Co-reference resolution

Yohan Chalier
Symbolic Natural Language Processing (SD213)
Télécom ParisTech

June 22, 2018

1 Theoretical approach

Co-reference occurs when two or more expressions refers to the same referent. Usually, one expression is in a full form (the *antecedent*) and the other one in a abbreviated form (a *proform*). For example:

(1) The music$_1$ was so loud that it$_1$ couldn't be enjoyed.

Note that referring expressions are indexed with the same integer.

Co-reference resolution is needed to derive a correct interpretation of a text. Most algorithms simply look for the nearest preceding individual that is compatible with the referring expression. This compatibility comes from features unification, on features like gender or number. Such algorithm may correctly solve sentences like:

(2) The girl$_1$ likes her$_1$ brother$_2$ and protects him$_2$.

However, it will fail at differentiating sentences such as:

(3) He$_1$ said that John$_2$ was coming.
(4) His$_1$ sister said that John$_1$ was coming.

In (3), *He* cannot refer to *John*, while in (4), such restriction do not apply. To thoroughly describe this phenomenon, one may use the following concepts of domination and c-command.

Definition 1.1 (Parse tree)**.** An ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.

Definition 1.2 (Domination)**.** Node N_1 dominates node N_2 if N_1 is above N_2 in the parse tree and one can trace a path from N_1 to N_2 moving only downwards in the tree (never upwards).

Definition 1.3 (c-command)**.** Node N_1 c-commands node N_2 if N_1 does not dominate N_2, N_2 does not dominate N_1 and the first (i.e. the lowest) branching node that dominates N_1 also dominates N_2.

For example, in the tree showed in Figure 1, M does not c-commands any other node since it dominates all of them, whereas A c-commands B and all its children.

Theorem 1.1. *One restriction between proform and antecedent is that the proform cannot appear in a position where it c-commands its antecedent.*

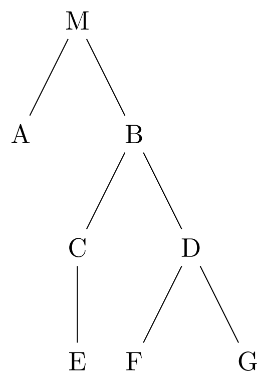


Figure 1: Dummy tree

This theorem explains the difference there is between *"He said"* and *"His sister said"*. As one can see in Figure 2, the node *He* c-commands the node *John*, since they do not dominate each other and the first branching node that dominates *He* is *S* (*NP* has only one child, it is not a branching node), and it dominates *John*.
On the other hand, as shown in Figure 3, the first branching node that dominates *His* is *NP*, and it does not dominate *John*, so *His* does not c-command its antecedent. They can be associated together.

Finding a proof of Theorem 1.1 is not trivial, and actually it has not been found at this day. This condition about co-reference binding is just hypothesized. Some works try to interpret this with cognitive aspects [1]. The main issue with this theorem is that the computational approach presents the result in a way it cannot be explained. Therefore people try to find other reasons to explain this phenomenon.

One hypothesis to this condition is based on the syntax derivation process. In linguistic, to determine if a string of words is a grammatical (well-formed) sentence of a language, one can try to derive its structure from a set of lexical items, operations such as a Merge or a Move, and some constraints [2].
When a pronoun is found and needs to be bound to an antecedent, the binding is possible across only one merge. Beyond that, relation between nodes is considered too weak to be linked together.

Another hypothesis, based on cognitive motivations, is the *Novelty condition*:

> An anaphorically dependent element cannot have more determinate reference than its antecedent. [3]

This generalizes the c-command and seems to more a more cognitive explanation as it also handles external anaphorically relationships such as in (5) or (6):

(5) The doctor$_1$ walks in. The man$_1$ keeps quiet.
*(6) *The man$_1$ walks in. The doctor$_1$ keeps quiet.*

Though, it may not handle all cases. And implementing this condition requires the computation of the determination of a reference, which is not trivial either.

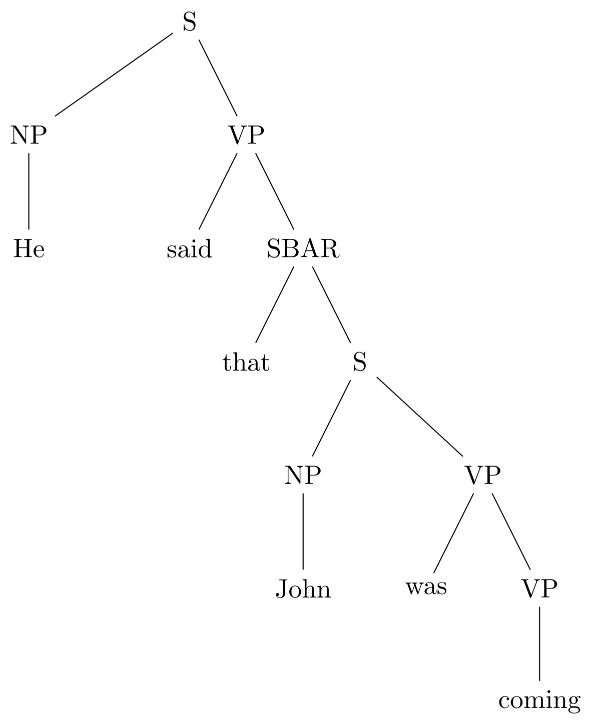


Figure 2: Parse tree of *"He said that John was coming."*

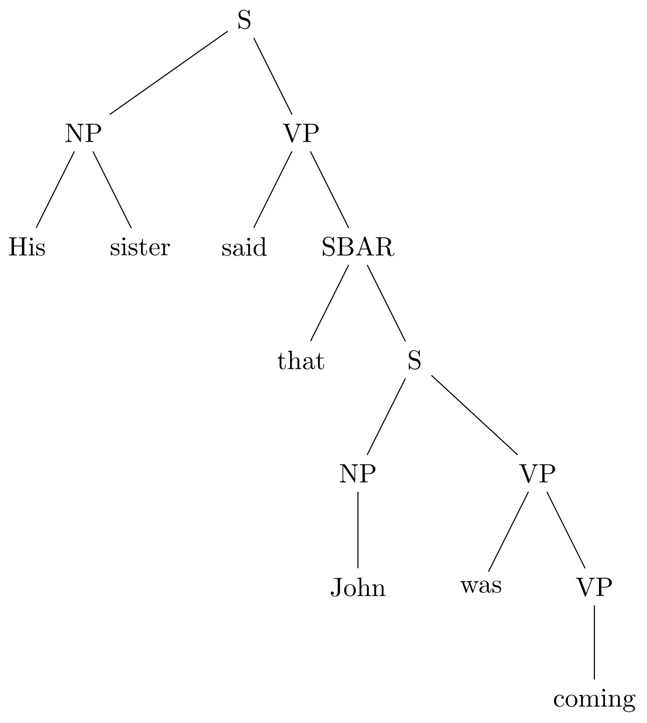


Figure 3: Parse tree of *"His sister said that John was coming."*

2 Implementation

The goal here is an implementation of a parser that solves co-reference, and rewrites the given sentence with indexes to identify the referencing groups. The final version of the Python script I wrote is available on GitHub [4].

Prolog struggles My first idea was to re-use the CYK parser from the lab work on parsing. Though, inserting new pieces of code within the already done parser turned out more difficult than expected. I tried to take inspiration from the lab work on semantics processing with a predicate `refers(,)`. In the end, I was spending more time on debugging than actually working on co-reference. Therefore, I pivoted to Python.

2.1 Grammar and lexicon

I used English grammar and vocabulary. I followed the same pattern as in the lab works. Here is my grammar file:

```
s --> np, vp
np --> n
np --> det, n
np --> pn, n
np --> pn
vp --> v, np
vp --> v, sub
sub --> p, n, v
vp --> vp, p, vp
```

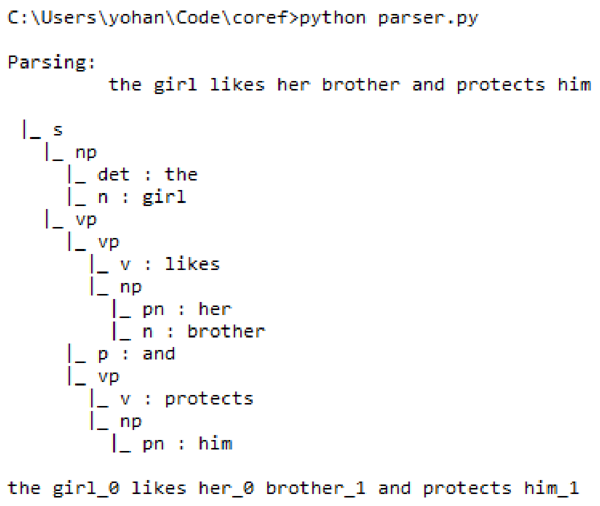
And here is my lexicon:

```
n --> boy[gender:male;num:sing],
      sister[gender:female;num:sing],
      John[gender:male;num:sing],
      girl[gender:female;num:sing],
      brother[gender:male]
det --> the
pn --> his[gender:male;num:sing],
       her[gender:female;num:sing],
       he[gender:male], him[gender:male]
v --> likes, protects, said, was_coming
p --> and, that
```

I used only two features, gender and number. Two functions, `read_grammar` and `read_lexicon`, process those files and produces Python dictionaries. Keys are the left part of the rules, values are a lists of the split right part. Features are also stored as a dictionary.

2.2 Parser

I implemented my own version of a simple top-down parser. Everything is achieved within a class `Parser`. The method `parse_aux` iterates through the



```
C:\Users\yohan\Code\coref>python parser.py

Parsing:
         the girl likes her brother and protects him

 |_ s
   |_ np
     |_ det : the
     |_ n : girl
   |_ vp
     |_ vp
       |_ v : likes
       |_ np
         |_ pn : her
         |_ n : brother
     |_ p : and
     |_ vp
       |_ v : protects
       |_ np
         |_ pn : him

the girl_0 likes her_0 brother_1 and protects him_1
```

Figure 4: Screenshot of the outputted parse tree

grammar rules and stops when the end of the sentence is reached. It uses recursion to try to parse smaller portions of the sentence.

Along the way, it produces the parse tree as an instance of the class `Node`, which is a recursive data type. Nodes store a label and a list of references of its children. Other attributes are stored, such as a reference to the parent. The Figure 4 is an example of an outputted parse tree for the sentence (2).

2.3 Co-reference

In this script, I focused on the c-command condition and features agreement. The use of the class `Node` makes co-reference resolution rather easy. The computation of domination is actually a search function in a recursive structure, since we only need to check whether the dominated target is within the offspring of the dominant one, which is easy to compute:

```
def dominates(self, other):
    """ Returns True if 'self' dominates 'other' """

    if len(self.childs) == 0:  # bottom of the tree,
        return False           # 'other' was not found

    for child in self.childs:  # recursive iteration
        if child == other or child.dominates(other):
            return True
    return False
```

Then, the `c_commands` method climbs up the tree until a branching node is found, and then uses the domination to return the answer:

```
def c_commands(self, other):
    """ Returns True if 'self' c-commands 'other' """
```

```
# first we make sure no one dominates the other
if self.dominates(other) or other.dominates(self):
    return False

parent = self.parent
while parent is not None:  # tree climbing
    if len(parent.childs) > 1 and parent.dominates(self):
        return parent.dominates(other)
    parent = parent.parent
return False
```

Then, once the parse tree has been computed, for each word recognized as a referring expression (for now, only pronouns), a method of the `Parser` finds a compatible word (*features agreement*), and if the proform does not c-command that word, it links them both. Results are the following:

```
Parsing: he said that John was_coming
he_0 said that John was_coming

Parsing: his sister said that John was_coming
his_0 sister said that John_0 was_coming

Parsing: the girl likes her brother and protects him
the girl_0 likes her_0 brother_1 and protects him_1
```

3 Conclusion

Although the script works on the sentences (2), (3) and (4), it does not scales up easily. The top-down parser is really flawed and breaks easily. Using the NLTK parser [5] would solve this. Features can be added directly to the lexicon, the unifier used in the script deals with abstract features instances. A great improvement could rely on the implementation of the Novelty condition.

References

[1] Denis Bouchard. Une explication cognitive des effets attribués à la c-commande dans les contraintes sur la coréférence. *Corela. Cognition, représentation, langage*, (HS-7), 2010.

[2] Gereon Müller. *Phrase Structure and Derivations*, Institut für Linguistik, Universität Leipzig. `http://home.uni-leipzig.de/muellerg/mu314.pdf`, 2008. Accessed: 2018-06-22.

[3] Thomas Wasow. *Anaphora in generative grammar*, volume 2. John Benjamins Publishing, 1979.

[4] `https://github.com/ychalier/coref/`. Accessed: 2018-06-21.

[5] Natural language toolkit. `http://www.nltk.org`. Accessed: 2018-06-21.