Machine Comprehension of Text Using Combinatory Categorial Grammar and Answer Set Programs

Piotr Chabierski, Alessandra Russo, Mark Law, and **Krysia Broda**
Department of Computing, Imperial College London, SW7 2AZ
{piotr.chabierski13, a.russo, mark.law09, k.broda}@imperial.ac.uk

Abstract

We present an automated method for generating Answer Set Programs from narratives written in English and demonstrate how such a representation can be used to answer questions about text. The proposed approach relies on a transparent interface between the syntax and semantics of natural language provided by Combinatory Categorial Grammars to translate text into Answer Set Programs, hence creating a knowledge base that, together with background knowledge, can be queried.

Introduction

Machine comprehension of text is a long-term open problem in Artificial Intelligence and can be assessed by a machine's ability to answer questions about passages of text. One of the main challenges is the capacity to automatically process the text and capture the natural language semantics. Various approaches have been proposed in the literature, e.g. (Bos et al. 2004), in particular for translating natural language sentences into first-order logic sentences. More recently, (Baral, Dzifcak, and Son 2008) has followed a different direction. Motivated by the need of expressing the (default) semantics of normative statements, Baral et al. have provided a first-step towards an automated way for translating natural language statements into ASP programs, by making use of an intermediate representation, called λ-ASP, to construct the semantic representation of sentences from its constituents. The λ-ASP calculus relies on a combinatory categorial grammar (CCG) derivation (Steedman 2000) to represent the syntactic structure of a sentence. To the best of our knowledge, although effective in expressing normative statements, this approach is mainly confined to a specific dataset of sentences. Consequently, the λ-ASP representation is limited to the generation of domain-specific semantic representations, and, as also pointed out by the authors in (Baral, Dzifcak, and Son 2008), it does not support more complex linguistic constructs such as adverbs or conjunction.

This paper addresses these limitations by presenting a fully automated method for generating Answer Set Programs (ASP) from narratives written in English in a general and principled manner, demonstrating how such a representation can be used to answer questions about text. Similarly to (Baral, Dzifcak, and Son 2008), the proposed approach relies on a transparent interface between syntax and semantics of natural language offered by Combinatory Categorial Grammars to translate the text to ASP. The outcome of this translation creates a form of knowledge base that, together with background knowledge, can later be queried. However, the generation of ASP representations from text uses an intermediate representation, called λ-ASP* calculus, that differs from the λ-ASP calculus proposed in (Baral, Dzifcak, and Son 2008) in two ways. It is more general and can handle more advanced grammatical and linguistic constructions such as, among others, relativisation, control and raising. It uses a general-purpose ASP representation language designed with the objective of making it suitable for efficient automated learning of common sense knowledge relevant to a given narrative, by means of current state-of-the-art systems for learning ASP programs (Law, Russo, and Broda 2014). Due to space limitations, this latter feature is outside the scope of the paper. Our proposed fully automated mechanism is also applicable to the generation of ASP representation of a large class of questions, including polar questions and "wh-word" questions that require single word answers. The main contributions of this paper are therefore:

- a general-purpose ASP representation for narratives written in natural language
- a λ-ASP* calculus that covers complex linguistic phenomena such as coordination, relativisation, control and raising
- a wide-coverage algorithm capable of performing translation from natural language to ASP, and
- an automatic translation into ASP of questions requiring one-word answers (*what*, *where*, *who*, *which*).

The effectiveness of the approach is demonstrated by using a publicly available dataset (Weston et al. 2015) as well as an hand-crafted set of sentences, specifically designed to capture the various complex linguistic constructs supported by the approach.

Background

In what follows, we outline the two main background formalisms used in our approach, namely answer set programming and combinatory categorical grammars.

Answer Set Programming (ASP) is an expressive language for knowledge representation and reasoning, based on the stable model semantics (Gelfond and Lifschitz 1988) and rooted in the field of non-monotonic reasoning and logic programming (Lifschitz 2008). Within the scope of this paper, Answer Set Programs are restricted to normal logic programs. So, an ASP program is a set of normal rules of the form:

$$r: \quad \underbrace{h}_{head(r)} \leftarrow \underbrace{b_1, b_2, \ldots, b_m, \text{not } b_{m+1}, \ldots \text{not } b_{m+k}}_{body(r)}$$

where $h, b_1, b_2, \ldots, b_m$ and not $b_{m+1}, \ldots$ not b_{m+k} are *positive* and *negated* atoms respectively with not denoting negation-as-failure. A fact is a rule whose body is empty ($m = k = 0$).

The Herbrand Base $HB(P)$ of an ASP program P is the set of all ground atoms constructed using predicate symbols, constants and function symbols in P. An Herbrand interpretation of P assigns a truth value to every atom in $HB(P)$ and is a model of P if for every ground rule r in P such that $body(r)$ is satisfied $head(r)$ is true. An Herbrand model M of P is minimal if no proper subset of M is a model of P. Given an ASP program P and a set M of ground atoms, the *reduct* P^M of P is the logic program obtained from P by removing every:

- rule that has a literal not p in its body, where $p \in M$
- negated literal in the bodies of the remaining rules

M is an answer set of P if it coincides with the minimal Herbrand model of P^M.

Combinatory Categorial Grammar (CCG) is an efficiently parsable grammar that enables semantic analysis of natural language by providing a transparent interface between syntax and semantics. In CCG, words are assigned syntactic categories defining their syntactic behaviour. The set of CCG categories comprises of *atomic categories* and *complex categories*, recursively constructed from the atomic categories. Examples of atomic categories include: N (bare noun), NP (noun phrase) and S (sentence). Complex categories are of the forms X/Y and $X\backslash Y$ and define functors that, when applied to an argument of category Y, produce a result of category X. Directionality of the slash indicates whether the argument has to occur immediately to the left "/" or to the right "\" of the functor. For example, $(S\backslash NP)/NP$ is the category of a transitive verb *buy* in a sentence *Jack bought a car.*

CCG provides a surface-compositional interface between syntax and semantics in which there is one-to-one mapping between the rules of syntactic and semantic composition (Hockenmaier and Steedman 2007). A set of syntactic combinatory rules specifies how adjacent constituents are combined. The most basic rules are forward (FA) and backward (BA) application:

$$\text{FA } (>)\text{: } X/Y : f \quad Y : a \implies X : f(a)$$
$$\text{BA } (<)\text{: } Y : a \quad X\backslash Y : f \implies X : f(a)$$

Composition combinatory rules allow two functor categories to combine and produce another functor. Example of a composition rule is forward composition (FC):

$$\text{FC } (> \mathbf{B})\text{: } X/Y : f \;\; Y/Z : g \implies X/Z : \lambda x.f(g(x))$$

Type-raising combinatory rules allow an argument category Y to become a functor category $X/(X\backslash Y)$ or $X\backslash(X/Y)$. Forward type-raising rule (FT) is given by:

$$\text{FT } (> \mathbf{T})\text{: } Y : a \implies X/(X\backslash Y) : \lambda f.f(a)$$

Composition together with type-raising rules are used to handle coordination and non-local dependencies introduced, for example, by relative clauses, which is illustrated by the following syntactic derivation for a relative clause: *that Jack wanted*, from a sentence: *The car that Jack wanted was expensive.*

$$\dfrac{\dfrac{that}{(N\backslash N)/(S/NP)} \quad \dfrac{\dfrac{\dfrac{Jack}{NP}}{S/(S\backslash NP)} > \mathbf{T} \quad \dfrac{wanted}{(S\backslash NP)/NP}}{S\backslash NP} > \mathbf{B}}{N\backslash N} >$$

The predicate structure can be obtained directly from the syntactic derivation, provided that the semantics of lexical items, assigned by the lexicon, is known. Using a CCG syntactic derivation and first-order logic augmented with λ-calculus, for a sentence: *Jack buys cars* derivation of a first-order representation can be performed as follows:

$$\dfrac{\dfrac{Jack}{NP : jack} \quad \dfrac{\dfrac{buys}{(S\backslash NP)/NP : \lambda x.\lambda y.buy(y,x)} \quad \dfrac{cars}{NP : cars}}{(S\backslash NP) : \lambda y.buy(y, cars)} >}{S : buy(jack, cars)} <$$

Representing Natural Language in ASP

The signature of the ASP logical representation used in our approach, to express sentences written in English, consists of four classes of predicates: *nominals*, *events*, *modifiers* and *prepositions*. The name of each predicate is composed of an arity prefix, which indicates the number of arguments taken by the corresponding word, and a class suffix, which takes one of the four aforementioned values. For example, a transitive verb *buy* is represented by the predicate: $binaryEvent(e_1, buy, c_1, n_0)$ where the first argument e_1 serves as an identifier, *buy* is a lemma of the corresponding word and c_1, n_0 are arguments of the verb (agent and theme respectively). For the sake of conciseness, throughout this paper the arity prefixes are skipped and nominals, events, modifiers and prepositions are abbreviated in the predicate names as *nom*, *ev*, *mod* and *prep* respectively.

Preliminary Text Analysis

First, the input text is split into sentences and tokens. Subsequently, every word in the input text is tagged with the following set of annotations:

- **Lemma** - inflection-free form of the word, used as an argument of the corresponding predicate
- **Part-of-speech (POS) tag** - used by the lexicon to perform coarse division of words into classes
- **Named entity tag** - enriches the predicate structure and allows to treat multi-word proper names as single entities, e.g. *Eiffel Tower* as `eiffel_tower` rather than a nominal and a modifier
- **Coreference cluster** - groups mentions of the same entity in the text. Cluster identifiers are used to form identifiers of *nominal* predicates corresponding to entity mentions, hence embedding coreference information in the predicate structure
- **Semantic role** - derived for predicate (verb) arguments. PropBank sense IDs (Kingsbury and Palmer 2002) are used to order predicate arguments when generating predicate structure for verbs. For example, in a sentence: *[Jack]$_{ARG0}$ kicked [the ball]$_{ARG1}$* the transitive verb *kicked* is translated as: $ev(e_1, kick, c_0, n_0)$, were c_0 and n_0 are identifiers of *Jack* and *the ball* and they are ordered according to their sense IDs (0 and 1 respectively)

After annotating the text, a CCG parse tree is derived for each sentence. The CCG parse tree is a binary tree whose leaf nodes correspond to lexical items and internal nodes to phrases, clauses or sentences. Leaves are identified by `L` as the first argument in the node's descriptor, followed by the CCG category, POS tag and the word itself. Descriptors of the internal nodes begin with the identifier `T`, followed by the CCG category.

Listing 1: CCG parse tree for a sentence *A new and safe car that Jack wanted to buy was really expensive.*

```
<T S[dcl]>
  <T NP>
    <L (NP/N) DT A>
    <T N>
      <T N>
        <T (N/N)>
          <L (N/N) JJ new>
          <T ((N/N)\(N/N))>
            <L CONJ CC and>
            <L (N/N) JJ safe>
        <L N NN car>
      <T (N\N)>
        <L ((N\N)/(S[dcl]/NP)) WDT that>
        <T (S[dcl]/NP)>
          <T (S[x]/(S[x]\NP))>
            <T NP>
              <L N NNP Jack>
          <T ((S[dcl]\NP)/NP)>
            <L ((S\NP)/(S\NP)) RB really>
            <T ((S[dcl]\NP)/NP)>
              <L ((S[dcl]\NP)/(S[to]\NP)) VBD wanted>
              <T ((S[to]\NP)/NP)>
                <L ((S[to]\NP)/(S[b]\NP)) TO to>
                <L ((S[b]\NP)/NP) VB buy>
  <T (S[dcl]\NP)>
    <L ((S[dcl]\NP)/(S[adj]\NP)) VBD was>
    <L (S[adj]\NP) JJ expensive>
```

Splitting text into sentences, tokenisation, lemmatisation, POS tagging, named entity recognition and coreference resolution is performed using Stanford CoreNLP (Manning et al. 2014). For syntactic parsing and semantic role labeling EasySRL (Lewis, He, and Zettlemoyer 2015) is used.

λ-ASP* Calculus

λ-ASP* calculus serves as an intermediate representation between natural language and ASP. It follows the syntax of ASP but extends it with abstraction and application as known from λ-calculus, which enables compositional derivation of semantic representations of phrases and sentences from their constituents. λ-ASP* expressions have the following general form:

$$\underbrace{[h_1, \dots h_k]}_{k\ heads} \underbrace{\lambda v_1 \dots \lambda v_l}_{l\ abstractors} . \underbrace{p_1(a^1_1, \dots a^1_{r_1}), \dots p_m(a^m_1, \dots a^m_{r_m})}_{m\ predicates},$$
$$\underbrace{w_1@(a^{m+1}_1, \dots a^{m+1}_{r_{m+1}}), \dots w_n@(a^{m+n}_1, \dots a^{m+n}_{r_{m+n}})}_{n\ applications}$$

where v_i and w_j are bound variables, arguments a^x_y are bound variables or constants and p_t are predicate names included in the previously defined signature. Arities of predicates and applications are denoted by $r_1, \dots r_{m+n}$. Every predicate has a non-zero arity; however, applications can take no arguments. In case of applications, bound variables w_j are referred to as *function placeholders* and are replaced by other λ-ASP* expressions to which arguments $a^{m+j}_j, \dots a^{m+j}_{r_{m+j}}$ are applied. Predicates and applications constitute the *body* of a λ-ASP* expression. Every λ-ASP* expression has a non-empty set H of bound variables or constants, called expression *heads*, which corresponds to the linguistic head of a phrase and can have more than one element to account for coordination. As an example, let us consider a λ-ASP* expression for a transitive verb *buy*, given by:

$$[e_0]\lambda v_1.\lambda v_0.ev(e_0, buy, v_0, v_1), (v_1), (v_0)$$

Constant e_0 is an identifier of the verb *buy* and also a head of the expression. Bound variables v_0, v_1 can be replaced, via the process of application (described further in this section), by constants and variables – when the bound variable occurs as a predicate or application argument, or by other λ-ASP* expression – when the bound variable is a function placeholder in an application. The list of bound variables is followed by an *event* predicate and two applications with no arguments, which act as placeholders for the subject and object of the verb *buy*.

For λ-ASP* expression e, $preds(e)$ and $apps(e)$ denote the sets of all predicates and applications in e respectively and $abs(e)$ denotes the ordered list of bound variables. The set of heads of e is denoted by H_e. For $p \in preds(e)$, $args(p)$ denotes the ordered list of arguments of p; the same notation is used for applications. Given two λ-ASP* expressions e and f, the application $g = e@f$, where v denotes the first bound variable in e, is computed as follows:

- for every $a \in apps(e)$ of the form $w@(b_1, b_2, \ldots b_k)$, if $v = w$, compute recursively $f' = f@(b_1, b_2, \ldots b_k)$, which is equivalent to $(((f@b_1)@b_2)\ldots)@b_k$, include $preds(f')$ and $apps(f')$ in g, and set $f = f'$. If $v \in args(a)$, $|H_f|$ copies of a are included in g and v in the i$^{\text{th}}$ copy is replaced by the i$^{\text{th}}$ head of f. If $v \neq w$ and $v \notin args(a)$, unmodified a is included in g
- for every $p \in preds(e)$, if $v \in args(p)$, $|H_f|$ copies of p are included in g and v in the i$^{\text{th}}$ copy is replaced by the i$^{\text{th}}$ head of f. If $v \notin p$, p is included in g unmodified
- bound variables of g are given by: $abs(g) = (abs(e) - \{v\}) + abs(f)$, where '$-$' denotes removal of an element from a list and '$+$' denotes list concatenation
- if $v \in H_e$, $H_g = (H_e \backslash \{v\}) \cup H_f$. Otherwise, $H_g = H_e$

Constant c is represented by an expression e such that $preds(e) = apps(e) = \emptyset$, $abs(e) = [\,]$ and $H_e = \{c\}$. For variable v, expression e is modified so that $abs(e) = [v]$.

Lexicon

In our approach, lexicon is an *algorithm*, which given an input word together with the relevant annotations (CCG category, POS tag, lemma, coreference cluster) derives the corresponding λ-ASP* expression. λ-ASP* expressions for content words and some prepositions consist of a single predicate and a different number of bound variables and applications. Each word is assigned to one of five sub-lexicons (nominal, event, modifier, preposition, wh-word) based on the POS tag and CCG category, which in turn determines the class of the corresponding predicate. The arity of a predicate, for all cases other than adverbs and some function words, is equal to the number of arguments of the CCG category of the corresponding word.

Co-indexation is performed for the relevant CCG categories, such as the ones corresponding to raising and control verbs or relative pronouns. Our approach performs co-indexation for all categories listed in (Hockenmaier and Steedman 2005) and it is realised via application primitive of λ-ASP* expressions. For example, for a control verb *wanted* with a co-indexed CCG category: $(S[dcl]\backslash NP_i)/(S[to]\backslash NP_i)$ the λ-ASP* expression is: $\lambda v_1.\lambda v_2.ev(e_1, want, v_2, v_1), v_1@v_2$. Cases when co-indexation has to be performed are identified by pattern matching on CCG categories.

Application primitive is also used to translate coordinate sentences whose constituents have the same CCG category. For example, in case of an adjective phrase *new and safe* the λ-ASP* expression for coordinator *and* is: $[v_2, v_1]\lambda v_2.\lambda v_1.\lambda v_0.(v_2)@(v_0), (v_1)@(v_0)$, which is inferred from the CCG categories of the conjuncts, namely (N/N). In case of conjuncts that take more than one argument, such as transitive verbs *buy* and *sell* in the sentence: *Jack buys and sells oil*, the number of arguments of applications occurring in the λ-ASP* expression is set accordingly: $[v_3, v_2]\lambda v_3.\lambda v_2.\lambda v_1.\lambda v_0.(v_3)@(v_1, v_0), (v_2)@(v_1, v_0)$.

To ensure the same ordering of verb arguments for different grammatical constructions (for example passive vs. active voice) the arguments are ordered according to their PropBank semantic role labels.

Word	**λ-ASP Expression**
a	$[v_0]\lambda v_0.(v_0)@(n_0)$
new	$[v_0]\lambda v_0.mod(new, v_0), (v_0)$
and	$[v_2, v_1]\lambda v_2.\lambda v_1.\lambda v_0.(v_2)@(v_0), (v_1)@(v_0)$
safe	$[v_0]\lambda v_0.mod(safe, v_0), (v_0)$
car	$[v_0]\lambda v_0.nom(v_0, car), (v_0)$
that	$[v_0]\lambda v_1.\lambda v_0.(v_1)@(v_0)$
Jack	$[c_0]nom(c_0, jack)$
really	$[v_0]\lambda v_0.mod(really, v_0), (v_0)$
wanted	$[e_0]\lambda v_1.\lambda v_0.ev(e_0, want, v_0, v_1), (v_1)@(v_0)$
to	$[v_0]\lambda v_0.v_0$
buy	$[e_1]\lambda v_1.\lambda v_0.ev(e_1, buy, v_0, v_1), (v_1), (v_0)$
was	$[v_0]\lambda v_0.(v_0)$
expensive	$[expensive]\lambda v_0.mod(expensive, v_0), (v_0)$

Table 1: Lexicon entries for the lexical items from the example sentence.

Semantic Composition

The λ-ASP* expression for a given phrase or sentence is constructed bottom-up following the structure of the CCG parse tree. For each leaf node, a λ-ASP* expression is assigned by the lexicon. For internal nodes, the relevant CCG combinatory rule is identified, based on the categories of the child nodes and used to derive the λ-ASP* expression compositionally. All combinatory rules are realised using the λ-ASP* expression: $\lambda v_0.\lambda v_1.(v_1)@(v_0)$. In case of type-raising rules, child node's λ-ASP* expression is assigned to v_0. For application and composition rules child nodes' λ-ASP* expressions are identified as *function* and *argument* based on their CCG categories, the former is assigned to v_1 and the latter to v_0.

In our running example, the λ-ASP* expression for a noun phrase *new and safe car* is derived in three steps. Referring to Table 1, semantic representation for node in line 9 in Listing 1 is derived as follows:

$$\begin{aligned}&\{[v_2, v_1]\lambda v_2.\lambda v_1.\lambda v_0.(v_2)@(v_0), (v_1)@(v_0)\}@\{\\&\quad [v_3]\lambda v_3.mod(safe, v_3), (v_3)\}\\&\equiv [v_0, v_1]\lambda v_1.\lambda v_0.mod(safe, v_0), (v_0), (v_1)@(v_0)\end{aligned}$$

For the node in line 7, we get the following expression:

$$\begin{aligned}&\{[v_0, v_1]\lambda v_1.\lambda v_0.mod(safe, v_0), (v_0), (v_1)@(v_0)\}@\{\\&\quad [v_2]\lambda v_2.mod(new, v_2), (v_2)\}\\&\equiv [v_0]\lambda v_0.mod(safe, v_0), mod(new, v_0), (v_0)\end{aligned}$$

Finally, for the node in line 5, we get:

$$\begin{aligned}&\{[v_0]\lambda v_0.mod(safe, v_0), mod(new, v_0), (v_0)\}@\{\\&\quad [v_1]\lambda v_1.nom(v_1, car), (v_1)\}\\&\equiv [v_1]\lambda v_1.mod(safe, v_1), mod(new, v_1),\\&\qquad nom(v_1, car), (v_1)\end{aligned}$$

Translation of Questions

Our approach for generating ASP representations from text is also applicable to polar questions and questions that require one word answer, such as questions starting with *wh-words*: *what*, *where*, *who* and *which*.

For polar questions, the same translation approach, described so far, is used to derive the corresponding λ-ASP* expressions with the only differences being that (inflected) auxiliary verbs: *be*, *do* and *have* are discarded (translated as the identity λ-ASP* expression: $[v_0]\lambda v_0.v_0$) and constants in the body of the λ-ASP* expression which do not correspond to definite noun phrases are replaced with variables. The generated predicates of λ-ASP* expression form the body of an ASP query rule: $q \leftarrow body$. Two additional rules of the form $ans(yes) \leftarrow q$ and $ans(no) \leftarrow not\ q$ are included to capture the *yes* and *no* answers respectively. If all answer sets of the generated ASP program include the same ground instance of the fact *ans*, then the corresponding value (namely *yes* or *no*) is returned as an answer. Otherwise the answer `UNKNOWN` is returned.

In case of wh-questions requiring a one-word answer, a query predicate is added, whose purpose is to map the identifier to the actual word corresponding to the answer. A rule, similar as to the case of polar questions, is also included, but this time with an additional literal: $ans(W) \leftarrow nom(I, W), body$. In case the $ans(\cdot)$ predicate is not in any model of the program, `UNKNOWN` answer is returned.

λ-ASP* to ASP Conversion

A λ-ASP* representation, generated from a given text, may or may not contain bound variables. Hence, the conversion of a λ-ASP* into an ASP representations has to consider two cases.

In the first case, no bound variables are present. Hence, the resultant λ-ASP* expression consists exclusively of instantiated predicates - ones for which all predicate arguments are constants. Therefore, each predicate is converted directly to an ASP fact. The λ-ASP* expression derived for our running example falls under this category, hence the following ASP program is generated:

$$\{nom(c_1, jack).\ nom(n_0, car).\ mod(expensive, n_0).\ mod(safe, n_0).\ mod(new, n_0).\ mod(really, e_0).\ ev(e_0, want, c_1, e_1).\ ev(e_1, buy, c_1, n_0).\}$$

In the second case, the list of bound variables in the resultant λ-ASP* expression is non-empty, the expression is converted to a set of normal rules and a set of facts, which corresponds to universal and existential quantification of the given sentence. For each rule the head is the predicate whose identifier is equal to one of the heads of the λ-ASP* expression. The body of each rule is composed incrementally by including every predicate from the λ-ASP* expression (different from the head predicate) which has a variable among its arguments that is also an argument of either the head of the given rule or some other predicate already included in the body. A set of facts, corresponding to the existential quantification, is also derived by applying Skolem constants to the λ-ASP* expression. As an example, let us consider a sentence: *Jack and Jim enjoy opera*, with the λ-ASP* expression given by:

$$[e_0]\lambda v_1.nom(c_1, jack), nom(c_2, jim), nom(v_1, opera), ev(e_0, enjoy, c_1, v_1), ev(e_0, enjoy, c_2, v_1)$$

Using the previously described procedure, the λ-ASP* expression is converted to the following ASP program:

$$\{ev(e_0, enjoy, c_1, X) \leftarrow nom(X, opera).\ ev(e_0, enjoy, c_2, X) \leftarrow nom(X, opera).\ nom(c_1, jack).\ nom(c_2, jim).\ nom(f_1, opera).\ ev(e_0, enjoy, c_1, f_1).\ ev(e_0, enjoy, c_2, f_1).\}$$

The first two rules of the above program can be intuitively read as: *Jack (Jim) enjoys everything that is an opera*. The last two facts state that: *there exists an opera that Jack and Jim enjoy* and they allow us to answer questions such as: *What does Jack (Jim) enjoy?*

Exceptions and Default Persistence

Natural language expressions are often associated with implicit information that does not follow directly from their literal meaning and requires background knowledge to be interpreted correctly. To associate words in the text with their interpretations, a *semantic* predicate is introduced for each predicate in the chosen signature. For example, for the binary *event* predicate, the semantic predicate is defined as: $\{sem\_ev(E, L, X, Y) \leftarrow ev(E, L, X, Y), \text{not } ab\_ev(E, L, X, Y)\}$. Definitions of semantic predicates specify an exception structure. Consider for instance the implicit negation introduced by the verb *forget*, like in the sentence: *Jim forgot to take an umbrella*. This implicit negation can be captured by the following rule: $\{ab\_ev(E_0, L, X, Y) \leftarrow sem\_ev(E_1, forget, X, E_0), ev(E_0, L, X, Y)\}$. When applied to the above example sentence, the rule has the following grounding: $\{ab\_ev(e_0, take, c_1, n_1) \leftarrow sem\_ev(e_1, forget, c_1, e_0), ev(e_0, take, c_1, n_1)\}$, where c_1 and n_1 correspond to *Jim* and *umbrella* respectively.

Understanding and keeping track of persistence of fluents over time plays an important role in text comprehension. In our approach we assume a linear time structure in the narratives. Verbs occurring in the text are associated with implicit time points which are represented in the generated ASP program by ground instances of the predicate $time(T, E)$. The argument T is a positive number equal to the position of the verb, identified by E, in the text relative to the other verbs. Persistence is captured using a set of rules motivated by Event Calculus (Kowalski and Sergot 1986), and formalised as follows:

$$\begin{aligned} sem\_ev(E, L, X, Y) &\leftarrow fluent(T, L, X, Y), time(T, E) \\ fluent(T, L, X, Y) &\leftarrow initiate(E, L, X, Y), time(T, E) \\ fluent(T_1, L, X, Y) &\leftarrow fluent(T_2, L, X, Y), \\ &\quad previous(T_2, T_1), time(T_1, E), \\ &\quad \text{not } terminate(E, L, X, Y) \end{aligned}$$

The truth value of the predicate $fluent(T, L, X, Y)$ persists and can change over timepoints. It is parametrised by the timepoint T, lemma L of the corresponding word, which is the name of the fluent, and arguments of the word. For example, $fluent(2, be, c_1, n_2)$, where c_1 and n_2 are identifiers of *Jim* and *kitchen* means that *Jim is in the kitchen* at time 2 and he will continue to be there until termination. Words that are treated as fluents are specified by providing initiation and termination rules in the background knowledge.

Discussion

The dataset used for evaluating our approach consists of 22 sentences and short stories, 10 of which were hand-crafted, 5 were taken from (Baral and Dzifcak 2012), 5 from online news articles and 2 from *STEP 2008 shared tasks* dataset (Bos 2008a). Regarding the hand-crafted examples, correct representations were generated for 9 out of 10. The remaining 12 examples consist of 18 sentences and a correct predicate structure was generated for 14 of them and for the other 4 the main source of errors was the CCG parser. Among the 14 sentences, the representations were fully correct for 9 of them and for the other 5 some predicates were incorrectly parametrised due to errors in coreference resolution, interpretation of noun phrases (distributive vs collective), semantics of prepositions and lack of support for expletive sentences and different types of elliptical constructions. The results of the evaluation confirm that our approach can correctly represent sentences with coordination and non-local dependencies, use coreference information to resolve personal and possessive pronouns and capture the semantics of different parts of speech.

The question answering functionality of the system was evaluated on *The (20) QA bAbI tasks* (Weston et al. 2015) that we also used for learning common sense knowledge using Inductive Logic Programming, which is however outside the scope of this paper. Our system was evaluated on 10 tasks (tasks number: 1, 2, 5, 6, 8, 9, 12, 15, 16, 18) which, among others, required correct representation of conjunction and generic sentences as well as temporal and default reasoning to be answered correctly. Using general (non task-specific) background knowledge our system achieved 100.0% accuracy for 9 tasks and 93.6% for the other task, which was task 16. The lower accuracy on that task was due to its requirement for task specific background knowledge.

Related Work

The two approaches most related to out work are the Boxer system (Bos et al. 2004) and the λ-ASP calculus described in (Baral, Dzifcak, and Son 2008). The Boxer system relies on C&C syntactic parser, which uses CCG and Discourse Representation Theory (DRT) to generate first-order logic representations from texts written in English. The system achieves more than 95% coverage of English newspaper texts (Bos 2008b) and the generated logical representations are used as an input to a theorem prover for recognising textual entailment (Bos and Markert 2005). The Boxer system uses λ-DRS (Discourse Representation Structure) formalism to derive semantic representation compositionally. The first step of the translation algorithm is assignment of λ-DRS expressions to the leaf nodes of the parse tree generated for the sentence by the C&C parser. The lexicon is derived by manually specifying λ-DRS expressions for the majority of the 409 CCG categories used by the C&C parser. Then, the combinatory rules, expressed in terms of λ-DRS expressions are used to derive the semantics for the internal nodes in a bottom-up manner as dictated by the structure of the parse tree. The method for constructing semantic representations used by the system is a general-purpose one and can be applied to formalisms other than DRT. Finally, the λ-DRS is translated into first-order logic. Our approach uses instead a different formalism, namely ASP rather than first-order logic, which makes it arguably easier to perform inferences necessary in the task of question answering. With regards to the translation algorithm, both our approach and the Boxer system generate the lexicon based on a set of hand-crafted rules which rely on the CCG category, part-of-speech tag, and lemma of the given word.

The λ-ASP calculus described in (Baral, Dzifcak, and Son 2008) is used to translate English sentences with normatives and exceptions into ASP. λ-ASP expressions, similarly λ-DRS used by Boxer, extend the underlying formalism with application and abstraction as known from λ-calculus. Then, λ-ASP expressions are used to build semantic representation of phrases and sentences compositionally. The λ-ASP expressions, as presented in (Baral, Dzifcak, and Son 2008), is constrained to a small subset of English consisting of nouns, verbs and adjectives and does not support, among others, adverbs, prepositions and conjunctions. In the subsequent work (Baral et al. 2011), a method for automatic generation of lexicon entries from training data specified as pairs (S_i, L_i) where S_i is the sentence and L_i the corresponding logical representation has been proposed. Although the approach achieves favorable results on Geoquery database querying dataset and on a dataset of logical puzzles (Baral and Dzifcak 2012), the generated logical representations are still domain-specific and the proposed translation algorithm does not handle binding and linguistic phenomena such as coreference or non-local dependencies. In comparison, our approach offers a systematic treatment of adverbs, coordination and non-local dependencies, which to the best of our knowledge, (Baral, Dzifcak, and Son 2008) is not capable of. Different from λ-ASP, our approach does not use a domain-specific logical representation, bur rather a wide-coverage semantic parser, similarly to the Boxer system.

Conclusion

In summary, we have presented a novel approach for automatically generating ASP representation from text and questions with single word answers, which makes use of the CCG's transparent interface between syntax and semantics of natural language. To support a wide-coverage semantic parsing of text, we have generalised the existing formalism of λ-ASP calculus into a general-purpose λ-ASP* calculus capable of handling additional complex linguistic constructs. We have also shown how additional common-sense background knowledge about persistence and exceptions can be represented and related with the generated ASP representation of the text in order to support the answers to questions that are not directly expressed in text. We are currently exploring the integration into our approach of current state-of-the-art systems for learning ASP programs (Law, Russo, and Broda 2016) in order to automatically learn, instead of hand-crafting, relevant common sense knowledge from examples of question-answer pairs. Our preliminary results on this line of current work have been very promising.

References

Baral, C., and Dzifcak, J. 2012. Solving puzzles described in English by automated translation to answer set programming and learning how to do that translation. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*, 573–577. AAAI Press.

Baral, C.; Dzifcak, J.; Gonzalez, M. A.; and Zhou, J. 2011. Using inverse lambda and generalization to translate English to formal languages. In *Proceedings of the Ninth International Conference on Computational Semantics*, IWCS '11, 35–44. Stroudsburg, PA, USA: Association for Computational Linguistics.

Baral, C.; Dzifcak, J.; and Son, T. C. 2008. Using Answer Set Programming and lambda calculus to characterize natural language sentences with normatives and exceptions. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 2 of *AAAI '08*, 818–823. AAAI Press.

Bos, J., and Markert, K. 2005. Recognising textual entailment with logical inference. In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, HLT '05, 628–635. Stroudsburg, PA, USA: Association for Computational Linguistics.

Bos, J.; Clark, S.; Steedman, M.; Curran, J. R.; and Hockenmaier, J. 2004. Wide-coverage semantic representations from a CCG parser. In *Proceedings of the 20th International Conference on Computational Linguistics*, COLING '04. Stroudsburg, PA, USA: Association for Computational Linguistics.

Bos, J. 2008a. Introduction to the shared task on comparing semantic representations. In *Proceedings of the 2008 Conference on Semantics in Text Processing*, STEP '08, 257–261. Stroudsburg, PA, USA: Association for Computational Linguistics.

Bos, J. 2008b. Wide-coverage semantic analysis with Boxer. In *Proceedings of the 2008 Conference on Semantics in Text Processing*, STEP '08, 277–286. Stroudsburg, PA, USA: Association for Computational Linguistics.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the 5th International Conference*, 1070–1080. MIT Press.

Hockenmaier, J., and Steedman, M. 2005. CCGbank: User's manual. Technical report, Department of Computer and Information Science, University of Pennsylvania.

Hockenmaier, J., and Steedman, M. 2007. CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Comput. Linguist.* 33(3):355–396.

Kingsbury, P., and Palmer, M. 2002. From TreeBank to PropBank. In *Third International Conference on Language Resources and Evaluation*, LREC-02.

Kowalski, R., and Sergot, M. 1986. A logic-based calculus of events. *New Generation Computing* 4(1):67–95.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive learning of answer set programs. In *European Workshop on Logics in Artificial Intelligence*, 311–325. Springer.

Law, M.; Russo, A.; and Broda, K. 2016. Iterative learning of answer set programs from context dependent examples. *arXiv preprint arXiv:1608.01946*.

Lewis, M.; He, L.; and Zettlemoyer, L. 2015. Joint A* CCG parsing and semantic role labelling. In *Empirical Methods in Natural Language Processing*.

Lifschitz, V. 2008. What is Answer Set Programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI '08, 1594–1597. AAAI Press.

Manning, C. D.; Surdeanu, M.; Bauer, J.; Finkel, J.; Bethard, S. J.; and McClosky, D. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, 55–60.

Steedman, M. 2000. *The Syntactic Process*. Cambridge, MA, USA: MIT Press.

Weston, J.; Bordes, A.; Chopra, S.; Rush, A. M.; van Merriënboer, B.; Joulin, A.; and Mikolov, T. 2015. Towards AI-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*.