Coreference in Knowledge Editing

Kees van Deemter and **Richard Power**
Information Technology Research Institute (ITRI)
University of Brighton, Lewes Road
Brighton, BN2 4GJ, UK
Kees.van.Deemter,Richard.Power@itri.brighton.ac.uk

Abstract

This paper briefly outlines the WYSIWYM (What You See is What You Meant) approach to knowledge editing and focuses on the role of coreferring Noun Phrases in the *feedback texts* that are generated by a WYSIWYM system and which play a key role in this approach. The paper pays special attention to the operations that a user of a WYSIWYM system can perform on feedback texts that contain coreferring Noun Phrases and to how they can be supported.

1 Introduction

1.1 Editing knowledge by WYSIWYM

WYSIWYM is a method of editing knowledge bases in which the user interacts with a *feedback text* generated by a natural language generation system (Power at al. 1997). The idea of WYSIWYM editing is to provide an interface to a knowledge base that is easy to use by an author who is a domain expert but not necessarily an expert in knowledge representation. The feedback text serves two purposes: it shows the knowledge that has already been specified; and it shows the options for adding new knowledge. These options are marked by *anchors*. By clicking on an anchor, the user obtains a pop-up menu from which a semantic option can be selected. The acronym WYSIWYM stands for 'What You See Is What You Meant': a natural language text ('what you see') presents a knowledge base that the author has built by purely semantic decisions ('what you meant').

This method was first implemented in DRAFTER-II (Power et al, 1998), a re-engineered version of DRAFTER (Paris et al, 1995) which is designed for use by the technical authors who produce software documentation. By interacting with the feedback text, the author defines a procedure for performing a task, e.g. the task of saving a document in a word processor.

When a new knowledge base is created, DRAFTER-II assumes that its root will be some kind of procedure. A procedure instance is created, and assigned an identifier (for internal use only), e.g. `proc1`. The definition of the concept `procedure` specifies that every procedure has two attributes: a goal, and a method. The goal must be some kind of action, and the method must be a list of actions. This information is conveyed to the author through a feedback text

> Achieve **this goal** by applying *this method*.

with several special features.

- Undefined attributes are shown through anchors marked by the use of boldface or italics.
- A **boldface** anchor indicates that the attribute is obligatory: its value must be specified. An *italicized* anchor indicates that the attribute is optional.
- All anchors are mouse-sensitive. By clicking on an anchor, the author obtains a pop-up menu listing the permissible values of the attribute; by selecting one of these options, the author updates the knowledge base.

Although the anchors may be tackled in any order, we will assume that the author proceeds from left to right. Clicking on **this goal** yields a pop-up menu that lists all the types of actions that the system knows about:

choose click ..... save schedule

(to save space, some options are omitted), from which the author selects 'save'. Although apparently selecting a word, the author is really selecting an option for editing the knowledge base. The program responds by creating a new instance, of type `save`, and adding it to the knowledge base as the value of the `goal` attribute on `proc1`:

```
procedure(proc1).
goal(proc1, save1).
save(save1).
```

From the updated knowledge base, the generator produces a new feedback text

> Save **this data** by applying *this method.*

including an anchor representing the undefined `actee` attribute on the `save1` instance. Note that this text has been completely regenerated. It was not produced from the previous text merely by replacing the anchor **this goal** by a longer string. By continuing to make choices at anchors, the author might expand the knowledge base in the following sequence:

- Save the document by applying *this method.*
- Save the document by performing **this action** (*further actions*).
- Save the document by clicking on **this object** (*further actions*).
- Save the document by clicking on the button with **this label** (*further actions*).
- Save the document by clicking on the Save button (*further actions*).

At this point the knowledge base is potentially complete (no boldface anchors remain), so an *output text* can be generated and incorporated into the software manual. For example,

> To save the document, click on the Save button.

To delete information, the author opens a pop-up menu on any span of the feedback text that presents a defined attribute. For instance, the span 'the document' presents the `actee` attribute on the instance `save1`. Clicking on this span in the feedback text, the author obtains the menu

Cut Copy

If 'Cut' is selected, the instance that is currently the value of the `actee` attribute is removed to a buffer, leaving the attribute undefined. The resulting feedback text introduces an anchor in place of 'the document'.

> Save **this data** by clicking on the Save button (*further actions*).

When the buffer is full, the pop-up menus that open on anchors contain a 'Paste' option if the instance in the buffer is a suitable value for the relevant attribute.

1.2 Limitations of DRAFTER-II

In the DRAFTER-II implementation of WYISYWM, the attribute value that is added at an anchor is always a *new* instance of the specified concept, never an existing instance. Suppose the author has developed the feedback text

> Save the document by entering the name of **this object** (*further actions*).

while aiming at the output text

> To save the document, enter its name and click on the Save button.

The current (incomplete) state of the knowledge base is

```
procedure(proc1).
goal(proc1, save1).
  save(save1).
  actee(save1, doc1).
    document(doc1).
method(proc1, list1).
  list(list1).
  first(list1, enter1).
    enter(enter1).
    actee(enter1, name1).
      name(name1).
```

The author now expands the anchor **this object**, which marks the `owner` attribute on `name1`. This can be done in two ways: opening a pop-up menu on the anchor and choosing 'document', or choosing 'Copy' on the earlier phrase 'the document' and pasting the copied material on to the anchor. In either case, the current implementation creates a new instance `doc2` of the concept `document` instead of using the existing instance `doc1`. In other words, it adds the two assertions

```
owner(name1, doc2).
document(doc2).
```

instead of the single assertion

```
owner(name1, doc1).
```

Our aim in this paper is to outline a way of overcoming this limitation, so that the author will be able to specify whether two similar descriptions are coreferential.

2 Wysiwym + indices

How can a feedback text indicate coreference? One possibility is to let feedback texts make use of anaphoric Noun Phrases, such as 'it' or 'this document'. In this paper we will assume that although *output* texts, which are meant to be read by a user who is not a domain expert, are expressions of some natural language, *feedback* texts may sometimes include artificial elements if this is necessary to avoid ambiguity. We will explore how the feedback language may be extended by the use of referential *indices*: If two NPs have the same subscript, they must have the same referent, whereas if they have different indices, they may or may not have the same referent. Brackets can be added to disambiguate the scope of indices in complex nominals. For example, the new feedback language can contain such expressions as

(the name of (the document)$_3$)$_1$ and
(the date of (the document)$_3$)$_2$,

where the indices imply that two documents involved in the two expressions are the same. In the remainder of this section, we will explore in what different ways an author may want to control the indices in the feedback text when she *inserts* a new Noun Phrase into the feedback text, or when she *cuts* or *copies* an expression in the feedback text.

2.1 Which options for editing

2.1.1 Inserting

Imagine the author wants to expand an anchor of the form 'this object', specifying that it is a document. Imagine, furthermore, that two other documents have already appeared in the feedback text, namely (document)$_1$ and (document)$_2$. Then the newly introduced object may corefer with (document)$_1$ or with (document)$_2$ or with neither. In order to avoid asking potentially superfluous questions, the system can first present the user with a menu containing the two options

Earlier-mentioned object? New object?

Only in case of the first choice will the system follow up with the menu

(document)$_1$? (document)$_2$?

In case of the second choice, the system will replace the anchor by some expression of the form (document)$_i$, where i is an index that has no earlier appearance in the feedback text.

2.1.2 Cutting

In the current version of DRAFTER, cutting is a conceptually simple operation that does not allow variations: an attribute is replaced by its anchor. When coreference enters the picture, this is no longer the case. In particular, there are two questions that we have to address when an indexed NP, say NP$_i$, is cut. Firstly, *Does the author intend to cut this occurrence of NP$_i$ alone, or does she intend to cut all occurrences of NP$_i$ that appear in the feedback text?* And secondly, *if* the author intends to cut all occurrences of NP$_i$, then *Does the author intend the system to respect the indices?* In other words, does she assume that the anchors that will result when NP$_i$ is de-selected must all be filled by NPs that have the same index? Depending on how these questions are answered, three different variants of the Cut operation arise. Let α_i be the NP on which the author has clicked. Then

- *Cut-one* only affects this occurrence of α_i.
- *Cut-all* affects all occurrences of α_i, also severing all coreference links between their anchors.
- *Cut-all^c* affects all occurrences of α_i, respecting all coreference links between their anchors.

Consider the example in Section 1.2, adapted to the situation where `doc1` is the value of two different attributes:

```
procedure(proc1).
  goal(proc1, save1).
  save(save1).
    actee(save1, doc1).
    document(doc1).
  method(proc1, list1).
  list(list1).
    first(list1, enter1).
    enter(enter1).
      actee(enter1, name1).
      name(name1).
      owner(name1, doc1).
```

To spell out the effect of the new operations, it will be convenient to use variables. If *Cut-one* is applied to the occurrence of `doc1` in the `owner` attribute then the only effect on the content of the knowledge base is that the `owner` of `name1` is undefined, which may be represented by a variable, say x. If, instead, *Cut-all* is applied to the same instance of `doc1`, then both the `owner` of `name1` and the `actee` of `save1` are undefined, which may be represented by using two different variables, say x and y. If, finally, *Cut-all*^{c} is applied then, once more, both are undefined, but their values must be equal. This may be represented by using the same variable, say x, in both cases: `actee(save1, x)`, `owner(name1, x)`.

2.1.3 Copying

Like cutting, copying allows variations now that the knowledge base can use one and the same instance (e.g., `doc1`) as the value of different attributes (e.g. `actee(save1, doc1)` and `owner(name1, doc1)`)). There are at least two options. One option amounts to a faithful copy of an instance. This option can be implemented exactly as in DRAFTER-II, namely by letting a buffer point to the instance that is copied. The other option (called Replicate) replicates the original feedback text, while renumbering all the subscripts in such a way that all the subscripts in the replica are new, respecting equalities between subscripts in the original. The second option arises when an author wants to reuse a part of the feedback text to refer to a new instance. Both editing operations are relatively straightforward to understand and implement.

2.2 Example

Imagine an author wanting to create the procedure given in Section 2.1.2, starting out with the incomplete representation presented in Section 1.2, where it is not yet specified what `name1` is the name of. The content of the knowledge base can be reflected by the feedback text

> Save document$_i$ by entering the name
> of **this object** (*further actions*),

where i is an arbitrary number. Suppose the author opens a pop-up menu on the anchor ‘this object’ and chooses the option ‘Earlier-mentioned object’. Note that, due to some obvious type constraints, the only instance in the existing knowledge base that can take the place of ‘this object’ is the object `doc1`. Consequently, there is no need for the system to ask further questions.

A slightly more round-about way in which the same potentially complete knowledge base can come about is as follows. Confronted with the just-presented menu, she decides to choose ‘New object’. The system then adds the statements

```
owner(name1, doc-j).
document(doc-j).
```

to the knowledge base, where `doc-j` is a constant that has not occurred before. The following feedback text may be generated:

> Save document$_i$ by entering the name
> of document$_j$ (*further actions*).

Seeing her mistake, the author can then click on either ‘document$_i$’ or ‘document$_j$’ and select Cut. This will cause the NP in question to be replaced by the anchor ‘this object’, which brings the author back to a situation where she can decide that the two documents should corefer after all.

3 Some problems

3.1 Limitations of the use of indices

Indices can be used to indicate, for two text spans, whether or not they are assumed, by the author, to refer to the same thing. This implies a number of problems familiar from the problem of tagging text corpora (e.g. Hirschman et al. 1997). Thus, it is unclear how indices should be used in relation to NPs occurring in intensional contexts. For example, it seems that the NPs in the following feedback text should receive different indices, even though their extension is equal:

> Make sure that (the date of (the program)$_i$)$_j$ equals (the current date)$_k$.

Furthermore, it can be tempting to view an NP as anaphoric, even if it is impossible to name any one text span as its antecedent. For example, this happens when the role of an antecedent is played by a combination of several NPs (e.g., the NP 'a file' and the NP 'a program' can together form the antecedent of the plural pronoun 'they') or when the antecedent is implied, rather than directly mentioned, by the text (e.g., in the case of bridging anaphora). In these cases, it is difficult to see how indices can be used. Note, however, that the problematic phenomena mentioned here may not form a serious problem in the context of the present work, since we are dealing with the construction of *feedback* texts (as opposed, for example, to the *output* texts generated by the system) which tend to make a simplified use of language. (cf. Section 2).

3.2 Embedded NPs

Suppose that the author has reached the following feedback text

> Save (the document)$_1$ by entering (the name of (the document)$_1$)$_2$ and entering (the name of (the document)$_1$)$_2$ (*further actions*).

with the intention of changing the second occurrence of 'document' into 'directory'. To avoid removing both occurrences of 'the document' she chooses the *Cut-one* option rather than *Cut-all*.

> Save (the document)$_1$ by entering (the name of (the document)$_1$)$_2$ and entering (the name of **this object**)$_2$ (*further actions*).

Now she clicks on the anchor **this object** and chooses 'directory'. What effect should this operation have on the noun phrase in which the anchor is embedded? There are two possibilities. Either the index of the embedding NP should remain the same:

> (the name of (the directory)$_3$)$_2$

or it should change, implying that the two names are distinct:

> (the name of (the directory)$_3$)$_4$

In this example, it seems clear that a name can belong only to one object, so that the latter result should be preferred. However, in some cases the index on the embedding phrase might plausibly remain the same after the embedded phrase has been changed. For instance, if we copy 'the menu containing the Save option', then cut 'the Save option' from the second occurrence of the phrase, replacing it by 'the Print option', there is no reason why the Save and Print options should not belong to the same menu. Thus to react appropriately to the Cut-one operation, the system may have to apply domain knowledge or seek guidance from the author.

4 Conclusion

We have outlined how the DRAFTER-II implementation can be adapted in such a way that it allows authors to specify coreference relations between objects. An experimental version of these ideas has been implemented successfully in the DRAFTER-3 system. In particular, no particular problems with the speed of the implementation have been encountered. The resulting offers the user the possibility of having coreference indicated either by indices, or by (colour-based) highlighting, or both. (Note that a Noun Phrase can be coreferential with other Noun Phrases anywhere in the text.) Given the purpose of feedback texts, it is essential that the feedback mechanism will remain as transparent as possible. For example,

1. The number of menus, and the number of options within each menu, should not become too large to handle. When the number of candidate-antecedents for a newly-introduced NP becomes too large, we will have to use alternative ways to let the author choose. (For example, authors may be allowed to do a string-based search.)

2. It is crucial that feedback texts avoid the use of concepts and expressions that may not be comprehensible to the domain expert using the WYSIWYM interface. This implies that the comprehensibility of the notion of coreference, as well as the way in which coreference relations are indicated, have to be investigated experimentally.

5 References

Heim 1982. Heim, I. *The Semantic of Definite and Indefinite Noun Phrases.* Ph.D. thesis, Univ. of Massachusets, Amherst, Mass.

Hirschman et al. 1997. Hirschman, L.,Robinson, P., Burger, J., and Vilain, M., Automating Coreference: The Role of Annotated Training Data. Proc. AAAI 1997.

Paris et al 1995. Paris, C., Vander Linden, K., Fischer, M., Hartley, A., Pemberton, L., Power, R. and Scott, D. 'A Support Tool for Writing Multilingual Instructions,' Proc. IJCAI-95, 1398-1404, Montreal.

Power et al. 1997. Power, R., Scott, D., and Evans, R. 'What You See Is What You Meant: direct knowledge editing with natural language feedback'. ITRI Technical Report No. 97-03, University of Brighton. Available from `http://www.itri.brighton.ac.uk/techreports/`.