Natural Language Understanding with Knowledge

Jiquan Ngiam
May 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

(condensed rough draft)

Senior Thesis Advisor
Scott Fahlman (sef@cs.cmu.edu)

Abstract

This paper examines the problem of extracting structure knowledge from unstructured free text. The extraction process is modeled after construction grammars, essentially providing a means of putting together form and meaning. The knowledge base is not simply treated as a destination, but also an important partner in the extraction process. In particular, the ideas are implemented as a module closely tied to the Scone Knowledge Base system. I demonstrate that with a reasonable knowledge base and general construction rules, one can easily extract structured knowledge.

Keywords: natural language processing, construction grammar, knowledge representation

1 Introduction

The ultimate goal of processing natural language in computer is really to be able to form an understanding of dialogue and then reason over what has been understood. This paper aims to examine how we can move closer to this goal. The knowledge representation adopted is the Scone Knowledge Base (KB) system. This is in contrast the to customary representations used for knowledge – first order logic or lambda calculus. The Scone KB system is a flexible efficient system that models knowledge in a semantic network.

I approach natural language understanding from the viewpoint that people are able to understand completely ungrammatical sentences (e.g. "john, mary lunch burritos"). Essentially, understanding natural language comes *mostly* from semantics. Hence, any approach to natural language understanding must first consider how the knowledge is going to be represented.

Construction grammar is a family of grammars that emphasizes the connection between syntax and semantics. They provide a linguistic basis for the approach adopted by this paper. In particular, the notion of constructions adopted most closely follows that of Radical Construction Grammar.

These ideas have been implemented in LISP as a practical language-processing engine meant to be used in conjunction with the Scone KB system.

This paper also explores generalizing and learning over constructions.

2 Scone Knowledge Base System

The Scone KB system represents knowledge as a semantic network. In this paper, Scone elements are referred to when encased in curly brackets. For example, {physical object} refers to the Scone element that represents a typical physical object. An element can also have roles and relations specified over it. Furthermore, elements also have children or multiple parents and the roles and relations will be inherited accordingly. Scone efficiently supports multiple inheritance and exception.

3 Constructions

3.1 Overview

Constructions are basic pairings of form and meaning. In theory, form can include phonological specifications, syntactical relations, etc. However, a simplified version of constructions is adopted in this paper, in which form is limited to a pattern, which is defined by simply word order. This follows the ideas in Radical Construction Grammar.

Semantics are represented by simply the Scone KB structure that is associated with the construction. For practical purposes, I also allow arbitrary LISP code to be part of the semantics.

These following sections describe constructions as they have been implemented in the actual language engine.

3.2 Variables and Element Restrictions

In a construction, variables form bridge between form and meaning. Variables are normally associated with an element-restriction. This restriction can be specified by primitive elements (e.g. strings, numbers) or internal Scone elements.

3.3 Patterns

A pattern is essentially an ordered list of variables (or element-restrictions).

Tokens form the input to the language engine. A token is ideally an element within the KB – this can be achieved by pre-processing the input stream. However, tokens can also be strings, which are then resolved by the language-processing engine. A series of tokens is said to match a construction when each element-restriction in the construction's pattern matches the tokens, in the same order.

3.4 Triggers

Triggers provide constructions a means of communication with other high level processing system (e.g. a dialogue understanding system). They also serve to allow constructions to return multiple values.

One trigger that is common across all constructions is the *head* trigger that specifies what is the representative element of the construction after it has been instantiated.

3.5 Instantiation

If a stream of tokens matches a construction, then one can choose to instantiate the construction to effectively produce the scone-network structure associated with the construction. Furthermore, the construction will also return an element (as specified by the head trigger), which in essence refers to whole entity produced by the construction. For example, if "green elephant" were matched, then the construction would return an elephant instance with having color green.

This implies that the structure produced by one construction can be in place of a token to satisfy an element-restriction in another construction. This allows for a stream of tokens to be matched to some composition of constructions.

In effect, constructions resemble Context-Free Grammar rules, with the specifications involving Scone elements. Further, each construction is paired with a semantic representation.

3.6 Example Constructions

The following are some example constructions:

Location-Action Construction	
Variables	(?x {location}) (?y {action})
Pattern	"In" (?x {location}) (?y {action})
Semantics	?x is-the {location} of ?y

Person-Action-Object Construction	
Variables	(?x {person}) (?y {action}) (?z {physical object})
Pattern	(?x {person}) (?y {action}) (?z {physical object})
Semantics	?w = a new instance of action ?y ?x is-the {agent} of ?w ?z is-the {object} of ?w

Since the *person-action-object* construction produces *?w* which is an action, it can be used to match part of the pattern for the *location-action* construction.

4 Language Processing Engine

4.1 Matching Engine

The core of the language-processing engine is based on the Earley parser. Interpretation of a series of tokens is performed in three phases – a pre-processing phase, a core matching phase and finally instantiation.

In the pre-processing phase, each variable of the pattern in each construction is linked to all other constructions that can produce some Scone structure that can satisfy the variable's restriction. This sets up constructions in a way that represents a context-free grammar.

The core of the matching engine is largely based on the Earley parser. In the matching phase, the interpreter runs left to right accumulating only constructions that are valid up till that token. In order to keep the space/time requirements manageable, the partially matched constructions are scored, ranked and pruned.

The matching phase will conclude with all possible matches. From these, the best construction is selected based on its rank and instantiated. This happens in a recursive fashion depending on the composition of the final match.

4.2 Scoring and Selection

A weighting of various sub-scores performs scoring of constructions. The sub-scores for a construction include

1. Length of the construction
2. Completion (matched) status of the construction
3. Base score for the construction
4. How well the underlying elements match the tokens

4.3 Practicalities

One of the major objectives was to provide an engine that was practical and applicable to future projects that wish to process unstructured text together with Scone. Many options have been implemented to this extent to make the system extremely flexible and easy to use.

For example, one would find that the same construction could often apply when matching to both individual variables and a set formed by individuals of that type. This is handled as a special case in the construction engine and a variable option can be set to enable it.

4.4 Performance

A performance evaluation of the engine will be included here. All demonstrations with the code take no more than a second for each sentence now.

In general, the runtime of the engine can be bounded by the scoring mechanisms. One can easily limit the number of constructions allowed. The Earley parser on its own has a complexity of $O(c \cdot n^3)$ where n is the length of the input and c is the number of constructions. Since n is normally 15–20 in general English sentences, one can expect very reasonable performance with the engine. (I note that this complexity bound might not hold anymore with some of the changes to the engine introduced.)

5 Context-Dependent Processing

5.1 Handling Sets

We often need to form sets based on the text. For example, a recipe might indicate for one to mix the tuna, mayonnaise, celery and salt. After forming the set, one might wish to characterize the typical member of the set, so that it can be used as part of another construction. More generally speaking, we need to characterize the roles and relations of

elements based on context as well. This is a similar issue to general constructions described below.

5.2 General and Specific Constructions

Unfortunately, not all constructions can be linked to other constructions during pre-processing. For example, in a very general construction that matches {color} {physical object}, it is not possible to use this construction in another construction that expects a {kitchen appliance} unless the production itself produces something of type {kitchen appliance}. Hence, the actual context of the text being processed matters.

As a result, we categorize linked constructions in two different categories – those that can be determined before processing and those that need to be checked in context. In order to keep the number of constructions processed to a manageable size, the constructions for the latter category are kept to those that produce elements which are super-sets of the expected type. This does not cover all the cases but it is hypothesized that they are sufficient for most purposes.

During processing, each construction is then linked to some representative element (pre-existing in the KB). This element is used for checking the context.

A true solution to this problem would be to actually instantiate constructions during processing and use those instantiations as representative elements. These instantiations are later removed when processing completes. This is however not implemented.

6 Generalization

6.1 Overview

Generalization for constructions was motivated by how learning of language develops. Children often learn that certain patterns of language translate into the same meaning and then generalize over these patterns. An example of this generalization happens with past-tense inflexions.

In the context of this paper, generalization serves as a means of supervised learning.

6.2 Generalizing over a Set

In this paper, I also explore an algorithm for generalization over a set that uses spreading activation. The aim of generalization of a set is to provide a representative element(s) of the set that serves as a means of determining if a new element should be part of this set or not. The general algorithm places an amount of activation on each element in the set and proceeds to pass this activation to its parent nodes via a transfer function. The algorithm is analogous to a voting system.

6.3 Generalizing Constructions

From generalization of a set, one can also experiment with generalizing constructions. I propose an algorithm for generalizing constructions and explore the limitations and performance of the algorithm. At this stage, the generalization algorithm has been implemented as a proof-of-concept.

7 Opportunistic Processing

This will be direction taken for the paper for the rest of the semester. I will be exploring boundary conditions and what can be done with constructions do not fully match. Ideas involved at this level processing to create placeholder elements that can be later filled in when more information is available later.

8 Discussion

One might be motivated to ask how this approach differs from the CFG parsers that have been widely used for syntactical parsing. Syntactical parsing with CFG parsers have operated by first tagging words in sentences according to some basic types and then building a parse tree over several rules. The approach here does not perform any tagging, but instead uses elements from a KB in place of them. This allows for a more general form of representation (the elements in the KB can be tagged) and further, allows for establishing a link to semantics.

References

TBA

Appendix A – User Manual

To use the engine, one would want to first load up Scone into the LISP environment, and then load "c-engine-loader.lisp"

The following are the macros/functions that will be of concerned at the user level:

(define-construction …)
(cp-init …)
(cp-interpret …)

(define-construction
(name parent var-list pattern trigger-list &key base-score)
&body scone-ops)

This macro provides a facility for defining constructions. A construction is essentially a Scone element. The **name** argument specifies the English name to use for the construction, while the **parent** argument specifies the parent construction (if any).

The **var-list** argument specifies a list of variables associated with the construction. Each element of this list is another list in the following form **(var-name option-1 option-2 …)**. The first element of this list should always be a **var-name** (e.g. V1, V2). This name is used to refer to the same variable later in the construction (or in children constructions).

Each **option** either specifies a variable restriction or a variable option. Valid options include **instance**, **string**, **inst-code, rel, role** and **match-set**. If the **instance** option is used, when the match is made, the element that the rest of the code operates on will be a proper instance of the match. If the **string** option is used, the restriction type for the variable is set to be a string match. The **inst-code** option specifies what instantiation code to use for the variable. This is used in cases where the variable is not part of the pattern to be matched, but instead something newly created. Variable restrictions are either **strings**, **Scone elements** or **predicate functions**. If a string is used as a restriction, but the string option is not enabled, then the function will attempt to resolve the string to a Scone element. The **rel** and **role** options provide a means of specifying restrictions based on the relations and roles of the elements. The **match-set** option makes is possible for the restriction to match a {set} of elements such that the typical member matches the variable restriction. *[Options that will be available soon include: proper-indv, optional]*

The **pattern** argument specifies the pattern to match to. This should be a list of **var-names**, where the var-names either come from a parent construction, or is specified in the var-list.

The **trigger-list** argument is again a list of lists. Each sub-list is a trigger and option pair. A mandatory trigger for all constructions to specify is the ***c-head*** trigger; the option for this trigger indicates which variable should be used as head (or production) of the construction. It is also possible to specify ***c-exp-eval*** as the option for this trigger. This

will cause the head variable that is returned to be return value of the last expression in **scone-ops**.

The optional **base-score** key argument assigns a base-score to the construction. This score should be in a range between 0–1. The base-score defaults to 0.0

The **scone-ops** argument specifies the semantics of the construction. It essentially should specify a list of **Scone operations** that will be evaluated when the construction is instantiated. Note that this list of operations will be evaluated upon the definition of the construction as well. If there is any code that should only be evaluated upon instantiation, one can wrap it with (r-eval …)

(cp-init)

This is the construction processor initialization function. This function performs the pre-processing steps to link up the constructions together so that matching later would be faster. You only need to call this function once before doing the interpretations.

(cp-interpret tokens-or-string &optional start-constructions)

This is the main interpretation function that takes in a list of **tokens** or a **string**. If a string is passed into **tokens-or-string**, it will be separated into parts using space as a delimiter. Optionally, one can also specify **start-constructions** that determine what the overall constructions the entire list of tokens is expected to match to. If start-constructions are not specified, then all possible constructions will be considered.

This function returns 3 values – the head element produced by a matching construction, the trigger table and the variable table associated with that construction. The trigger table is a mapping of triggers to options. The variable table maps each **var-name** of the matched construction to a specific **Scone element** or **string**.

(cp-set-target-constructions target-construction)

This serves as an alternative to using the start-constructions parameter in cp-interpret. This basically specifies the default construction type to use. Effectively, it expects a Scone element and downscans from it to determine the constructions to match for.

Appendix B – Recipe Example

The following examples on recipes require the core and cooking knowledge bases to be loaded.

Example Construction:

```
(define-construction

  ;; Name
  "a and b - forms a set"

  ;; Parent
  nil

  ;; Variable List
  ((v1 {edible} instance) (v2 {edible} instance)
   (v3 "and" "or" string)
   (v4 {set} (inst-code (new-indv nil {set}))))

  ;; Pattern List
  (v1 v3 v2)

  ;; Trigger List
  ((*c-head* v4))

  ;; Scone Operations
  (new-is-a (get-the-x-role-of-y {member} v4) {edible})
  (x-is-a-y-of-z v1 {member} v4)
  (x-is-a-y-of-z v2 {member} v4))
```

In this construction **"a and b – forms a set"** is the English name of the construction. There are 4 variables in the construction – **v1** thru **v4**. **v1** and **v2** are used to match to two edible elements, while **v3** is used to match to either the string **"and"** or **"or"**. **v4** is not used in the pattern but is produced by the code, the **inst-code** option of **v4** has been set to produce it. The **head** of the construction is **v4**, and there are 3 Scone operations which follow, essentially saying that the set produced represents a set of edible objects and that **v1** and **v2** are part of this set.

The list of constructions used is available in the appendix. Note that the representation adopted for the actions are simplified so as to focus on the use of the construction engine.

The following is the original text of a recipe:

Chocolate Chip Butter Cookies

Melt butter in a microwave or double boiler; stir in vanilla. Cool completely. In a large bowl, combine flour and sugar; stir in butter mixture and chocolate chips (mixture will be crumbly). Shape into 1-in. balls. Place 2 in. apart on ungreased baking sheets; flatten slightly. Bake at 375 degrees F for 12 minutes or until edges begin to brown. Cool on wire racks.

The constructions defined (see appendix) allows for the following sentences to be matched as expected.

```
(cp-interpret "Melt butter in a microwave or double-boiler")
(cp-interpret "stir in vanilla")
(cp-interpret "Cool completely")
(cp-interpret "In a large bowl , combine flour and sugar")
(cp-interpret "Stir in butter mixture and chocolate chips")
(cp-interpret "Shape into 1 in. balls")
(cp-interpret "flatten slightly")
(cp-interpret "Cool on wire racks")
(cp-interpret "Place on ungreased baking sheets")
(cp-interpret "Place 2 in. apart on ungreased baking sheets")
(cp-interpret "Bake at 375 degrees F")
```

The constructions also generalized to various other sentences in other recipes.

```
(cp-interpret "In a bowl , mix together tuna , celery , mayonnaise and salt")
(cp-interpret "Mix into cheese mixture")
(cp-interpret "Combine the chocolate chips and cream in a medium metal bowl")
(cp-interpret "Stir pecan halves into the chocolate")
```

However, there were many other sentences that could not successfully processed as well. This was mainly due to other patterns of representations that were not yet captured by the constructions or the simple representation used here.