A Knowledge-based System for Translating FOL Formulas into NL Sentences

Aikaterini Mpagouli, Ioannis Hatzilygeroudis

University of Patras, School of Engineering

Department of Computer Engineering & Informatics, 26500 Patras, Hellas

E-mail: {mpagouli, ihatz}@ceid.upatras.gr

Abstract In this paper, we present a system that translates first order logic (FOL) formulas into natural language (NL) sentences. The motivation comes from an intelligent tutoring system teaching logic as a knowledge representation language, where it is used as a means for feedback to the users. FOL to NL conversion is achieved by using a rule-based approach, where we exploit the pattern matching capabilities of rules. So, the system consists of a rule-based component and a lexicon. The rule-based unit implements the conversion process, which is based on a linguistic analysis of a FOL sentence, and the lexicon provides lexical and grammatical information that helps in producing the NL sentences. The whole system is implemented in Jess, a java-based expert system shell. The conversion process currently covers a restricted set of FOL formulas.

1 Introduction

To help teaching the course of "Artificial Intelligence" in our Department a web-based intelligent tutoring system has been created. One of the topics that it deals with is first-order logic (FOL) as a knowledge representation language. One of the issues in the topic is the translation of natural language (NL) sentences into FOL formulas. Given that this is a non-automated process [1, 2], it is difficult to give some hints to the students-users during their effort to translate an "unknown" (to the system) NL sentence into a FOL formula. However, some kind of help could be provided, if the system could translate the proposed by the student FOL formula into a NL sentence. We introduce here a method for converting/translating FOL formulas into NL sentences, called FOLtoNL algorithm. The structure of the paper is as follows. Section 2 deals with related work. Section 3 presents the basic algorithm and Section 4 refers to its implementation and lexicon. Section 5 concludes the paper.

2 Related Work

Our work can be considered as belonging to the field of Natural Language Generation [3], since it generates NL sentences from some source of information, which are FOL formulas. In the existing literature, we couldn't trace any directly similar effort, i.e. an effort to translate FOL sentences into natural language sentences. However, we traced a number of indirectly related efforts, those of translating some kind of natural language expressions into some kind of FOL ones.

In [4] an application of Natural Language Processing (NLP) is presented. It is an educational tool for translating Spanish text of certain types of sentences into FOL implemented in Prolog. This effort gave us a first inspiration about the form of the lexicon we use in our FOLtoNL system.

In [5], ACE (Attempto Controlled English), a structured subset of the English language, is presented. ACE has been designed to substitute for formal symbolisms, like FOL, in the input of some systems to make the input easier to understand and to be written by the users.

Finally, in [6], a Controlled English to Logic Translation system, called CELT, allows users to give sentences of a restricted English grammar as input. The system analyses those sentences and turns them into FOL. What is interesting about it is the use of a PhraseBank, a selection of phrases, to deal with the ambiguities of some frequently used words in English like have, do, make, take, give etc.

3 FOLtoNL Conversion Process

Our FOLtoNL conversion algorithm takes as input FOL formulas [1] of the following form (in a BNF notation, where '[ ]' denotes optional and '< >' nonterminal symbols): [<quant-expr>] [<stmt1> =>] <stmt2>, where <quant-expr> denotes the expression of quantifiers in the formula, '=>' denotes implication and <stmt1> and <stmt2> denote the *antecedent* and the *consequent* statements of the implication. These statements do not contain quantifiers. So, the input formula is in its Prenex Normal Form [1, 2]. Furthermore, <stmt1> and <stmt2> can not contain implications. Hence, the system currently focuses on the translation of simple FOL implications or FOL expressions that do not contain implications at all. For typing convenience, we use the following symbols in our FOL formulas: '~' (negation), '&' (conjunction), 'V' (disjunction), '=>' (implication), 'forall' (universal quantifier), 'exists' (existential quantifier).

The key idea of our conversion method, based on a FOL implication, is that when both the antecedent and the consequent statements exist, the consequent can give us the Basic Structure (BS) of that implication's NL translation. BS may contain variable symbols. In that case, the antecedent of the implication can help us to define the entities represented by those variable symbols. In other words, we can

find NL substitutes for those variables and then use them instead of variable symbols in BS to provide the final NL translation of the implication.

If the FOL expression does not contain variables, the translation is simpler: "if <ant-translation> then <con-translation>", where <ant-translation> and <con-translation> consist of appropriately combined interpretations of atoms and connectives. In case we have only <stmt2>, i.e. an expression without implications, we use the same method with one difference: variable NL substitutes emerge from the expression of quantifiers, since there is no antecedent. Of course, there is a special case in which we choose some atoms of the expression for the estimation of NL substitutes and the rest of them for the BS and we work as if we had an implication.

The basic steps of our algorithm are the following:

1. Scan the user input and determine <quant-expr>, <stmt1> and <stmt2>. Gather information for each variable (symbol, quantifier etc). Each atom represents a statement. Analyze each atom in the three parts of its corresponding statement: subject-part, verb-part and object-part.
2. If <stmt1> $\neq \varnothing$,
   - 2.1 Find the basic structure (BS) of the final sentence based on <stmt2>.
   - 2.2 For each variable symbol in BS specify the corresponding NL substitute based on <stmt1>. If there are no variables, then BS is in NL. In that case, find also the Antecedent Translation (AT) based on <stmt1>.
3. If <stmt1> = $\varnothing$,
   - 3.1 Find BS based on all or some of the atoms of <stmt2>.
   - 3.2 For each variable symbol in BS specify the corresponding NL substitute based on the information of quantifiers or particular atoms of <stmt2>. If there are no variables, then BS is build via all the atoms of <stmt2> and is in NL.
4. Substitute each variable symbol in BS for the corresponding NL substitute and give the resulting sentence as output. If there are no variables, distinguish two cases: If the initial FOL sentence was an implication then return: "If <AT> then <BS>". Otherwise, return BS.

In the sequel, we explain steps 2 and 3 of our algorithm, which are quite similar.

3.1 Finding the Basic Structure of the Final Sentence

In this subsection, steps 2.1 and 3.1 are analyzed. First of all, we find the atoms in <stmt2> that can aggregate, i.e. atoms that have the same subject-part and verb-part but different object-parts, or the same subject-part but different verb-parts and object-parts or different subject-parts but the same verb-part and the same object-part. Atoms that can aggregate are combined to form a new sentence which is called a *sub-sentence*. If an atom cannot be aggregated, then it becomes a sub-sentence itself. This process ends up with a set of sub-sentences, which cannot be

further aggregated and, when divided by commas, give BS. Let us consider the following input sentences as examples:

(i) (forall x) (exists y) human(x) & human(y) => loves(x,y)
(ii) (exists x) cat(x) & likes(Kate,x)
(iii) (forall x) (exists y) (exists z) dog(x) & master(y,x) & town(z) & lives(y,z) => lives(x,z) & loves(x,y)
(iv) (forall x) bat(x) => loves(x,dampness) & loves(x,darkness) & small(x) & lives(x,caves)
(v) (forall x) bird(x) & big(x) & swims(x) & ~flies(x) => penguin(x)

The basic structures produced for these input sentences are the following (note the aggregation in (iii) and (iv) and the exclusion of the atom 'cat(x)' from BS in (ii)):

(i) x loves y.
(ii) Kate likes x.
(iii) x lives in z and loves y.
(iv) x loves dampness and darkness and x is small and lives in caves.
(v) x is a penguin.

The next stage is the specification of NL substitutes for the variable symbols.

3.2 Finding Natural Language Substitutes for Variable Symbols

Natural language substitutes are specified based on <stmt1>, in case of an implication, or based on quantifiers and maybe some particular single-term atoms of <stmt2>, in case of an implication-less expression. The first step is to determine the primary NL substitutes for all variables. A primary NL substitute contains all the information provided for a variable symbol by single-term atoms. The second step is to enrich each variable's primary NL substitute with information about that variable's relation to other entities, via appropriate two-term atoms. This step is ignored in case of an implication-less expression. We use <name-of-x> to indicate the NL substitute of a variable with symbol x and <refer-x> as a kind of referring expression that we can use instead of <name-of-x> when the latter appears more than once in BS.

For each variable symbol, say x, the algorithm performs the first step as follows:

A-1. If there is only one atom P(x): Depending on the quantifier information we have <quant> = "every"/"some"/"not every"/"no". According to the type of P, the primary name of x is determined as follows: If P is a noun, then <name-of-x>="<quant><P>" and <refer-x>="that <P>". If P is an adjective, then <name-of-x>="<quant> <P> thing" and <refer-x>="that thing". If P is a verb, then <name-of-x>="<quant> thing that <P>" and <refer-x>="that thing".

A-2. If there are more than one atoms, say $P_i(x)$ (i=1,…,k): These atoms are divided into three categories according to the type of their predicate (ad-

jective, noun, verb). We denote by Pa, Pn and Pv predicates of type adjective, noun and verb respectively. The primary name will be of the form: "<quant> [<Pa_1>,<Pa_2>,…and <Pa_n>] thing/<Pn_1>,<Pn_2>,...and <Pn_m> [that <Pv_1>,<Pv_2>,…and <Pv_s>]" and <refer-x> will be of the form "that thing"/"that <Pn_1>, that <Pn_2>,…and that <Pn_m>".

A-3. If there is no atom P(x), then, according to the quantifier, <name-of-x> will be: "everything"/"something"/"not everything"/"nothing".

A-4. Special case: For each variable y, different from x, if there is no atom Pn(y), but there are atoms Pn_i(y, x) and maybe atoms Pa(y) or/and Pv(y), then we treat each of the atoms Pn_i(y, x) as an atom 'has(x,y)', after we have computed the primary name of y, <name-of-y>="a [<Pa_1>, <Pa_2>,…and <Pa_n>] <Pn_1>,<Pn_2>,...and <Pn_m> [that <Pv_1>,<Pv_2>,…and <Pv_s>]" and <refer-y>="that <Pn_1>, that <Pn_2>,…and that <Pn_m>". Hence, <name-of-x> becomes "<name-of-x> that has <name-of-y>".

The primary names of the variables for the example input sentences 1-5 are:

(i) <name-of-x> = "every human", <name-of-y> = "some human" (A-1)

(ii) <name-of-x> = "some cat" (A-1)

(iii) <name-of-x> = "every dog", <name-of-y> = "a master", <name-of-z> = "some town" (A-4)

(iv) <name-of-x> = "every bat" (A-1)

(v) <name-of-x> = "every big bird that swims and does not fly" (A-2)

The second step is the enrichment of primary names via two term atoms, to achieve the final NL substitutes. The enrichment takes place via the recursive call of the function "build-name" which is described below. The enrichment function, each time it is called, uses only the atoms that have not been used in previous calls. For each variable symbol, say x, in BS, following the order of occurrence, the following actions are performed by "build-name":

B-1. If there are no two-term atoms to be used for the enrichment of <name-of-x>: If x has not been referred to in the NL substitute of the previous variable symbol in BS, or x is BS's first variable, then the final NL substitute for x is the primary substitute computed in the previous step.

B-2. If there are two-term atoms for the enrichment of <name-of-x> and x has not been referred to in the NL substitute of the previous variable symbol in BS, he enrichment begins: If x is a subject in BS and appears only as a second term in two-term atoms, the corresponding atoms are transformed appropriately to treat x as a subject-part, either by using passive tense for verb predicates or by analyzing noun predicate atoms Pn(y,x) as "x has y as <Pn>". Then, for each atom having x as a first term we enrich <name-of-x> as follows: <name-of-x>="<name-of-x> [and] that <pred-translation> <build-name(y)>", where <pred-translation>="<Pv>"/ "is the <Pn> of"/ "is <Pa> than>", according to the predicate type.

B-3. If the current variable symbol x has a reference in the NL substitute of the previous one due to recursive calls of "build-name", we do not need to build its NL substitute again. We use, instead, <refer-x> determined earlier.

Via the substitution of variable symbols for NL substitutes, we get the next NL sentences as outputs for the previous five input sentences:

(i) Every human loves some human.
(ii) Kate likes some cat.
(iii) Every dog that has some master that lives in some town lives in that town and loves that master.
(iv) Every bat loves dampness and darkness and every bat is small and lives in caves.
(v) Every big bird that swims and does not fly is a penguin.

4 Implementation Aspects

The FOLtoNL process has been implemented in Jess [7]. Jess is a rule-based expert system shell written in Java, which however offers adequate general programming capabilities, such as definition and use of functions. The system includes two Jess modules, MAIN and LEX. Each Jess module has its own rule base and its own facts and can work independently from the rest of Jess modules. Focus is passed from one module to the other to execute its rules. MAIN is the basic module of the system, whereas LEX is the system's lexicon.

The lexicon consists of a large number of facts concerning words, called word-facts. Each word-fact is an instance of the following template: (word ?type ?gen ?form ?past ?exp ?stem ?lem), where 'word' declares that it is a fact describing the word ?lem and the rest are variables representing the fields that describe the word (part of speech, gender, number, special syntax, stem).

5 Conclusions and Discussion

In this paper, we present an approach for translating FOL formulas into NL sentences, called the FOLtoNL algorithm. The whole system is implemented in Jess and consists of a rule-based system that implements the conversion algorithm and a lexicon. Of course there are some restrictions that are challenges for further work. One problem is the interpretation of sentences which are entirely in the scope of a negation. Yet another constraint is forced upon the use of '=>', which can only occur once in the input sentence. Another restriction is that currently we do not take into consideration the order of quantifiers in the user input. Finally, the lexicon at the moment contains a limited number of words. It should be further extended. All the above problems constitute our next research goals, concerning our algorithm and system.

References

1. Genesereth MR, Nilsson NJ (1988) Logical foundations of AI. Morgan Kaufmann
2. Brachman RJ, Levesque HJ (2004) Knowledge representation and reasoning. Morgan Kaufmann
3. Reiter E, Dale R (2006) Building natural language generation systems. Cambridge University Press
4. Rodríguez Vázquez de Aldana E (1999) An application for translation of Spanish sentences into first order logic implemented in prolog. http://aracne.usal.es/congress/PDF/EmilioRodriguez.pdf
5. Fuchs NE, Schwertel U, Torge S (1999) Controlled natural language can replace first order logic. Proceedings 14th IEEE International Conference on Automated Software Engineering (ASE'99). 295-298. http://www.ifi.unizh.ch/groups/req/ftp/papers/ASE99.pdf
6. Pease A, Fellbaum C (2004) Language to logic translation with PhraseBank. Proceedings 2nd Global Conference (GWC'04). 187-192
7. Friedman Hill E (2003) Jess in action: rule-based systems in Java. Manning Publishing. 2003