. In *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, Venice, Italy, 2006.

[9] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[10] C. Baral and M. Gelfond. Reasoning about intended actions. In *Proceedings of AAAI 05*, pages 689–694, 2005.

[11] C. Baral, M. Gelfond, G. Gelfond, and R. Scherl. Textual inference by combining multiple logic programming paradigms. In *AAAI'05 Workshop on Inference for Textual Question Answering*, 2005.

[12] C. Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. In *Proceedings of LPNMR-7*, pages 21–33, Jan 2004.
[13] S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In *Proc. of ICLP'05*, pages 52–66, 2005.
[14] P. Blackburn and J. Bos. *Representation and Inference for Natural Language. CSLI Studies in Computational Linguistics*. CSLI, 2005.
[15] J. Bos. Underspecification, resolution, and inference. *Logic, Language, and Information*, 12(2), 2004.
[16] J. Bos. Towards wide-coverage semantic interpretation. In *Proceedings of Sixth International Workshop on Computational Semantics (IWCS-6)*, pages 42–53, 2005.
[17] J. Bos and K. Markert. Recognising textual entailment with logical inference. In *Proceeding of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 628–635, 2005.
[18] M.E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.
[19] E. Charniak. Toward a model of children's story comprehension. Technical Report AITR-266, MIT, 1972.
[20] C. Clark, S. Harabagiu, S. Maiorano, and D. Moldovan. COGEX: A logic prover for question answering. In *Proc. of HLT-NAACL*, pages 87–93, 2003.
[21] C. Clark and D. Moldovan. Temporally relevant answer selection. In *Proceedings of the 2005 International Conference on Intelligence Analysis*, May 2005.
[22] P.R. Cohen and H.J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
[23] J. Curtis, G. Matthews, and D. Baxter. On the effective use of CYC in a question answering system. In *Proceedings of the IJCAI Workshop on Knowledge and Reasoning for Answering Questions*, 2005.
[24] I. Dagan, O. Glickman, and M. Magnini. The PASCAL recognizing textual entailment challenge. In *Proc. of the First PASCAL Challenge Workshop on Recognizing Textual Entailment*, pages 1–8, 2005.
[25] R. de Salvo Braz, R. Girju, V. Punyakanok, D. Roth, and M. Sammons. An inference model for semantic entailment in natural language. In *Proc. of AAAI*, pages 1043–1049, 2005.
[26] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
[27] C. Fillmore and B. Atkins. Towards a frame-based organization of the lexicon: The semantics of risk and its neighbors. In A. Lehrer and E. Kittay, editors. *Frames, Fields, and Contrast: New Essays in Semantics and Lexical Organization*, pages 75–102. Lawrence Erlbaum Associates, Hillsdale, 1992.
[28] A. Fowler, B. Hauser, D. Hodges, I. Niles, A. Novischi, and J. Stephan. Applying COGEX to recognize textual entailment. In *Proceedings of the PASCAL Challenges Workshop on Recognising Textual Entailment*, 2005.
[29] N.S. Friedland, P.G. Allen, M. Witbrock, G. Matthews, N. Salay, P. Miraglia, J. Angele, S. Staab, D.J. Israel, V. Chaudhri, B. Porter, K. Barker, and P. Clark. Towards a quantitative, platform-independent analysis of knowledge systems. In D. Dubois, C.A. Welty, and M.-A. Williams, editors. *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning*, pages 507–515. AAAI Press, Menlo Park, CA, 2004.

[30] T. Gaasterland, P. Godfrey, and J. Minker. Relaxation as a platform for cooperative answering. *Journal of Intelligent Information Systems*, 1(3–4):293–321, Dec 1992.
[31] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992.
[32] M. Gelfond. Answer set programming. In V. Lifschitz, F. van Hermelen, and B. Porter, editors. *Handbook of Knowledge Representation*. Elsevier, 2006.
[33] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors. *Logic Programming: Proc. of the Fifth Internat. Conf. and Symp.*, pages 1070–1080. MIT Press, 1988.
[34] M. Gelfond. Going places—notes on a modular development of knowledge about travel. In *AAAI Spring 2006 Symposium on Knowledge Repositories*, 2006.
[35] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.
[36] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.
[37] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2–4):301–321, 1993.
[38] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), 1998.
[39] B. Green, A. Wolf, C. Chomsky, and K. Laughery. BASEBALL: An automatic question answer. In *Computers and Thought*, pages 207–216. 1963.
[40] C. Green. The application of theorem proving to question-answering systems. PhD thesis, Stanford University, 1969.
[41] A. Haghighi, A. Ng, and C. Manning. Robust textual inference via graph matching. In *Proc. of HLT-EMNLP*, 2005.
[42] S. Harabagiu, G.A. Miller, and D. Moldovan. WordNet 2—A morphologically and semantically enhanced resource. In *Proceedings of SIGLEX-99*, pages 1–8, Jun 1999.
[43] S. Harabagiu and D. Moldovan. A parallel inference system. *IEEE Transactions on Parallel and Distributed Systems*:729–747, Aug 1998.
[44] J. Hobbs. Ontological promiscuity. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 61–69, Jul 1985.
[45] J. Hobbs. *The Logical Notation: Ontological Promiscuity*, 1985.
[46] H. Kamp and U. Reyle. *From Discourse to Logic, vols. 1, 2*. Kluwer, 1993.
[47] G. Katz, J. Pustejovsky, and F. Schilder, editors. *Annotating, Extracting and Reasoning about Time and Events*, 10–15 April 2005. *Dagstuhl Seminar Proceedings, Dagstuhl Seminar Proceedings*, vol. 05151, 2005.
[48] W. Kintsch. *Comprehension: A Paradigm for Cognition*. Cambridge University Press, 1998.
[49] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.
[50] D. Lenat and R. Guha. *Building Large Knowledge Base Systems*. Addison-Wesley, 1990.
[51] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.

[52] H. Liu and P. Singh. Commonsense reasoning in and over natural language. In M.Gh. Negoita, R.J. Howlett, and L.C. Jain, editors. *Knowledge-Based Intelligent Information and Engineering Systems*, *Lecture Notes in Computer Science*, vol. 3215, pages 293–306. Springer, Berlin, 2004.

[53] M. Maybury. *New Directions in Question Answering*. AAAI Press/MIT Press, 2004.

[54] G.A. Miller. WordNet: A lexical database for English. *Communications of the ACM*:39–41, 1995.

[55] R. Miller and M. Shanahan. Some alternative formulations of the event calculus. In A.C. Kakas and F. Sadri, editors. *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, vol. 2408, pages 452–490. Springer-Verlag, Berlin, 2002.

[56] A. Mohammed, D. Moldovan, and P. Parker. Senseval-3 logic forms: A system and possible improvements. In *Proceedings of Senseval-3: The Third International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*, pages 163–166, July 2004.

[57] D. Moldovan, S. Harabagiu, R. Girju, P. Morarescu, A. Novischi, F. Lacatusu, A. Badulescu, and O. Bolohan. Lcc tools for question answering. In E. Voorhees and L. Buckland, editors. *Proceedings of TREC 2002*, 2002.

[58] D. Moldovan and V. Rus. Transformation of WordNet glosses into logic forms. In *Proceedings of FLAIRS 2001 Conference*, May 2001.

[59] R. Montague. The proper treatment of quantification in ordinary English. In *Formal Philosophy: Selected Papers of Richard Montague*, pages 247–270, 1974.

[60] R. Moore. Problems in logical form. In *Proc. of 19th ACL*, pages 117–124, 1981.

[61] E. Mueller. Event calculus. In V. Lifschitz, F. van Hermelen, and B. Porter, editors. *Handbook of Knowledge Representation*. Elsevier, 2006.

[62] E.T. Mueller. Understanding script-based stories using commonsense reasoning. *Cognitive Systems Research*, 5(4):307–340, 2004.

[63] F. Nouioua and P. Nicolas. Using answer set programming in an inference-based approach to natural language semantics. In *Proc. of Inference in Computational Semantics (ICoS-5)*, Buxton, England, 20–21 April, 2006.

[64] A. Parmar. The representation of actions in KM and Cyc. Technical Report FRG-1, Department of Computer Science, Stanford University, Stanford, CA, 2001. http://www-formal.stanford.edu/aarati/techreports/action-reps-frg-techreport.ps.

[65] V. Rus. Logic forms for Wordnet glosses. PhD thesis, Southern Methodist University, May 2002.

[66] L. Schubert and F. Pelletier. From English to logic: Context free computation of conventional logical translation. *AJCL*, 1:165–176, 1982.

[67] S. Schulz. A comparison of different techniques for grounding near-propositional CNF formulae. In *Proceedings of the 15th International FLAIRS Conference*, pages 72–76, 2002.

[68] M. Shanahan. A circumscriptive calculus for events. *Artificial Intelligence*, 75(2), 1995.

[69] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Commonsense Law of Inertia*. MIT Press, 1997.

[70] D.D. Sleator and D. Temperley. Parsing English with a link grammar. In *Third International Workshop on Parsing Technologies*, 1993.

[71] L. Tari and C. Baral. Using AnsProlog with link grammar and WordNet for QA with deep reasoning. In *AAAI Spring Symposium Workshop on Inference for Textual Question Answering*, 2005.
[72] E. Voorhees. Overview of the TREC 2002 Question Answering Track. In *Proc. of the 11th Text Retrieval Evaluation Conference*. NIST Special Publication 500-251, 2002.
[73] W. Woods. Semantics and quantification in natural language question answering. In M. Yovitz, editor. *Advances in Computers*, vol. 17. Academic Press, 1978.
[74] M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.