A Logical Approach to Sentiment Analysis

Niklas Christoffer Petersen

Kongens Lyngby 2012
IMM-MSc-2012-126

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary (English)

This thesis presents a *formal logical approach* for *entity level* sentiment analysis which utilizes *machine learning techniques* for efficient syntactical tagging, and performs a deep structural analysis of the syntactical properties of texts in order to yield precise results.

The method should be seen as an alternative to pure machine learning methods for sentiment analysis, which are argued to have high difficulties in capturing long distance dependencies, and be dependent on significant amount of domain specific training data.

To demonstrate the method a *proof of concept* implementation is presented, and used for testing the method on real data sets. The results shows that the method yields high correctness, but further investment are needed in order to improve its robustness.

Summary (Danish)

Denne afhandling præsenterer en *formel logisk tilgang* for meningsanalyse på *enhedsniveau*, som anvender *machine learning*-teknikker for effektiv syntaktisk *tagging*, og udfører en dyb struktural analyse af syntaktiske egenskaber af tekster for at give præcise resultater.

Metoden skal ses som et alternativ til metoder for meningsanalyse baseret på ren *machine learning*. Det argumenteres at disse har høje vanskeligheder med at opfange langdistance relationer, samt være afhængig af en betydelig mængde af domænespecifik træningsdata.

For at demonstrere metoden præsenteres en *proof of concept* implemtentering, som anvendes til afprøvning af metoden på virkelige datasæt. Resultaterne viser, at metoden giver høj korrekthed, men yderligere investeringer er nødvendige for at forbedre sikre robustheden af metoden.

Preface

This thesis was prepared at Department of Informatics and Mathematical Modelling at the Technical University of Denmark in partial fulfilment of the requirements for acquiring the MSc degree in Computer Science and Engineering.

The project concerns extraction of opinions occurring in natural language text, also known as sentiment analysis and the thesis presents a formal logical method as a proposed solution. The reader is assumed to have reasonable knowledge of combinatorial logic and formal languages, as well as fundamental knowledge of computational linguistics.

This project was conducted in the period April 1, 2012 to September 30, 2012 under the supervision of Jørgen Villadsen, and was valued at 35 ECTS credit points. The project specific learning objectives for the project were:

- *Understand and extend modern techniques for processing of natural language texts using formal logical systems.*
- *Demonstrate methods for formal reasoning with respect to natural language understanding.*
- *Present a proof of concept system, that is a fully functional implementation of essential theoretical presented methods.*

Kgs. Lyngby, September 30, 2012

Niklas Christoffer Petersen

Acknowledgements

I would like to thank my supervisor Jørgen Villadsen for his help and guidance during the entire project, and for his lectures in the course Formal Logical Systems (02156) which sparked my interest in the area of formal logic and later lead to my deep interest in computational linguistics. He has also acted as supervisor for several free-standing projects on my master study with focus on natural language processing and formal semantics, and also supervised my bachelor project which shared a few areas with this work, however on a far more novice level.

Also the courses Program Analysis (02242) and Functional Programming (02157) have contributed with knowledge crucial to the completion of my thesis. I attended the 24th European Summer School in Logic, Language and Information (ESSLLI-2012) in Opole, Poland during the project, which also provided highly advanced knowledge that has been applied in this thesis.

I would also like to give thanks to Henriette Jensen and Johannes Svante Spurkeland for providing test data by individually labeling review texts. Further thanks to Michael Lunøe and Johannes Svante Spurkeland for sparring and constructive feedback during this prject.

Contents

Chapter 1

Introduction

The study of opinion is a one of the oldest fields, with roots in philosophy, going back to the Ancient Greek philosophers. The wide adoption of the Internet has made it possible for individuals to express their subjective opinions to an extent much more far-reaching then possible before. This has recently been intensified even more due to the explosive popularity of social networks and microblogging services.

The amount of opinion data available is often huge compared to what traditional opinion analyses, e.g. questionnaire surveys, requires to yield significant results. Furthermore the opinions cover nearly every thinkable topic. This gives incentive, given that the potential value of such opinions can be great, if information can be extracted effectively and precisely. Given enough opinions on some topic of interest, they can yield significant indication of *collective opinion shifts*, e.g. shifts in market trends, political sympathies, etc. The interest in such shifts is far from recent, and is a well established subfield of the *psychometrics* and has strong scientific grounds in both psychology and statistics.

However, since these opinions are often stated in an informal setting using natural language, usual methods developed for traditional opinion analyses, e.g. questionnaire surveys, cannot be directly applied on the data. The burst of computational power available has meanwhile made it possible to automatically analyze and classify these huge amounts of opinion data. The application of computational methods to extract such opinions are more commonly known as *sentiment analysis*.

This thesis presents a *formal logical method* to extract the *sentiment* of natural language text reviews. In this chapter traditional methods for data collection of sentiments are briefly considered and thereafter the overall challenges involved in collecting reviews stated in natural language are presented. The opinions considered in this thesis are in form of product and service reviews, however most of the techniques presented can be generalized to other types of topics.

1.1 Classical data collection

One of the most used approaches to collect data for opinion analyses is through questionnaire surveys. Most of us are familiar with such surveys, where the subject is forced to answer questions with a fixed scale. For instance, given the statement "The rooms at the Swissôtel Hotel are of high quality.", a subject must answer by selecting one of a predefined set of answers, e.g. as shown in Figure 1.1.

1. Strongly disagree
2. Disagree
3. Neither agree nor disagree
4. Agree
5. Strongly agree

Figure 1.1: Likert scale.

Such scales, where the subject indicates the *level of agreement*, are know as *Likert scales*, originally presented by Likert [1932], and has been one of the favorite methods of collection data for opinion analyses. Other scales are also widely used, for instance the *Guttman scale* [Guttman, 1949], where the questions are binary (yes/no) and ordered such that answering yes to a questions implies the answer yes to all questions ordered below this. An example is shown in Figure 1.2. Thus the answer on both a Likert and a Guttman scale can be captured by a single *opinion value*.

Given a set of answers, the result of such surveys are fairly easy to compute. At its simplest it can be a per question average of the opinion values, however it is mostly also interesting to connect the questions – for instance how does subjects' answer to the above statement influent their answer to the statement "The food at the Swissôtel Restaurant is of high quality.", etc.

1. I like eating out
2. I like going to restaurants
3. I like going to themed restaurants
4. I like going to Chinese restaurants
5. I like going to Beijing-style Chinese restaurants

Figure 1.2: Guttman scale.

One advantage of using fixed frameworks as the Likert and Guttman scales is that the result of the data collection is highly well-structured, and multiple answers are known to be provided by the same subject. This makes further analysis as the example just mentioned possible, something that will be much harder to achieve when harvesting reviews from the Internet, where the author of the review is presumably unknown, or at least not connected to any other reviews. Furthermore, since most questionnaire surveys are conducted in relatively controlled settings, where the subjects in many cases have been preselected to constitute a representative sample of some population, the results intuitively have relative high certainty.

However these properties also contributes to some of the disadvantages of classical data collection, namely the difficulty of getting people to answer them. Another issue is that people only can answer on the questions that are provided, which mean that significant aspects of the subjects opinion might not be uncovered if it is not captured by a question.

1.2 Natural language data collection

In this thesis it is argued that a far more natural way for subjects to express their opinions is through their most natural communication form, i.e. their language. The strongest incentive for considering natural language texts as a data source is simply the amount of data available through the Internet. This especially includes posts on social networking and microblogging services, e.g. *Facebook*[1] and *Twitter*[2], where people often express the opinion on products and services, but also online resellers allowing their consumers to publicly review their producs such as *Amazon*[3]

[1]Facebook, http://www.facebook.com/
[2]Twitter, http://www.twitter.com/
[3]Amazon, http://www.amazon.com/

This though introduces the need for efficient candidate filtering as the posts in general, of cause, are not constrained to a specific entity or topic of interest. This can be fairly easy achieved as most of the services provides APIs that allows keyword filtering. The approach also raises ethical issues, since the author of the post might never realize that it is being used for the purpose of opinion analysis. Larger texts, such as blog posts, could indeed also be considered, however the contextual aspects of large, contiguous texts often makes interpretation extremely complex, thus making it a difficult task to extract opinions on a specific entity. In this thesis only relatively short reviews are thus considered.

One concern is whether texts harvested from the Internet can constitute a representative sample of the population in question. The actual population, of course, rely on the target of the analysis. This is a non-trivial study itself, but just to demonstrate the sample bias that often are present consider Figure 1.3. The figure shows the age distribution of respectively Twitter Users and the population of Denmark cf. [Pingdom, 2010] and [Eurostat, 2010]. If the target group was Danes in general, harvesting opinions from Twitter without any correction would presumably cause some age groups to be vastly overrepresented, i.e. the mid-aged Danes, while others would be underrepresented, i.e. young and old Danes.

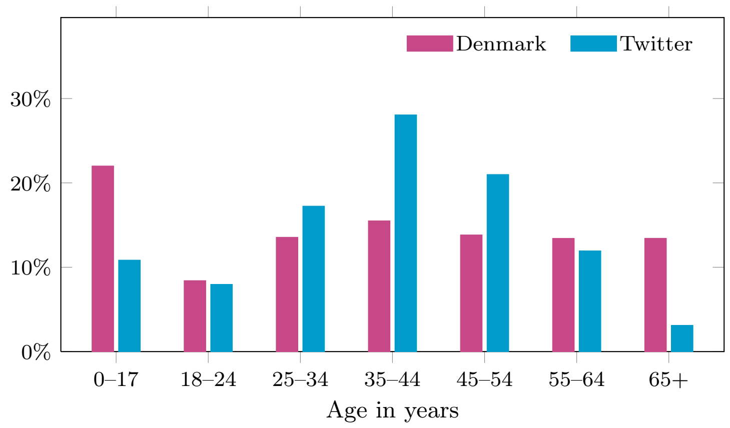

Figure 1.3: Age of Twitter Users and population of Denmark.

Further details on this issue will not be concerned, but it is indeed necessary to correct collected data for sampling bias in order to draw any significant conclusions, such that the distribution of collected opinions indeed follows the target of the analysis.

Another more progressive approach for natural language data collection could be *opinion seeking queries* as the one shown in (1.1). Such queries are intended to ensure succinct reviews that clearly relate to the *entity* in question (e.g. product or service) with respect to a specific *topic of interest*.

What do you think about pricing at the Holiday Inn, London? (1.1)

This method might not seem that different from that of the previously mentioned Likert scales, but it still allows the reviewer to answer with a much broader sentiment and lets the reviewer argue for his/hers answer as shown in the examples (1.2, 1.3).

The price is moderate for the service and the location. (1.2)

Overall an above average hotel based on location and price but not one for a romantic getaway! (1.3)

1.3 Sentiment of a text

This section gives a succinct presentation of sentiment analysis, and introduce it as a research field. The research in sentiment analysis has only recently enjoyed high activity cf. [Liu, 2007], [Pang and Lee, 2008], which probably is due to a combination of the progress in machine learning research, the availability of huge data sets through the Internet, and finally the commercial applications that the field offers. Liu [2007, chap. 11] identify three *kinds* of sentiment analysis:

- *Sentiment classification* builds on text classification principles, to assign the text a *sentiment polarity*, e.g. to classify the entire text as either positive or negative. This kind of analysis works on *document level*, and thus no details are discovered about the entity of the opinions that are expressed by the text. The result is somewhat coarse, e.g. it seems to be hard to classify (1.4) as *either* positive or negative, since it contains multiple opinions.

  The buffet was expensive, but the view is amazing. (1.4)

- *Feature-based sentiment analysis* works on *sentence level* to discover opinions about entities present in the text. The analysis still assigns *sentiment polarities*, but on an entity level, e.g. the text (1.4) may be analyzed to express a negative opinion about the *buffet*, and a positive opinion about the *view*.
- *Comparative sentence and relation analysis* focus on opinions that describes similarities or differences of more than one entity, e.g. (1.5).

  The rooms at Holiday Inn are cleaner than those at Swissôtel. (1.5)

The kind of analysis presented by this thesis is closest to the *feature-based sentiment analysis*, however Liu [2007, chap. 11] solely describes methods that uses *machine learning approaches*, whereas this thesis focuses on a *formal logical approach*. The difference between these approaches, and arguments for basing the solution on formal logic will be disclosed in the next section, and further details on the overall analytic approach is presented in Chapter 2.

Finally Liu [2007, chap. 11] identify two *ways* of expression opinion in texts, respectively *explicit* and *implicit* sentiments. An explicit sentiment is present when the sentence directly expresses an opinion about a subject, e.g. (1.6), whereas an implicit sentiment is present when the sentence implies an opinion, e.g. (1.7). Clearly sentences can contain a mix of explicit and implicit sentiments.

The food for our event was delicious. (1.6)

When the food arrived it was the wrong order. (1.7)

Most research focus on the explicit case, since identifying and evaluating implicit sentiment is an extremely difficult task which requires a high level of domain specific knowledge, e.g. in (1.7) where most people would regard it as negative if a restaurant served another dish then what they ordered. To emphasize this high domain dependency Pang and Lee [2008] considers the sentence (1.8), which in the domain of *book reviews* implies a positive sentiment, but the exact same sentence implies a negative sentiment in the domain of *movie reviews*.

Go read the book! (1.8)

The thesis will thus focus on the explicit case, since the implicit case was considered to simply require too much domain specific knowledge. This is due to two reasons, firstly the presented solution should be adaptable to *any* domain, and thus tying it too closely to one type of domain knowledge was not an option, secondly the amount of domain knowledge required is in the vast number of cases simply not available, and thus needs to be constructed or collected. With that said the explicit case is neither domain independent, which is a problematic briefly touched in the next section, and detailed in Section 2.4.

1.4 The logical approach

A coarse classification of the different approaches to sentiment analysis is to divide it into two classes: *formal approaches* and *machine learning approaches*. To avoid any confusion this thesis will present a method that belong to the formal class.

- *Formal approaches* models the texts to analyze as a formal language, i.e. using a formal grammar. This allows a deep syntactic analysis of the texts, yielding the structures of the texts, e.g. sentences, phrases and words for *phrase structure grammars*, and binary relations for *dependency grammars*. Semantic information is then extractable by augmenting and inspecting these structures. The result of the semantic analysis is then subject to the actual sentiment analysis, by identifying positive and negative concepts, and how these modifies the subjects and objects in the sentences.

- *Machine learning approaches* uses feature extraction to train probabilistic models from a set of labeled train data, e.g. a set of texts where each text is labeled as either positive or negative for the *sentiment classification*-kind analysis. The model is then applied to the actual data set of which an analysis is desired. If the feature extracting *really* does captures the features that are significant with respect to a text either being negative or positive, and the texts to analyze has the *same* probability distribution as the training data, then the text will be classified correctly.

Notice that the presented classification only should be interpreted for the process of the actual sentiment analysis, not any preprocessing steps needed in order to apply the approach. Concretely the presented formal approach indeed do rely on machine learning techniques in order to efficiently identify lexical-syntactic properties of the text to analyze as will be covered in Chapter 4.

The motivation for focusing on the formal approach is two-folded: Firstly, different domains can have very different ways of expressing sentiment. What is considered as positive in one domain can be negative in another, and vice-verse. Likewise what is weighted as significant (i.e. either positive or negative) in one domain maybe completely nonsense in another, and again vice-verse, Scientific findings for this are shown by Blitzer *et al.* [2007], but also really follows from basic intuition. Labeled train data are sparse, and since machine learning mostly assumes at least some portion of labeled target data are available this constitutes an issue with the pure machine learning approach. The end result is that the models follows different probability distributions, which complicates the approach, since such biases needs to be corrected, which is not a trivial task.

Secondly, machine learning will usually classify sentiment on document, sentence or simply on word level, but not on an entity level. This can have unintended results when trying to analyze sentences with coordination of sentiments for multiple entities, e.g. (1.4). The machine learning approaches that do try to analyze on entity level, e.g. *feature-based sentiment analysis* by Liu [2007, chap. 11], relies on some fixed window for feature extraction, e.g. Liu [2007, chap. 11] uses n-grams. As a result such methods fails to detect long distance dependencies between an entity and opinion stated about that entity. An illustration of this is shown by the potentially unbound number of *relative clauses* allowed in English, e.g. (1.9), where *breakfast* is described as *best*, however one would need to use a window size of at least 9 to detect this relation, which is much larger then normally considered (Liu only considers up to trigrams).

The breakfast that was served Friday morning was the best I ever had! (1.9)

Formal logical systems are opposed to machine learning extremely precise in results. A conclusion (e.g. the sentiment value for a specific subject in a given text) is only possible if there exists a logical proof for this conclusion.

***Thesis:** It is the thesis that a logical approach will be able to capture these complex and long distance relationships between entities and sentiments, thus achieving a more fine-grained entity level sentiment analysis.*

With that said, a logical approach indeed also suffers from obvious issues, most notable robustness, e.g. if there are missing, or incorrect axioms a formal logical system will not be able to conclude anything, whereas a machine learning approach will always be able to give an estimate, which might be a very uncertain estimate, but at least a result. This issue of robustness is crucial in the context of review texts, since such may not always be grammatical correct, or even be constituted by sentences. In Section 2.2 this issue will be addressed further, and throughout this thesis it will be a returning challenge. Details on the logical approach is presented in Chapter 3

1.5 Related work

In the following notable related work on sentiment analysis is briefly presented. As mentioned there are two main flavors of sentiment analysis, namely implicit and explicit. Most of the work found focus solely on the explicit kind of sentiment, just like this work does.

Furthermore it seems that there is a strong imbalance between the *formal approaches* and *machine learning approaches*, with respect to amount of research, i.e. there exists a lot of research on sentiment analysis using machine leaning compared to research embracing formal methods.

Notably related work using formal approaches include Tan *et al.* [2011], who presents a method of extracting sentiment from dependency structures, and also focus on capturing long distance dependencies. As dependency structures simply can be seen as binary relations on words, it is indeed a formal approach. However what seems rather surprising is that in the end they only classify on sentence-level, and thus in this process loose entity of the dependency.

The most similar work on sentiment analysis found using a formal approach is the work by Simančík and Lee [2009]. The paper presents a method to detect sentiment of newspaper headlines, in fact partially using the same grammar formalism that later will be presented and used in this work, however without the combinatorial logic approach. The paper focus on some specific problems arising with analyzing newspaper headlines, e.g. such as headline texts often do not constitute a complete sentence, etc. However the paper also present more general methods, including a method for building a highly covering map from words to polarities based on a small set of positive and negative seed words. This method has been adopted by this thesis, as it solves the assignment of polarity values on the lexical level quite elegantly, and is very loosely coupled to the domain. However their actual semantic analysis, which unfortunately is described somewhat shallow in the paper, seem to suffer from severe problems with respect to certain phrase structures, e.g. *dependent clauses.*

1.6 Using real data sets

For the presented method to be truly convincing it is desired to present a fully functional *proof of concept* implementation that shows at least the most essential capabilities. However, for such product to be demonstrated properly, real data is required. Testing in on some tiny pseudo data set constructed for the sole purpose of this demonstration would not be convincing. Chapter 5 presents essential aspects of this *proof of concept* implementation.

An immediate concern that raises when dealing with real data sets is the possibility of incorrect grammar and spelling. A solution that would only work on *perfect texts* (i.e. text with perfectly correct grammar and spelling) would not be adequate. Reasons for this could be that word is simply absent from the system's vocabulary (e.g. misspelled), or on a grammatical incorrect form (e.g. wrong person, gender, tense, case, etc.).

Dealing with major grammatical errors, such as wrong word order is a much harder problem, since even small changes in, for instance, the relative order of subject, object, verb etc. may result in an major change in interpretation. Thus it is proposed, only to focus on minor grammatical errors such as incorrect form. Chapter 6 presents an evaluation of the implementation on actual review data.

CHAPTER 2

Sentiment analysis

An continuous analog to the sentiment polarity model presented in the introduction is to weight the classification. Thus the polarity is essential a value in some predefined interval, $[-\omega; \omega]$, as illustrated by Figure 2.1. An opinion with value close to $-\omega$ is considered highly negative, whereas a value close to ω is considered highly positive. Opinions with values close to zero are considered almost neutral. This model allows the overall process of the sentiment analysis presented by this thesis to be given by Definition 2.1.

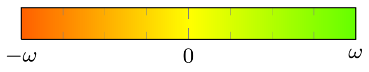

Figure 2.1: Continuous sentiment polarity model.

DEFINITION 2.1 A sentiment analysis $\mathcal{A}$ is a computation on a review text $T \in \Sigma^\star$ with respect to a *subject of interest* $s \in \mathbb{E}$, where $\Sigma^\star$ denotes the set of all texts, and $\mathbb{E}$ is the set of all entities. The result is an normalized score as shown in (2.1). The yielded score should reflect the *polarity* of the given subject of interest in the text, i.e. whether the overall opinion is positive, negative, or neutral.

$$\mathcal{A} : \Sigma^\star \to \mathbb{E} \to [-\omega; \omega] \tag{2.1}$$

■

It should be evident that this computation is far from trivial, and constitutes the cornerstone of this project. There are several steps needed, if such computation should yield any reasonable result. As mentioned in the introduction the goal is a logical approach for achieving this. The following outlines the overall steps to be completed, their associated problematics in this process, and succinctly presents different approaches to solve each step. The chosen approach for each step will be presented in much more details in later chapters.

2.1 Tokenization

In order to even start processing natural language texts, it is essential to be able to identify the elementary parts, i.e. *lexical units* and *punctuation marks*, that constitutes a text. Decent tokenization is essential for all subsequent steps. However even identifying the different sentences in a text can yield a difficult task. Consider for instance the text (2.2) which is taken from the Wall Street Journal (WSJ) corpus [Paul and Baker, 1992]. There are six periods in it, but only two of them indicates sentence boundaries, and delimits the text into its two sentences.

Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29. Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group. (2.2)

The domain of small review texts allows some restrictions and assumptions, that at least will ease this issue. For instance it is argued that the review texts will be fairly succinct, and thus it seems like a valid assumption that they will consists of only a few sentences. Its is argued that this indeed is achievable by sufficient instructing and constraining the reviewers doing data collection, e.g. only allowing up to a certain number of characters. This allows sentences in such phrases to be processed independently (i.e. as two separate review texts).

Even with this assumption, the process of identifying the sentences, and the lexical units and punctuation marks within them, is not a trivial task. Webster and Kit[1992] criticizes the neglection of this process, as most natural language processing (NLP) studies focus purely on analysis, and assumes this process has already been performed. Such common assumptions might derive from English being a relatively easy language to tokenize. This is due to its space marks as explicit delimiters between words, as opposed to other languages, e.g. Chinese which has no delimiters at all. This might hint that tokenization is very language dependent. And even though English is considered simple to tokenize, a naive approach like segmenting by the occurrence of spaces fails for the text (2.3), which is also from the WSJ corpus, as it would yield lexical units such as "(or", "perceived," and "rate),". Simply consider all groups of non-alphanumerics as punctuation marks does not work either, since this would fail for i.a. ordinal numbers, currency symbols, and abbreviations, e.g. “Nov.” and “Elsevier N.V.” in text (2.2). Both of these methods also fail to recognize “Pierre Vinken” and “Elsevier N.V.” as single proper noun units, which is arguably the most sane choice for such.

One of the fastest growing segments of the wine market is the category of superpremiums – wines limited in production, of exceptional quality (or so perceived, at any rate), and with exceedingly high prices. (2.3)

Padró *et al.* [2010] presents a framework of analytic tools, developed in the recent years, for various NLP tasks. Specially interesting is the morphological analyzer, which applies a cascade of specialized (i.e. language dependent) processors to solve exactly the tokenization. The most simple of them use pattern matching algorithms to recognize numbers, dates, quantity expressions (e.g. ratios, percentages and monetary amounts), etc. More advanced processing are needed for proper nouns, which relies on a two-level solution: first it applies a fast pattern matching, utilizing that proper nouns are mostly capitalized; and secondly statistical classifiers are applied as described by Carreras *et al.* [2002]. These recognize proper nouns with accuracy of respectively 90% and over 92%. The analyzer also tries to identify lexical units that are composed of multiple words, e.g. proper nouns and idioms.

It is thus possible, by the use of this framework, to preprocess the raw review texts collected from users, and ensure that they will be tokenized into segments that are suitable for the lexical-syntactic analysis. Thus more details on the tokenizer will not be presented.

2.2 Lexical-syntactic analysis

The syntactic analysis determines the grammatical structure of the input texts with respect to the rules of the English language. It is expected that the reader is familiar with English grammar rules and syntactic categories, including phrasal categories and lexical categories (also called parts of speech). As mentioned earlier it is essential that the presented method is able to cope with *real* data, collected from actual review scenarios. This implies a robust syntactic analysis, accepting a large vocabulary and a wide range of sentence structures. In order to calculate the actual polarity it is essential to have semantic annotations on the lexical units. It is argued that a feasible and suitable solution is to use a grammar that is *lexicalized*, i.e. where the rules are essentially language independent, and the syntactic properties are derived from a lexicon. Thus the development of a lexicalized grammar is mainly a task of acquiring a suitable lexicon for the desired language.

Even though the task of syntactic analysis now is largely reduced to a task of lexicon acquisition, which will be addressed in Chapter 4, there are still general concerns that are worth acknowledging. Hockenmaier *et al.* [2004, p. 108-110] identifies several issues in being able to efficiently handle natural language texts solely with lexicalized grammars, mainly due to the need for entries for various combinations of proper nouns, abbreviated terms, dates, numbers, etc. Instead they suggest to use pattern matching and statistical techniques as a preprocessing step, for which efficient components exists, which translate into reduced complexity for the actual syntactic analysis. The tokenization framework [Padró *et al.*, 2010] introduced in the previous section does exactly such kind of identification, and thus this should not yield significant problems.

However the domain of small review texts also introduce problematics that may not an constitute major concerns in other domains, most notable the possibility of incorrect grammar and spelling, since the texts comes unedited from humans with varying English skills. Recall from the Section that a solution that only would work on *perfect texts* (i.e. texts of sentences with completely correct grammar and spelling) would not be adequate. The grammar shoulf at least be able to handle minor misspellings. Reasons for this could be that word is simply absent from the system's vocabulary (e.g. misspelled), or on a grammatical incorrect form (e.g. wrong person, gender, tense, case, etc.).

2.3 Mildly context-sensitive grammars

There exists formal proofs that some natural language structures requires formal power beyond *context-free grammars* (CFG), i.e. [Shieber, 1985] and [Bresnan *et al.*, 1982]. Thus the search for grammars with more expressive power has long been a major study within the field of computational linguistics. The goal is a grammar that is so restrictive as possible, allowing efficient syntactic analysis, but still capable of capturing these structures. The class of *mildly context-sensitive grammars* are conjectured to be powerful enough to model natural languages while remaining efficient with respect to syntactic analysis cf. [Joshi *et al.*, 1990].

Different grammar formalisms from this class has been considered, including *Tree Adjunct Grammar* (TAG) [Joshi *et al.*, 1975], in its lexicalized form (LTAG), *Head Grammar* (HG) [Pollard, 1984] and *Combinatory Categorial Grammar* (CCG) [Steedman, 1998]. It has been shown that these are all equal in expressive power by Vijay-Shanker and Weir [1994]. The grammar formalism chosen for the purpose of this thesis is *Combinatory Categorial Grammar* (CCG), pioneered largely by Steedman [2000]. CCG adds a layer of combinatory logic onto pure Categorial Grammar, which allows an elegant and succinct formation of *higher-order* semantic expressions directly from the syntactic analysis. Since the goal of this thesis is a logical approach to sentiment analysis, CCG's native use of combinatory logic seemed like the most reasonable choice. Chapter 3 will formally introduce the CCG in much more detail.

2.4 Semantic analysis

The overall process of semantic analysis in the context of sentiment analysis is to identify the polarity of the entities appearing in the text, and to relate these entities to the *subject of interest* of the sentiment analysis. The approach is to *annotate* the lexical units of adjectives and adverbs with suitable polarities, and then fold these onto the phrasal structures, yielded by the syntactic analysis, in order to identify the bindings of these polarities, i.e. which entities they modify directly or indirectly.

There exists datasets that tries to bind a general polarity to each word in a lexicon, e.g. [Esuli and Sebastiani, 2006] and [Baccianella *et al.*, 2010]. While such might be fine for general sentiment analyses, or analyses where the domain is not known, it is argued that better results can be achieved by using a domain specific annotation. For instance the adjective "huge" might be considered positive for a review describing rooms at a hotel, while negative for a review describing sizes of cell phones.

As already mentioned, the use of a lexicalized syntactic analysis allows the annotation to appear directly on the entries in the lexicon. A manual annotation of a large lexicon is evidently not a feasible approach. Furthermore the model must also be generic enough so it can be adapted to ideally any domain contexts with minimum efforts, i.e. it is not desired to tie the model to any specific domain, or type of domains. To achieve such a model that is loosely coupled to the domain the concept of *semantics networks* was chosen cf. Russell and Norvig [2009, p. 454–456].

A semantic network is in its simplest form just a collection of different semantic concepts, and relations between them. The idea is to dynamically construct such semantic networks from a small set of domain specific knowledge, namely a set of positive and negative *seed concepts* in the domain – a technique presented by Simančík and Lee [2009]. Section 4.3 in Chapter 4 will presents details on the approach of calculating the polarities of adjectives and adverbs and additionally present some handling of negations.

The final result of the sentiment analysis is simply the aggregation of the results yielded for each of the results of the semantic analysis.

Chapter 3

Combinatory categorial grammar

In this chapter the formalism of Combinatory Categorial Grammar (CCG) is introduced, and based on this applied to the proposed sentiment analysis introduced in the previous chapter. For the purpose of explaining and demonstrating CCG a small fragment of English is used. This allow the usage of a "handwritten" lexicon initially. In Chapter 4 the issues related to acquiring, and analyzing with, a wide coverage lexicon are addressed. A CCG lexicon is defined cf. Definition 3.1.

Definition 3.1 A CCG lexicon, $\mathcal{L}_{\text{CCG}}$, is mapping from a lexical unit, $w \in \Sigma^\star$, to a set of 2-tuples, each containing a lexical category and semantic expression that the unit can entail cf. (3.1), where Γ denotes the set of lexical and phrasal categories, and Λ denotes the set of semantic expressions.

$$\mathcal{L}_{\text{CCG}} : \Sigma^\star \to \mathcal{P}(\Gamma \times \Lambda) \tag{3.1}$$

■

A *tagging* of a lexical unit $w \in \Sigma^\star$ is simply the selection of one of the pairs yielded by $\mathcal{L}_{\text{CCG}}(w)$. Thus given some ordered set of lexical units, which constitutes the text $T \in \Sigma^\star$ to analyse, there might exists many different taggings. This is simply due to the fact that a lexical unit can entail different lexical categories (e.g. "service" is both a noun and a verb), and different semantic expressions (e.g. the noun "service" can both refer to assistance and tableware). The number of taggings can thus be large, but is always finite.

The set of lexical and phrasal categories, Γ, is of a somewhat advanced structure in the CCG presented, since it follows recent work by Baldridge and Kruijff [2003] to incorporate *modalities*. A category is either *primitive* or *compound*. The set of primitive categories, $\Gamma_{\text{prim}} \subset \Gamma$, is language dependent and, for the English language, it consists of S (sentence), NP (noun phrase), N (noun) and PP (prepositional phrase). Compound categories are recursively defined by the infix operators $/_\iota$ (forward slash) and $\backslash_\iota$ (backward slash), i.e. if α and β are members of Γ, then so are $\alpha/_\iota\beta$ and $\alpha\backslash_\iota\beta$. This allows the formation of all other lexical and phrasal categories needed. The operators are left associative, but to avoid confusion inner compound categories are always encapsulated in parentheses througout this thesis.

The basic intuitive interpretation of $\alpha/_\iota\beta$ and $\alpha\backslash_\iota\beta$ is as a function that takes a category β as argument and yields a result of category α. Thus the argument is always stated on the right side of the operators, and the result on the left. The operator determines the dictionality of the application, i.e. *where* the argument should appear relative to the function: the forward operator ($/_\iota$) denotes that the argument must appear on the right of the function, whereas the backward operator ($\backslash_\iota$) denotes that the argument must appear on the left. The subscript, ι, denotes the *modality* of the operator, which is a member of a finite set of modalities $\mathcal{M}$ and will be utilized to restrict acceptence in the next section.

The syntactic categories constitutes a type system for the semantic expressions, with a set of primitive types, $\mathcal{T}_{\text{prim}} = \{\tau_x \mid x \in \Gamma_{\text{prim}}\}$. Thus, if a lexicon entry has category $(N\backslash_\iota N)/_\iota(S/_\iota NP)$ then the associated semantic expression must honor this, and have type $(\tau_{\text{NP}} \rightarrow \tau_{\text{S}}) \rightarrow \tau_{\text{N}} \rightarrow \tau_{\text{N}}$ ($\rightarrow$ is right assosiative). This is a result of the *Principle of Categorial Type Transparency* [Montague, 1974], and the set of all types are denoted $\mathcal{T}$. For now it is sufficient to describe the set of semantic expressions, Λ, as the set of *simply-typed* λ-expressions, Λ', cf. Definition 3.2. In Section 3.4 this is extended to support the desired sentiment analysis.

Definition 3.2 The set of simply typed λ-expressions, Λ', is defined recursively, where an expression, e, is either a variable x from an infinite set of typed variables $\mathcal{V} = \{v_1 : \tau_\alpha, v_2 : \tau_\beta, \ldots\}$, a functional abstraction, or a functional application. For futher details see for instance [Barendregt *et al.*, 2012].

$$\begin{aligned}
x : \tau \in \mathcal{V} \quad &\Rightarrow \quad x : \tau \in \Lambda' && \text{(Variable)}\\
x : \tau_\alpha \in \mathcal{V},\ e : \tau_\beta \in \Lambda' \quad &\Rightarrow \quad \lambda x.e : \tau_\alpha \rightarrow \tau_\beta \in \Lambda' && \text{(Abstraction)}\\
e_1 : \tau_\alpha \rightarrow \tau_\beta \in \Lambda',\ e_2 : \tau_\alpha \in \Lambda' \quad &\Rightarrow \quad (e_1 e_2) : \tau_\beta \in \Lambda' && \text{(Application)}
\end{aligned}$$

■

3.1 Combinatory rules

CCGs can be seen as a logical deductive proof system where the axioms are members of $\Gamma \times \Lambda$. A text $T \in \Sigma^\star$ is accepted as a sentence in the language, if there exists a deductive proof for S, for some tagging of T.

The inference rules of the proof system are known as *combinators*, since they take one or more function pairs, in the form of instances of $\Gamma \times \Lambda$, and produces new instances from the same set. The combinators determines the expressive power of the grammar. A deep presentation of which rules are *needed*, and thus the linguistic motivation behind this, is out of the scope of this thesis. In the following essential combinators covered by Steedman [2011, chap. 6] are succinctly described, which constitutes a *midely context-sensitive* class grammar. These are the development of the combinatory rules Steedman presented in [2000, chap. 3], however with significant changes with respect to coordinating conjucntions, due to the introduction of modalities on the infix operators.

The set of modalities used, $\mathcal{M}$, follows [Baldridge and Kruijff, 2003] and [Steedman, 2011], where $\mathcal{M} = \{\star, \diamond, \times, \cdot\}$. The set is partially ordered cf. the lattice (3.2).

$$\begin{tikzcd}
 & \star \arrow[dl, dash] \arrow[dr, dash] & \\
\diamond \arrow[dr, dash] & & \times \arrow[dl, dash] \\
 & \cdot &
\end{tikzcd} \tag{3.2}$$

The basic concept of annotating the infix operators with $\iota \in \mathcal{M}$, is to restrict the application of inferrence rules during deduction in order ensure the soundness of the system. Categories with $\star$ is most restrictive, allowing only basic rules, $\diamond$ allows rules which perserves the word order, $\times$ allows rules which permutate the word order, and finally categories with $\cdot$ allows any rule without restrictions. The partial ordering allows the most restrictive categories to also be included in the less restrictive, e.g. any rule that assumes $\alpha/_\diamond\beta$ will also be valid for $\alpha/.\beta$. Since $\cdot$ permits any rule it is convenient to simply write $/$ and $\backslash$ instead of respectively $/.$ and $\backslash.$, i.e. the dot is omitted from these operators.

The simplest combinator is the *functional application*, which simply allows the instances to be used as functions and arguments, as already described. The forward and backward functional application combinator can be formulated as respectivly $(>)$ and $(<)$, where X and Y are variables ranging over lexical and phrasal categories, and f and a are variables ranging over semantic expressions. Since the operators are annotated with $\star$, the rules can apply to even the most restrictive categories. For readability instances (α, e) of $\Gamma \times \Lambda$ is written $\alpha : e$. Notice that since the semantic expressions are typed, the application of f on a is sound.

$$X/_\star Y : f \quad Y : a \quad \Rightarrow \quad X : f\,a \tag{>}$$
$$Y : a \quad X\backslash_\star Y : f \quad \Rightarrow \quad X : f\,a \tag{<}$$

With only these two simple combinatory rules, (>) and (<), the system is capable of capturing any context-free langauge cf. Steedman [2000, p. 34]. For the fragment of English, used to demonstrate CCG, the lexicon is considered to be finite, and it is thus possible, and also convinient, to simply write the mapping of entailment as a subset of $\Sigma^\star \times \Gamma \times \Lambda$. Figure 3.1 shows a fragment of this demonstration lexicon. For readability, instances (w, α, e) of $\Sigma^\star \times \Gamma \times \Lambda$ is written $w \models \alpha : e$. Notice that the semantic expressions are not yet specified, since it for now is sufficient that just the type of the expressions is correct, and this follows implicitly from the category of the entry.

$$\begin{aligned}
\textbf{the} &\models NP/_\diamond N : (\ldots) && \text{(Determiners)}\\
\textbf{an} &\models NP/_\diamond N : (\ldots) && \\
\textbf{hotel} &\models N : (\ldots) && \text{(Nouns)}\\
\textbf{service} &\models N : (\ldots) && \\
\textbf{had} &\models (S\backslash NP)/NP : (\ldots) && \text{(Transative verbs)}\\
\textbf{exceptional} &\models N/N : (\ldots) && \text{(Adjectives)}
\end{aligned}$$

Figure 3.1: A fragment of a tiny handwritten lexicon.

The lexicon for instance shows how determiners can be modeled by the category which takes a noun on the right and yields a noun phrase. Likewise a transitive verb is modeled by a category which first takes a noun phrase on the right (the object), then a noun phrase on the left (the subject) and lastly yields a sentence. Figure 3.2 shows the deduction of S from the simple declarative sentence "the hotel had an exceptional service" (semantics are omitted).

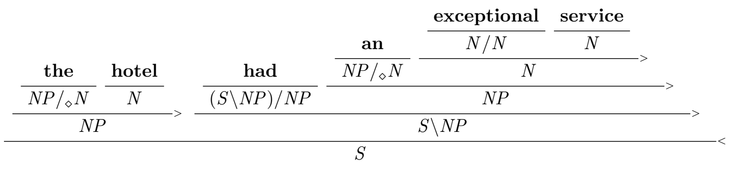

Figure 3.2: Deduction of simple declarative sentence.

Besides functional application, CCG also has a set of more restrictive rules, including *functional composition*, defined by the forward and backward functional composition combinators, respectively ($>_\mathbf{B}$) and ($<_\mathbf{B}$), where Z likewise is a variable ranging over Γ, and g over Λ.

$$X/_\diamond Y : f \quad Y/_\diamond Z : g \quad \Rightarrow \quad X/_\diamond Z : \lambda a.f(g\ a) \qquad (>_\mathbf{B})$$
$$Y\backslash_\diamond Z : g \quad X\backslash_\diamond Y : f \quad \Rightarrow \quad X\backslash_\diamond Z : \lambda a.f(g\ a) \qquad (<_\mathbf{B})$$

Notice that the semantic expression yielded by ($>_\mathbf{B}$) and ($<_\mathbf{B}$) is equivalent to regular functional composition ($\circ$) of f and g, but since $f \circ g \notin \Lambda$ they need to be written as λ-expressions.

Functional composition is often used in connection with another rule, namely *type-raising*, defined by the forward and backward type-raising combinators, respectively ($>_\mathbf{T}$) and ($<_\mathbf{T}$), where T is a variable ranging over categories.

$$X : a \quad \Rightarrow \quad T/_\iota(T\backslash_\iota X) : \lambda f.fa \qquad (>_\mathbf{T})$$
$$X : a \quad \Rightarrow \quad T\backslash_\iota(T/_\iota X) : \lambda f.fa \qquad (<_\mathbf{T})$$

Type-rasing allows a often primitive category, X, to raise into a category that instead captures a compound category, which is a function over X. The modality of the result is not controllable and is thus often suppressed, however any constrains of the applicability of X of cause continue cf. [Baldridge and Kruijff, 2003].

Notice that the introduction of these rules, i.e. functional composition and type-raising, allows deductional ambiguity, i.e. a proof for a sentence may be achievable by multiple deductions as shown in Figure 3.3 (trivial deductions are assumed). However such ambiguities are immaterial, since they do not correspond to semantic ambiguities.

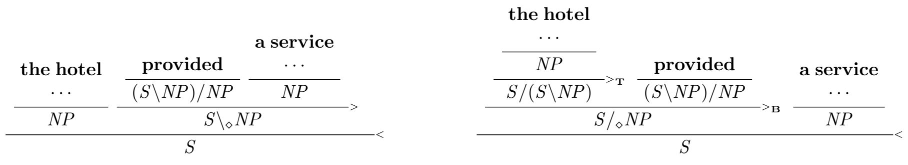

Figure 3.3: Multiple deductions of the same sentence.

A system with these rules demonstrates what is arguably CCG's most unique advantage, namely the ability to handle *unbounded dependencies* without any additional lexicon entries. For instance a transitive verb, with the *same* category as shown in

$$\textbf{that} \models (N\backslash_{\diamond}N)/(S/_{\diamond}NP) : (\ldots) \qquad \text{(Relative pronouns)}$$
$$\textbf{that} \models (N\backslash_{\diamond}N)/(S\backslash_{\diamond}NP) : (\ldots)$$

Figure 3.4: Fragment of lexicon for the relative pronoun “that”.

Figure 3.1, can participate in relative clauses as shown in Example 3.1, given the presence of a small set of entries for relative pronouns, e.g. Figure 3.4.

Example 3.1 *Figure 3.5 shows an example of both type-rasing and functional composition. The transitive verb (provided) is requiring an object in the form of a noun phrase to its right. However, since it participate in a relative clause, its object is given by the noun that the clause modifies. Type raising allows the subject of the relative clause to raise into a category that can compose with the verb, and thus allows the relative pronoun (that) to bind the relative clause to the noun.*

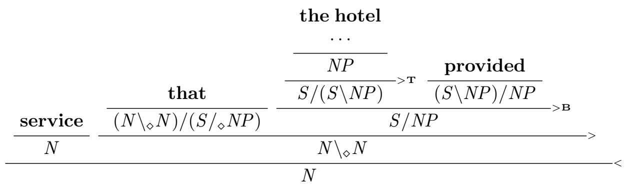

Figure 3.5: Deduction of noun phrase with relative clause.

■

The last set of rules presented here is the *crossed functional composition*, defined by the forward and backward crossed functional composition combinators, respectively $(>_{\mathbf{B}_\times})$ and $(<_{\mathbf{B}_\times})$.

$$X/_{\times}Y : f \quad Y\backslash_{\times}Z : g \quad \Rightarrow \quad X\backslash_{\times}Z : \lambda a.f(g\ a) \qquad (>_{\mathbf{B}_\times})$$
$$Y/_{\times}Z : g \quad X\backslash_{\times}Y : f \quad \Rightarrow \quad X/_{\times}Z : \lambda a.f(g\ a) \qquad (<_{\mathbf{B}_\times})$$

Crossed functional composition allows *permutation* of the word order. This is usefull to allow adverbs in sentences with shifting of heavy noun phrases as shown in Example 3.2.

Example 3.2 *Normally an adverb is put after the object of the verb it modifies in English, e.g. "the hotel served breakfast daily". However if the object of the verb becomes "heavy" it may sometimes be moved to the end of the sentence, e.g. "the hotel served daily a large breakfast with fresh juice".*

In such cases the adverb needs to compose with the verb, before the verb combines with its object. The crossed functional composition allows exatly such structures as shown in Figure 3.6.

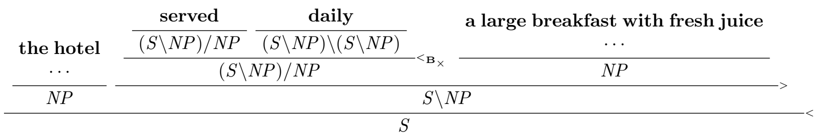

Figure 3.6: Deduction of "heavy" noun phrase shifting.

■

Steedman [2000; 2011] introduces a few additional combinators to capture even more "exotic" linguistic phenomenas. Recollect that the rules are language independent, and indeed some of the additional phenomenas covered by Steedman are either considered infrequent (e.g. *parasitic gaps*), or even absent (e.g. *cross-serial dependencies*), from the English language desired to cover by this sentiment analysis. It will later be shown (Chapter 4) that the rules already presented indeed cover a substantial part of English.

3.2 Coordination

As mentioned in the introduction, one of the goals is to correctly capture the sentiment of entities in sentences with coordination of multiple opinions.

Coordination by appearance of a coordinating conjunction, such as *and*, *or*, *but*, punctuation and comma, etc., can be modeled simply by the intuition that such should bind two constituents of same syntactic category, but with different semantic expressions, and yield a result also of that category. Some examples of the *and* coordinating conjunction are shown in Figure 3.7.

$$\begin{aligned} \textbf{and} &\models (S\backslash_\star S)/_\star S : (\ldots) && \text{(Conjunctions)} \\ \textbf{and} &\models (N\backslash_\star N)/_\star N : (\ldots) \\ \textbf{and} &\models (NP\backslash_\star NP)/_\star NP : (\ldots) \\ &\ldots \end{aligned}$$

Figure 3.7: Fragment of lexicon for the coordinating conjunction "and".

It now becomes evident, why the modalities are needed, since application of the crossed composition combinators without any restrictions could allow scrambled sentences to be deducted falsely, e.g. Figure 3.8.

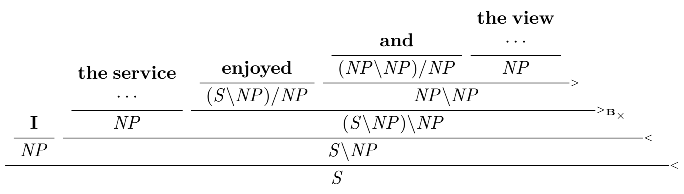

Figure 3.8: Unsound deduction of sentence given absence of modalities.

Similar pit-falls are possible if unresticted application of ($>_{\mathbf{B}}$) and ($<_{\mathbf{B}}$) was allowed, as shown by Baldridge [2002, chap. 4] for the Turkish language. This justifies the requirement for the modalities Baldridge originally proposed in [2002, chap. 5] and Baldridge and Kruijff presented in a refined version in [2003].

3.3 Features and agreement

The syntactic analysis until now has concerned the acceptable order of lexical units based on their categories. However, to guarantee that the accepted phrases indeed follows correct grammar, the *features* of the lexical units must also *agree*. The set of features that might apply is language dependent, for instance most indo-european languages state features for person (e.g. 1st, 2nd or 3rd), number (e.g. singular or plural), gender (e.g. male or female), etc. To incorporate this the primitive categories, Γ_{prim}, cannot be seen as atomic entities, but instead as structures that carries features, e.g. S_{dcl} and $NP_{\text{sg,3rd}}$ denotes respectively a *declarative* sentence, and a *singular, 3rd-person* noun phrase. A set of features *agrees* with another if they do not contain different elements of the same *kind*. For instance $NP_{\text{sg,3rd}}$ agree with NP_{sg}, but not with $NP_{\text{pl,3rd}}$, etc.

However, as mentioned in Section 2.2, a strict enforcement is not intended for the purpose of sentiment analysis, e.g. reviews containing small grammatical errors, such as wrong number as shown in (3.3), should not be discarded simply for this reason.

The hotel have great service (3.3)

However completely ignoring the features is neither an option. An evident demonstration of this is the usage of *predicative adjectives*, e.g. adjectives that modify the subject in a sentence with a *linking verb* as shown in Figure 3.9. Without the correct features, having such entries in the lexicon would allow sentences as "the hotel great", which of cause is not desired. The linguistic background for the which features are considered necessary for English is not within the scope of this thesis, but one is given by Hockenmaier [2003], and that feature-set will be used.

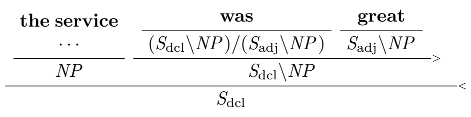
the service
...
NP
was
$(S_{\mathrm{dcl}}\backslash NP)/(S_{\mathrm{adj}}\backslash NP)$
great
$S_{\mathrm{adj}}\backslash NP$
>
$S_{\mathrm{dcl}}\backslash NP$
<
S_{dcl}

Figure 3.9: Sentence with predicative adjective.

3.4 Extending the semantics

The CCG presented in the previous sections has been based on established literature, but in order to apply the grammar formalism to the area of sentiment analysis the expressive power of the semantics needs to be adapted to this task. Until now the semantics has not been of major concern, recall that it just was defined as simply typed λ-expressions cf. Definition 3.2. Furthermore the actual *semantics* of these semantic expressions has not been disclosed, other than the initial use of λ-expressions might hint that ordinary conventions of such presumably apply. The syntax of the semantic expressions are given by Definition 3.3.

Definition 3.3 The set of semantic expressions, Λ, is defined as a superset of Λ' (see Definition 3.2). Besides variables, functional abstraction and functional application, the following structures are available:

- A n-ary *functor* ($n \geq 0$) with name f from an infinite set of functor names, polarity $j \in [-\omega; \omega]$, and *impact argument* k ($0 \leq k \leq n$).
- A *sequence* of n semantic expressions of the *same* type.

- The *change of impact argument.*
- The *change* of an expression's polarity.
- The *scale* of an expression's polarity. The magnitude of which an expression's polarity may scale is given by $[-\psi; \psi]$.

Formally this can be stated:

$$\begin{array}{rcll}
e_1, \ldots, e_n \in \Lambda, 0 \leq k \leq n,\ j \in [-\omega; \omega] & \Rightarrow & f_j^k(e_1, \ldots, e_n) \in \Lambda & \text{(Functor)} \\
e_1 : \tau, \ldots, e_n : \tau \in \Lambda & \Rightarrow & \langle e_1, \ldots, e_n \rangle : \tau \in \Lambda & \text{(Sequence)} \\
e : \tau \in \Lambda, 0 \leq k' & \Rightarrow & e^{\rightsquigarrow k'} : \tau & \text{(Impact change)} \\
e : \tau \in \Lambda,\ j \in [-\omega; \omega] & \Rightarrow & e_{\circ j} : \tau \in \Lambda & \text{(Change)} \\
e : \tau \in \Lambda,\ j \in [-\psi; \psi] & \Rightarrow & e_{\bullet j} : \tau \in \Lambda & \text{(Scale)}
\end{array}$$

■

The semantics includes normal α-conversion and β-, η-reduction as shown in the semantic rewrite rules for the semantic expressions given by Definition 3.4. More interesting are the rules that actually allow the binding of polarities to the phrase structures. The *change of a functor* itself is given by the rule (FC1), which applies to functors with, impact argument, $k = 0$. For any other value of k the functor acts like a non-capturing enclosure that passes on any change to its k'th argument as follows from (FC2). The *change of a sequence* of expressions is simply the change of each element in the sequence cf. (SC). Finally it is allowed to *push change* inside an abstraction as shown in (PC), simply to ensure the applicability of the β-reduction rule. Completely analogue rules are provided for the scaling as shown in respectively (FS1), (FS2), (SS) and (PS). Finally the *change of impact* allows change of a functors impact argument cf. (IC). Notice that these *change*, *scale*, *push* and *impact change* rules are type preserving, and for readability type annotation is omitted from these rules.

Definition 3.4 The rewrite rules of the semantic expressions are given by the following, where $e_1[x \mapsto e_2]$ denotes the *safe* substitution of x with e_2 in e_1, and $FV(e)$ denotes the set of free variables in e. For details see for instance [Barendregt *et al.*, 2012].

$$\begin{array}{rcllr}
(\lambda x.e) : \tau & \Rightarrow & (\lambda y.e[x \mapsto y]) : \tau & y \notin FV(e) & (\alpha) \\
((\lambda x.e_1) : \tau_\alpha \rightarrow \tau_\beta)\,(e_2 : \tau_\alpha) & \Rightarrow & e_1[x \mapsto e_2] : \tau_\beta & & (\beta) \\
(\lambda x.(e\,x)) : \tau & \Rightarrow & e : \tau & x \notin FV(e) & (\eta)
\end{array}$$

$$
\begin{aligned}
f_j^0(e_1,\dots,e_n)_{\circ j'} &\Rightarrow f^0_{j\widehat{+}j'}(e_1,\dots,e_n) && \text{(FC1)}\\
f_j^k(e_1,\dots e_n)_{\circ j'} &\Rightarrow f^k_j(e_1,\dots,e_{k\circ j'},\dots e_n) && \text{(FC2)}\\
\langle e_1,\dots,e_n\rangle_{\circ j'} &\Rightarrow \langle e_{1\circ j'},\dots,e_{n\circ j'}\rangle && \text{(SC)}\\
(\lambda x.e)_{\circ j'} &\Rightarrow \lambda x.(e_{\circ j'}) && \text{(PC)}\\
\\
f_j^0(e_1,\dots,e_n)_{\bullet j'} &\Rightarrow f^0_{j\widehat{\cdot}j'}(e_1,\dots,e_n) && \text{(FS1)}\\
f_j^k(e_1,\dots e_n)_{\bullet j'} &\Rightarrow f^k_j(e_1,\dots,e_{k\bullet j'},\dots e_n) && \text{(FS2)}\\
\langle e_1,\dots,e_n\rangle_{\bullet j'} &\Rightarrow \langle e_{1\bullet j'},\dots,e_{n\bullet j'}\rangle && \text{(SS)}\\
(\lambda x.e)_{\bullet j'} &\Rightarrow \lambda x.(e_{\bullet j'}) && \text{(PS)}\\
\\
f_j^k(e_1,\dots e_n)^{\rightsquigarrow k'} &\Rightarrow f^{k'}_j(e_1,\dots e_n) && \text{(IC)}
\end{aligned}
\tag{3.4}
$$

■

It is assumed that the addition and multiplication operator, respectively $\widehat{+}$ and $\widehat{\cdot}$, always yields a result within $[-\omega;\omega]$ cf. Definition 3.5.

Definition 3.5 The operators $\widehat{+}$ and $\widehat{\cdot}$ are defined cf. (3.5) and (3.6) such that they always yield a result in the range $[-\omega;\omega]$, even if the pure addition and multiplication might not be in this range.

$$
j\widehat{+}j' = \begin{cases} -\omega & \text{if } j+j' < -\omega \\ \omega & \text{if } j+j' > \omega \\ j+j' & \text{otherwise} \end{cases} \tag{3.5}
$$

$$
j\widehat{\cdot}j' = \begin{cases} -\omega & \text{if } j\cdot j' < -\omega \\ \omega & \text{if } j\cdot j' > \omega \\ j\cdot j' & \text{otherwise} \end{cases} \tag{3.6}
$$

■

The presented definition of semantic expressions allows the binding between expressed sentiment and entities in the text to be analyzed, given that each lexicon entry have associated the proper expression. Chapter 4 will go into more detail on how this is done for a wide-covering lexicon, but for know it is simply assumed that these are available as part of the small "handwritten" demonstration lexicon. Example 3.3 shows how to apply this for the simple declarative sentence from Figure 3.2, while Example 3.4 considers an example with long distance dependencies.

Example 3.3 *Figure 3.10 shows the deduction proof for the sentence "the hotel had an exceptional service" including semantics. The entity "service" is modified by the adjective "exceptional" which is immediately to the left of the entity. The semantic expression associated to "service" is simply the zero-argument functor, initial with a neutral sentiment value. The adjective has the "changed identity function" as expression with a change value of 40. Upon application of combinatorial rules, semantic expressions are reduced based on the rewrite rules given in Definition 3.4. The conclusion of the deduction proof is a sentence with a semantic expression preserving most of the surface structure, and includes the bounded sentiment values on the functors. Notice that nouns, verbs, etc. are reduced to their lemma for functor naming.*

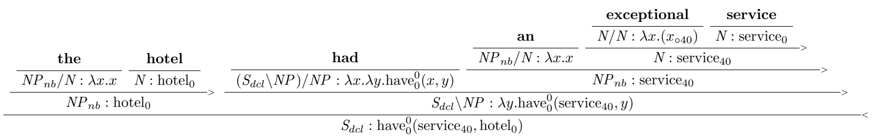
exceptional
service
$N/N : \lambda x.(x_{\circ 40})$
$N : \text{service}_0$
an
$NP_{nb}/N : \lambda x.x$
$N : \text{service}_{40}$
the
hotel
had
$NP_{nb}/N : \lambda x.x$
$N : \text{hotel}_0$
$(S_{dcl}\backslash NP)/NP : \lambda x.\lambda y.\text{have}^0_0(x, y)$
$NP_{nb} : \text{service}_{40}$
$NP_{nb} : \text{hotel}_0$
$S_{dcl}\backslash NP : \lambda y.\text{have}^0_0(\text{service}_{40}, y)$
$S_{dcl} : \text{have}^0_0(\text{service}_{40}, \text{hotel}_0)$
>

Figure 3.10: Deduction of simple declarative sentence with semantics.

■

Example 3.4 *Figure 3.11 shows the deduction proof for the sentence "the breakfast that the restaurant served daily was excellent" including semantics, and demonstrates variations of all combinator rules introduced. Most interesting is the correct binding between "breakfast" and "excellent", even though these are far from each other in the surface structure of the sentence. Furthermore the adverb "daily" correctly modifies the transitive verb "served", even though the verb is missing it's object since it paritipates in a relative clause.*

When the relative pronoun binds the dependent clause to the main clause, it "closes" it for further modification by changing the impact argument of the functor inflicted by the verb of the dependent clause, such that further modification will impact the subject of the main clause.

■

As demonstrated by the examples, the CCG grammar formalism has successfully adapted to the area of sentiment analysis, and is indeed capable of capturing the long distance dependencies that pure machine learning techniques struggles with.

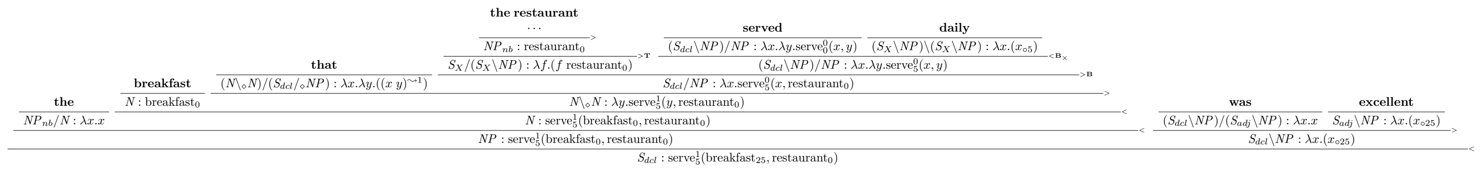
the restaurant
...
NP_{nb} : restaurant$_0$
$S_X/(S_X\backslash NP) : \lambda f.(f\ \text{restaurant}_0)$
served
daily
$(S_{dcl}\backslash NP)/NP : \lambda x.\lambda y.\text{serve}^0_0(x,y)$
$(S_X\backslash NP)\backslash(S_X\backslash NP) : \lambda x.(x_{\circ 5})$
$(S_{dcl}\backslash NP)/NP : \lambda x.\lambda y.\text{serve}^0_5(x,y)$
that
$(N\backslash_\diamond N)/(S_{dcl}/_\diamond NP) : \lambda x.\lambda y.((x\ y)^{\rightsquigarrow 1})$
$S_{dcl}/NP : \lambda x.\text{serve}^0_5(x, \text{restaurant}_0)$
breakfast
N : breakfast$_0$
$N\backslash_\diamond N : \lambda y.\text{serve}^1_5(y, \text{restaurant}_0)$
the
$NP_{nb}/N : \lambda x.x$
$N : \text{serve}^1_5(\text{breakfast}_0, \text{restaurant}_0)$
$NP : \text{serve}^1_5(\text{breakfast}_0, \text{restaurant}_0)$
was
excellent
$(S_{dcl}\backslash NP)/(S_{adj}\backslash NP) : \lambda x.x$
$S_{adj}\backslash NP : \lambda x.(x_{\circ 25})$
$S_{dcl}\backslash NP : \lambda x.(x_{\circ 25})$
$S_{dcl} : \text{serve}^1_5(\text{breakfast}_{25}, \text{restaurant}_0)$
$>$
$>\mathbf{T}$
$<\mathbf{B}_\times$
$>\mathbf{B}$
$<$

Figure 3.11: Sentiment of sentence with long distance dependencies.

Chapter 4

Lexicon acquisition and annotation

The tiny languages captured by "handwritten" lexicons, such as the one demonstrated in the previous chapter, are obviously not a sane option when the grammar is to accept a large vocabulary and a wide range of sentence structures.

In order to use the presented model on actual data, the acquisition of a wide covering lexicon is crucial. Initially considerable effort was made to *try* to build a CCG lexicon from a *POS-tagged corpus* (part-of-speech-tagged corpus). A POS-tagged corpus is simply a corpus where each token is tagged with a POS-tag, e.g. noun, verb, etc. There is no deep structure in such a corpus as opposed to a *treebank*. This approach turned up to have extensive issues as a result of this lack of structure, some of which are detailed in Appendix A which succinctly describes the process of the attempt.

There exists some wide covering CCG lexicons, most notable *CCGbank*, compiled by Hockenmaier and Steedman [2007] by techniques presented by [Hockenmaier, 2003]. It is essentially a translation of almost the entire Penn Treebank [Marcus *et al.*, 1993], which contains over 4.5 million tokens, and where each sentence structure has been analyzed in full and annotated. The result is a highly covering lexicon, with some entries having assigned over 100 different lexical categories. Clearly such lexicons only constitutes half of the previous defined $\mathcal{L}_{\text{CCG}}$ map, i.e. only the lexical categories, Γ. The problem of obtaining a full lexicon, that also yields semantics expressions, is addressed in the next section. It is also worth mentioning that since

Baldridge's [2002] work on modalities only slightly predates Hockenmaier's [2003] work on CCGBank, the CCGBank does not incorporate modalities[1]. However more unfortunately is that CCGBank is not *free to use*, mainly due to license restrictions on the Penn Treebank.

What might not be as obvious is that besides obtaining a wide-covering lexicon, $\mathcal{L}_{\text{CCG}}$, an even harder problem is for some text T to select the *right* tagging from $\mathcal{L}_{\text{CCG}}(w)$ for each token $w \in T$. Hockenmaier and Steedman [2007] calculate that the expected number of lexical categories per token is 19.2 for the CCGBank. This mean that an exhaustive search of even a short sentence (seven tokens) is expected to consider over 960 million ($19.2^7 \approx 961\,852\,772$) possible taggings. This is clearly not a feasible approach, even if the parsing can explore all possible deductions in polynomial time of the number of possible taggings. The number of lexical categories assigned to each token needs to be reduced, however simple reductions as just assigning the most frequent category observed in some training set (for instance CCGBank) for each token is not a solution. This would fail to accept a large amount of valid sentences, simply because it is missing the correct categories.

4.1 Maximum entropy tagging

Clearly a solution need to base the reduction on the setting of the token, i.e. in which *context* the token appears. Clark [2002] presents a machine learning approach based on a *maximum entropy model* that estimate the probability that a token is to be assigned a particular category, given the *features* of the local context, e.g. the POS-tag of the current and adjacent tokens, etc. This is used to select a subset of possible categories for a token, by selecting categories with a probability within a factor of the category with highest probability. Clark shows that the average number of lexical categories per token can be reduced to 3.8 while the parser still recognize 98.4% of unseen data. Clark and Curran [2007] presents a complete parser, which utilizes this tagging model, and a series of (log-linear) models to speed-up the actual deduction once the tagging model has assigned a set of categories to each token. What maybe even more interesting is that models trained on the full CCGBank, along with toolchain to use them (called the C&C tools), can be licensed freely for education or research purposes. For this reason it was chosen to use these models and tools.

Furthermore, even though the models neither incorporates modalities, since they are trained on the CCGBank, the deduction models solve many of these problems, since a more plausible deduction (i.e. a deduction seen more frequent in the CCGBank)

[1]There does exists another project, OpenCCG, started by Baldridge, which actually does incorporate modalities, but it has little documentation and was therefore not deemed mature enough.

always will suppress other less plausible deductions. Special care are taken about coordination, so neither here seems the lack of modalities to yield significant issues. Furthermore since the tagging models are based on trained data, which also can contain minor grammatical errors and misspellings, it is still able to assign categories to lexical entries even though they might be incorrect spelled or of wrong form.

4.2 Annotating the lexicon

The output from the C&C toolchain can be printed in various formats, including Prolog, which was considered the closest to the presented model, as it, given some set of tokens, $w_1, \dots, w_n$, simply returns a lexicon and a deduction. An illustrative output for the tokens "the service was great" is given in Figure 4.1. In Chapter 5 more details on the actual format and the processing of it is given.

$$\begin{aligned}
\alpha_1 &\equiv \textbf{the} : \text{the}_{\text{DT}} \models NP_{\text{nb}}/N \\
\alpha_2 &\equiv \textbf{service} : \text{service}_{\text{NN}} \models N \\
\alpha_3 &\equiv \textbf{was} : \text{be}_{\text{VBD}} \models (S_{\text{dcl}}\backslash NP)/(S_{\text{adj}}\backslash NP) \\
\alpha_4 &\equiv \textbf{great} : \text{great}_{\text{JJ}} \models S_{\text{adj}}\backslash NP
\end{aligned}$$

(a) Lexicon

$$\dfrac{\dfrac{\alpha_1\ \alpha_2}{NP_{\text{nb}}}{>} \qquad \dfrac{\alpha_3\ \alpha_4}{S_{\text{dcl}}\backslash NP}{>}}{S_{\text{dcl}}}{<}$$

(b) Deduction

Figure 4.1: Illustration of output from the C&C toolchain.

Clearly, deductions in the style previously presented is trivially obtained by substituting the axioms placeholders with the lexicon entries associated. The C&C toolchain also has a build-in morphological analyzer which allow the lexicon to provide the *lemma* of the tokens, as well as its POS-tag[2]. Both of these will be proven convenient later.

There is however one essential component missing from the lexicon, namely the semantic expressions. However due to the *Principle of Categorial Type Transparency* it is known exactly *what* the types of the semantic expressions should be. There are currently a total of 429 different tags in the C&C tagging model, thus trying to handle each of these cases individually is almost as senseless choice as trying to manually construct the lexicon, and certainly not very robust for changes in the lexical categories. The solution is to handle some cases that need special treatment, and then use a generic annotation algorithm for all other cases. Both the generic and the special case algorithms will be a transformation $(\mathcal{T}, \Sigma^\star) \to \Lambda$, where the first argument is the type, $\tau \in \mathcal{T}$, to construct, and the second argument is the lemma, $\ell \in \Sigma^\star$, of

[2]Since the C&C models are trained on CCGBank, which in turn are a translation of The Penn Treebank (PTB), the POS-tag-set used is equivalent to that of PTB cf. [Marcus *et al.*, 1993].

the lexicon entry to annotate. Since the special case algorithms will fallback to the generic approach, in case preconditions for the case are not met, it is convenient to start with the generic algorithm, $\mathcal{U}_{\text{GEN}}$, which is given by Definition 4.1.

Definition 4.1 The *generic semantic annotation algorithm*, $\mathcal{U}_{\text{GEN}}$ (4.1), for a type τ and lemma ℓ is defined by the auxiliary function $\mathcal{U}'_{\text{GEN}}$, which takes two additional arguments, namely an infinite set of variables $\mathcal{V}$ cf. Definition 3.2, and an ordered set of sub-expressions, (denoted A), which initially is empty.

$$\mathcal{U}_{\text{GEN}}(\tau, \ell) = \mathcal{U}'_{\text{GEN}}(\tau, \ell, \mathcal{V}, \emptyset) \tag{4.1}$$

If τ is primitive, i.e. $\tau \in \mathcal{T}_{\text{prim}}$, then the generic algorithm simply return a functor with name ℓ, polarity and impact argument both set to 0, and the ordered set A as arguments. Otherwise there must exist unique values for $\tau_\alpha, \tau_\beta \in \mathcal{T}$, such that $\tau_\alpha \to \tau_\beta = \tau$, and in this case the algorithm return an abstraction of τ_α on variable $v \in V$, and recursively generates an expression for τ_β.

$$\mathcal{U}'_{\text{GEN}}(\tau, \ell, V, A) = \begin{cases} \ell^0_0(A) : \tau & \text{if } \tau \in \mathcal{T}_{\text{prim}} \\ \lambda v.\mathcal{U}'_{\text{GEN}}(\tau_\beta, \ell, V \setminus \{v\}, A') : \tau & \text{otherwise, where:} \end{cases}$$

$$\begin{aligned} v &\in V \\ \tau_\alpha \to \tau_\beta &= \tau \\ A' &= \begin{cases} A[e : \tau_\alpha \to \tau_\gamma \mapsto ev : \tau_\gamma] & \text{if } e' : \tau_\alpha \to \tau_\gamma \in A \\ A[e : \tau_\gamma \mapsto ve : \tau_\delta] & \text{if } \tau_\gamma \to \tau_\delta = \tau_\alpha \wedge e' : \tau_\gamma \in A \\ A \cup \{v : \tau\} & \text{otherwise} \end{cases} \end{aligned}$$

The recursive call also removes the abstracted variable v from the set of variables, thus avoiding recursive abstractions to use it. The ordered set of sub-expressions, A, is modified cf. A', where the notation $A[e_1 : \tau_1 \mapsto e_2 : \tau_2]$ is the substitution of all elements in A of type τ_1 with $e_2 : \tau_2$. Note that e_1 and τ_1 might be used to determine the new value and type of the substituted elements. Since the two conditions on A' are not mutual exclusive, if both apply the the first case will be selected. The value of A' can be explained in an informal, but possibly easier to understand, manner:

- If there is a least one function in A, that takes an argument of type τ_α, then apply v (which is known to by of type τ_α) to all such functions in A.
- If the type of v itself is a function (i.e. $\tau_\gamma \to \tau_\delta = \tau_\alpha$), and A contains at least one element that can be used as argument, then substitute all such arguments in A by applying them to v.
- Otherwise, simply append v to A.

■

The get a little familiar with how the generic semantic annotation algorithm works, Example 4.1 shows the computation of some types and lemmas.

Example 4.1 *Table 4.1 shows the result of applying $\mathcal{U}_{\text{GEN}}$ on some lemmas and types. The result for a noun as "room" is simply the zero-argument functor of the same name. The transitive verb "provide" captures two noun phrases, and yields a functor with them as arguments.*

More interesting is the type for the determiner "every", when used for instance to modify a performance verb, as shown in Figure 4.2. It starts by capturing a noun, then a function over noun phrases, and lastly a noun phrase. The semantic expression generated for this type is a functor, with simply the noun as first argument, and where the second argument is the captured function applied on the noun phrase.

Lemma	Type	Generic semantic expression
room	τ_{N}	room_0^0
provide	$\tau_{\text{NP}} \to \tau_{\text{NP}} \to \tau_{\text{S}}$	$\lambda x.\lambda y.\text{provide}_0^0(x, y)$
every	$\tau_{\text{N}} \to (\tau_{\text{NP}} \to \tau_{\text{S}}) \to (\tau_{\text{NP}} \to \tau_{\text{S}})$	$\lambda x.\lambda y.\lambda z.\text{every}_0^0(x, y\ z)$

Table 4.1: Some input/output of generic annotation algorithm

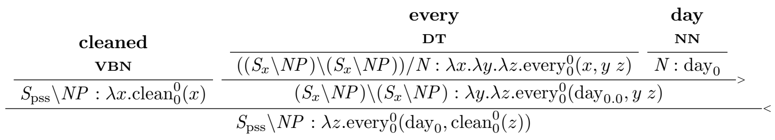
cleaned
VBN
$S_{\text{pss}}\backslash NP : \lambda x.\text{clean}_0^0(x)$
every
DT
$((S_x\backslash NP)\backslash(S_x\backslash NP))/N : \lambda x.\lambda y.\lambda z.\text{every}_0^0(x, y\ z)$
day
NN
$N : \text{day}_0$
>
$(S_x\backslash NP)\backslash(S_x\backslash NP) : \lambda y.\lambda z.\text{every}_0^0(\text{day}_{0.0}, y\ z)$
<
$S_{\text{pss}}\backslash NP : \lambda z.\text{every}_0^0(\text{day}_0, \text{clean}_0^0(z))$

Figure 4.2: Complex determiner modifying performance verb.

■

Clearly the generic algorithm does not provide much use with respect to extracting the sentiment of entities in the text, i.e. it only provide some safe structures that are guaranteed to have the correct type. The more interesting annotation is actually handled by the special case algorithms. How this is done is determined by a combination of the POS-tag and the category of the entry. Most of these treatments are very simple, with the handling of adjectives and adverbs being the most interesting. The following briefly goes through each of the special case annotations.

- *Determiners* with simple category, i.e. NP/N, are simply mapped to the identity function, $\lambda x.x$. While determiners have high focus in other NLP tasks, such as determine if a sentence is valid, the importance does not seem significant in sentiment analysis, e.g. whether an opinion is stated about *an entity* or *the entity* does not change the overall polarity of the opinion bound to that entity.

- *Nouns* are in general just handled by the generic algorithm, however in some cases of multi-word nouns, the sub-lexical entities may be tagged with the category N/N. In these cases the partial noun is annotated with a list structure, that eventually will capture the entire noun, i.e. $\lambda x.\langle \mathcal{U}_{\textsc{gen}}(\tau_{\textsc{n}}, \ell), x\rangle$, where ℓ is the lemma of the entity to annotate.

- *Verbs* are just as nouns in general handled by the generic algorithm, however *linking verbs* is a special case, since they relate the subject (i.e. an entity) with one or more *predicative adjectives*. Linking verbs have the category $(S_{\text{dcl}}\backslash NP)/(S_{\text{adj}}\backslash NP)$, and since the linked adjectives directly describes the subject of the phrase such verbs are simply annotated with the identity function, $\lambda x.x$.

- *Adjectives* can have a series of different categories depending on how they participate in the sentence, however most of them have the type $\tau_\alpha \rightarrow \tau_\beta$, where $\tau_\alpha, \tau_\beta \in \mathcal{T}_{\text{prim}}$. These are annotated with the *change* of the argument, i.e. $\lambda x.x_{\circ j}$, where j is a value determined based on the lemma of the adjective. Notice that this assumes implicit type conversion of the parameter from τ_α to τ_β, however since these are both primitive, this is a sane type cast. Details on how the value j is calculated are given in Section 4.4.

- *Adverbs* are annotated in a fashion closely related to that of adjectives. However the result might either by a *change* or a *scale*, a choice determined by the lemma: normally adverbs are annotated by the change in the same manner as adjectives, however *intensifiers* and *qualifiers*, i.e. adverbs that respectively strengthens or weakens the meaning, are scaled. Section 4.5 gives further details on how this choice is made. Finally special care are taken about negating adverbs, i.e. “not”, which are scaled with a value $j = -1$.

- *Prepositions* and *relative pronouns* need to change the impact argument of captured partial sentences, i.e. *preposition phrases* and *relative clauses*, such that further modification should bind to the subject of the entire phrase as were illustrated by Example 3.4.

- *Conjunctions* are annotated by an algorithm closely similar to $\mathcal{U}_{\textsc{gen}}$, however instead of yielding a functor of arguments, the algorithm yields a list structure. This allow any modification to bind on each of the conjugated sub-phrases.

4.3 Semantic networks

An understanding of the domain of the review is needed in order to reason about the polarity of the entities present in the text to analyse. For this purpose the concept of *semantic networks* is introduced. Formally a semantic network is a structure cf. Definition 4.2.

Definition 4.2 a semantic network is a quadruple (L, S, R, M) where:

- $L \subset \Sigma^\star$ is the set of lexical units recognized by the network.
- S is the set of *semantic concepts* in the network.
- R is a set of binary relations on S where the relation $r \in R$ describes links between semantic concepts, i.e. $r \subset S \times S$.
- M is a mapping from lexical units to a set of semantic entities that the lexical unit can entail, i.e. $M : L \to \mathcal{P}(S)$.

■

Notice that S and R constitutes a set of graphs, i.e. for each relation $r \in R$ the graph (S, r). The graph is undirected if r is *symmetric*, and directed if r is *asymmetric*. An illustrative example of such a graph, denoting a relation in an extract of a semantic network of adjective concepts is given in Figure 4.3. Since a lexical unit might entail many different concepts cf. Definition 4.2, semantic concepts are described by the set of lexical unit that can entail it, each denoted with the index that uniquely identifies that meaning of the lexical unit. For instance in Figure 4.3 the 4th meaning of *olympian* entail the same semantic concept as the 1st meaning of both *exceptional* and *exceeding*.

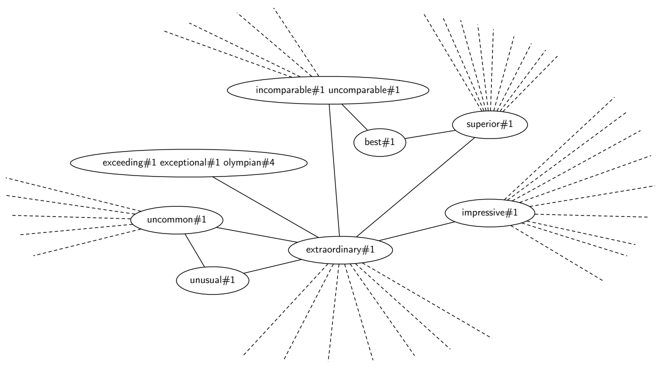
incomparable#1 uncomparable#1
superior#1
best#1
exceeding#1 exceptional#1 olympian#4
impressive#1
uncommon#1
extraordinary#1
unusual#1

Figure 4.3: Illustration of relation in a semantic network.

The concrete semantic network used is WordNet, originally presented by Miller [1995], and later presented in depth by Fellbaum [1998]. Like it was the issue when acquiring a lexicon for the syntactic analysis, the availability of free semantic networks is very sparse. It has not been possible to find other competing networks with a coverage close to that of WordNet, and thus the decision of using WordNet is really based on it being the only choice. With that said, it is argued that WordNet is a quite extensive semantic network, which in its most recent revision (3.0) contains a relatively large number of semantic concepts, namely $|S| \approx 117\,000$, while the number of lexical units recognized may be argued is not as covering as ideally, namely $|L| \approx 155\,000$.

WordNet contains a variety of relations, R, however for the purpose of calculating sentiment polarity values, only the following were considered interesting:

- The *similar*-relation, r_{similar}, links *closely similar* semantic concepts, i.e. concepts having almost the synonymies mensing in most contexts. The relation is present for most concepts entailed by adjectives.
- The *see also*-relation, $r_{\text{see-also}}$, links *coarsely similar* semantic concepts, i.e. concepts having a clear different meaning, but may be interpreted interchangeably for some contexts. Figure 4.3 on the preceding page shows an example of exactly the *see also*-relation.
- The *pertainym*-relation, $r_{\text{pertainym}}$, links the adjectives from which an adverb was derived, e.g. *extreme* is the pertainym of *extremely*.

4.4 Sentiment polarity of adjectives

The approach used to calculate sentiment polarity values for adjectives is very similar to the one presented by Simančík and Lee [2009], namely a domain specific set of positive and negative *seed concepts* are identified, respectively S_{pos} and S_{neg}. Each semantic concept in the sets can be described by a lexical unit and the specific index for that lexical unit that entail the desired concept, e.g. as shown in (4.2) and (4.3).

$$S_{\text{pos}} = \{\text{clean\#1}, \text{quiet\#1}, \text{friendly\#1}, \text{cheap\#1}\} \tag{4.2}$$
$$S_{\text{neg}} = \{\text{dirty\#1}, \text{noisy\#1}, \text{unfriendly\#1}, \text{expensive\#1}\} \tag{4.3}$$

The graph G_{adj}, given by (4.4), is considered to calculate the sentiment polarity change inflicted by the adjective. Notice that the graph includes both the *similar*-relation and the *see also*-relation, as is desired to be able to reason about as large a set of adjectives as possible, and thus both closely and coarsely related concepts are considered.

$$G_{\text{adj}} = (S, r_{\text{similar}} \cup r_{\text{see-also}}) \tag{4.4}$$

Given an adjective with lemma $\ell \in \Sigma^\star$, the set of semantic concepts that this adjective may entail, $M(\ell)$, should provide the basis for the sentiment polarity change inflicted by the adjective. The overall approach is to base the polarity change on the *distances* between the concepts yielded by $M(\ell)$ and the seed concepts in G_{adj}. However, as the system only has very limited domain knowledge, namely S_{pos} and S_{neg}, there are no sure way of choosing which of the semantic concepts yielded by $M(\ell)$ is the "right" interpretation of ℓ. Considering all of the possible concepts yielded by $M(\ell)$ nigher seem like a sane choice, since this would evidently flatten the polarities and end in vague results. The method used to solve this semantic ambiguity follows from the rational assumption that adjectives stated in the texts presumably are to be interpreted within the domain given by the seed concepts. Thus concepts from $M(\ell)$ that are strongly related to one or more seed concepts should be preferred over weaklier related concepts. The solution is to select the n *closest* relations between $M(\ell)$ and respectively S_{pos} and S_{neg}, thus reasoning greedily positively, respectively greedily negatively, about ℓ. The *multiset* of all distances to either seed set for some lemma ℓ is given by $D_{\text{adj}}(\ell, S')$, where S' is either S_{pos} or S_{neg} cf. (4.5), and where d_{adj} is the distance function for G_{adj}. The sub-multiset of $D_{\text{adj}}(\ell, S')$ which contains the minimal n values is denoted $D^n_{\text{adj}}(\ell, S')$.

$$D_{\text{adj}}(\ell, S') = [d_{\text{adj}}(s, s') \mid s \in M(\ell), s' \in S'] \qquad \text{where } S' \in \{S_{\text{pos}}, S_{\text{neg}}\} \tag{4.5}$$

Finally the sentiment polarity for an adjective with lemma ℓ, denoted $p_{\text{adj}}(\ell)$, is then given by the *difference* of the sums of the n minimal *normalized distances* cf. (4.6), where N is a normalization factor, ensuring that $p_{\text{adj}}(\ell) \in [-\omega; \omega]$. This follows the intuition, that positive adjectives are *closer* to S_{pos} than S_{neg} and vice-verse.

$$p_{\text{adj}}(\ell) = \left(\sum_{j \in D^n_{\text{adj}}(\ell, S_{\text{neg}})} N \cdot j \right) - \left(\sum_{j \in D^n_{\text{adj}}(\ell, S_{\text{pos}})} N \cdot j \right) \tag{4.6}$$

Clearly this intuition is only valid if $|S_{\text{pos}}| \approx |S_{\text{neg}}|$, or more precisely, the probability of some random semantic concept is close to a positive concept should be the same as the probability of it being close to a negative concept.

The semantic expression of an adverb with type $\tau_\alpha \to \tau_\alpha$ is given by $e_{\text{adj}}(\ell)$ in (4.7).

$$e_{\text{adj}}(\ell) = \lambda x.x_{\circ p_{\text{adj}}(\ell)} \tag{4.7}$$

The calculation of $p_{\text{adj}}(\ell)$ can be generalized for any semantic graph G, given its distance function d, and sets of respectively positive and negative *seed concepts* S_{p} and S_{n} cf. (4.8). This generalization will be convenient in a moment, and p_{adj} is trivially expressible in terms of p: $p_{\text{adj}}(\ell) = N \cdot p(d_{\text{adj}}, S_{\text{pos}}, S_{\text{neg}}, \ell)$.

$$p(d, S_{\text{p}}, S_{\text{n}}, \ell) = \left(\sum_{j \in D_d^n(\ell, S_{\text{n}})} j \right) - \left(\sum_{j \in D_d^n(\ell, S_{\text{p}})} j \right) \tag{4.8}$$

where:

$$D_d(\ell, S') = [d(s, s') \mid s \in M(\ell), s' \in S']$$

4.5 Sentiment polarity of adverbs

The approach for calculating sentiment polarity values for adverbs is very similar to the approach used for adjectives. In fact, since WordNet does not define neither r_{similar} nor $r_{\text{see-also}}$ on adverbs the basic approach is simply to lookup the adjective from which the adverb is derived from, if any, using $r_{\text{pertainym}}$. However since adverbs might also *intensify* or *qualify* the meaning of a verb, adjective, or another adverb, some special treatment are presented for this. Analog to the positive and negative concepts, sets of respectively intensifying and qualifying seed adjectives are stated, e.g. (4.9) and (4.10).

$$S_{\text{intensify}} = \{\text{extreme\#1}, \text{much\#1}, \text{more\#1})\} \tag{4.9}$$

$$S_{\text{qualify}} = \{\text{moderate\#1}, \text{little\#1}, \text{less\#1})\} \tag{4.10}$$

Recall from Section 4.2 that intensifiers and qualifiers are *scaling* the value of the verb, adjective or adverb they modify. To calculate the factor of which an intensifier or qualifier inflicts, the graph G_{scale}, given by (4.11), is considered. The graph only contains the *similar*-relation, since related intensifiers, respectively qualifiers, seem to be captured most precisely by only considering *closely similar* adjectives to the selected seeds cf. [Simančík and Lee, 2009]. Also notice that, unlike S_{pos} and S_{neg}, these sets does not rely on the domain, as they only strengthens or weakens domain specific polarities.

$$G_{\text{scale}} = (S, r_{\text{similar}}) \tag{4.11}$$

The value of a polarity scaling is then given by the exponential function (4.12) with range $\left[\frac{1}{2}; 2\right]$, i.e. the value of $p(d_{\text{scale}}, S_{\text{intensify}}, S_{\text{qualify}}, \ell)$ is normalized using N such it yields the range $[-1; 1]$. Thus a strong intensifier can double the sentiment polarity value of the unit it modifies, analogously a strong qualifier can reduce it by half.

$$p_{\text{scale}}(\ell) = 2^{N \cdot p(d_{\text{scale}}, S_{\text{intensify}}, S_{\text{qualify}}, \ell)} \tag{4.12}$$

Whether an adverb is considered an intensifier/qualifier or an normal adverb is determined by the value of $p_{\text{scale}}(\ell')$, where ℓ' is the pertainym of the adverb. The semantic expression of an adverb with type $\tau_\alpha \to \tau_\alpha$ is given by $e_{\text{adverb}}(\ell)$ in (4.13). If $p_{\text{scale}}(\ell') \neq 1$ then the adverb is considered as an intensifier/qualifier and scales the lexical unit it inflicts, otherwise if the adverb is an derivation of an adjective it simply changes the inflicted unit with the value of the adjective. Finally the adverb can be one of a small set of predefined negatives, L_{neg}, in this case the polarity of the inflicted unit is flipped. Otherwise the adverb is discarded by simply being annotated with the identity function.

$$e_{\text{adverb}}(\ell) = \begin{cases} \lambda x.x_{\bullet p_{\text{scale}}(\ell')} & \text{if } M(\ell) \times M(\ell') \subset r_{\text{pertainym}} \text{ and } p_{\text{scale}}(\ell') \neq 1 \\ \lambda x.x_{\circ p_{\text{adj}}(\ell')} & \text{if } M(\ell) \times M(\ell') \subset r_{\text{pertainym}} \\ \lambda x.x_{\bullet -1} & \text{if } \ell \in L_{\text{neg}} \\ \lambda x.x & \text{otherwise} \end{cases} \tag{4.13}$$

4.6 Completing the analysis

All the components needed in order to calculate the sentiment for entities in a text have now been presented. The final step is to connect the components in a pipeline, allowing to compute the complete analysis originally presented originally in the start of Chapter 2. Recall that the presented analysis in a calculation of type (2.1), repeated here for convenience.

$$\mathcal{A} : \Sigma^\star \to \mathbb{E} \to [-\omega; \omega] \tag{2.1}$$

After initial preprocessing (i.e. tokenization), the text is processed by the lexical-syntactic analysis using the C&C toolchain. The resulting lexicon is annotated with semantic expressions using the algorithms and semantic network structures described previous in this chapter. The deduction proof is then reconstructed based on the partial (i.e. syntax only) proof from the C&C toolchain. The resulting deduction proof uses the combinator rules over both lexical and semantic expressions presented in Chapter 3. During the reconstruction process, semantic expressions are reduced using the rewrite rules presented in Section 3.4. If these steeps completes successfully the deduction proof should yield a conclusion for a sentence with a *closed* semantic expression. By *closed* semantic expression is meant that the expression is only consisting of functors and sequences. Details and examples of what might fail during this process will be elaborated upon evaluation of the system in Chapter 6.

The resulting semantic expression, e, is then inspected by the auxiliary sentiment extraction function $\mathcal{E} : \mathbb{E} \rightarrow \Lambda \rightarrow \mathcal{P}([-\omega; \omega])$ cf. Definition 4.3.

DEFINITION 4.3 The extraction of sentiment, for a given *subject of interest*, s, from a given semantic expression, e, is defined recursively by the function $\mathcal{E}$ cf. 4.14. If the expression is a functor with *same* name as the *subject of interest*, then the sentiment value of this functor is included in the resulting set, otherwise the value is simply discarded. In both cases the function is recursively applied to the subexpressions of the funtor. If the expression is a sequence, the function is simply recursively applied to all subexpressions in the sequence. Should the expression unexpectively include expressions of other kinds than functors and sequences, the function just yields the empty set.

$$\mathcal{E}(s, e) = \begin{cases} \{j\} \cup \bigcup_{e' \in E'} \mathcal{E}(s, e') & \text{if } e \text{ is } f_j^i(E') \text{ and } s = f \\ \bigcup_{e' \in E'} \mathcal{E}(s, e') & \text{if } e \text{ is } f_j^i(E') \text{ and } s \neq f \\ \bigcup_{e' \in E'} \mathcal{E}(s, e') & \text{if } e \text{ is } \langle E' \rangle \\ \emptyset & \text{otherwise} \end{cases} \tag{4.14}$$

■

Finally the algorithm for the complete sentiment analysis can be formulated cf. Figure 4.4. The SELECT function simply selects the strongest opinion extracted (i.e. the sentiment with largest absolute value). This is a very simple choice but are sufficient for evaluating the algorithm, which will be done in Chapter 6.

```
$\mathcal{A}(text, s)$
    $text' \leftarrow$ PREPROCESS$(text)$
    $(lexicon, proof) \leftarrow$ SYNTAXANALYSIS$(text')$
    $lexicon' \leftarrow$ ANNOTATE$(lexicon)$
    $proof' \leftarrow$ RECONSTRUCTPROOF$(lexicon', proof)$
    $(\alpha : e) \leftarrow$ CONCLUSION$(proof')$
    return SELECT$(\mathcal{E}(s, e))$
```

Figure 4.4: Algorithm for sentiment analysis.

CHAPTER 5

Implementation

In order to demonstrate the logical approach, introduced in the previous chapters, a *proof of concept* system was implemented. In the following sections key aspects of the implementation of this system will be presented. A complete walk-though will not be presented, but the complete source code for the implementation is available in Appendix **??**. Also notice that code segments presented in this chapter maybe simplified from the source code to ease understanding. For instance the C&C-toolchain uses some additional primitive categories to handle conjunctions, commas and punctuations that are not consider theoretical or implementationwise interesting, as they are translatable to the set of categories already presented. In the actual implementation of the proof of concept system this is exactly what is done, once the output from the C&C-toolchain has been parsed.

It was chosen to use the purely functional, non-strict programming language *Haskell* for implementing the proof of concept system. The reason Haskell, specifically the *Glasgow Haskell Compiler*, was chosen as programming language and platform, was i.a. its ability to elegantly and effectively implement a parser for the output of the C&C-toolchain. Data structures are like in many other functional languages also possible to state in a very succinct and neat manner, which allow Haskell to model the extended semantics presented in Section 3.4, as well as any other structure presented, e.g. deduction proofs, lexical and phrasal categories, etc.

5.1 Data structures

Data structures are stated in Haskell by the means of *type constructors* and *data constructors*. To model for instance lexical and phrasal categories the two infix operators, `:/` and `:\` are declared (using `/` and `\` was not considered wise, as `/` is already used for devision by the Haskell Prelude) as shown in Figure 5.1. The *agreement* of an primitive category is simply a set of features cf. Section 3.3, which is easiest modeled using the list structure. As features are just values from some language specific finite set they are simply modeled by *nullary data constructors*. One might argue that features have *different* types, e.g. person, number, gender, etc. However it is convenient to simply regard all features as being of the *same* type, a model borrowed from van Eijck and Unger [2010, chap. 9].

```haskell
infix 9 :/   -- Forward slash operator
infix 9 :\   -- Backward slash operator

type Agreement = [Feature]

data Category = S Agreement                -- Sentence
              | N Agreement                -- Noun
              | NP Agreement               -- Noun Phrase
              | PP Agreement               -- Preposision Phrase
              | Category :/ Category       -- Forward slash
              | Category :\ Category       -- Backward slash

data Feature = FDcl | FAdj | FNb | FNg | ...
```

Figure 5.1: Example of declaring the data structure for categories.

The code shown in Figure 5.1 is really all what is needed to represent the syntactic structure of categories. Another illustration of one of the data structural advantages of using a functional programming language is shown in Figure 5.2. Notice how the declaration of the syntax for the semantic expressions is completely analog to the formal syntax given in Definition 3.2 and 3.3, with the exception that the implemented syntax is untyped. The reason why types are omitted from the implemented model of semantic expressions is simply that they are always accompanied by a category, and thus the type of the expression is trivially obtainable when needed.

```haskell
data SExpr = Var String                       -- Variable
           | Abs String SExpr                 -- Lambda abstraction
           | App SExpr SExpr                  -- Lambda application
           | Fun String Float Int [SExpr]     -- Functor
           | Seq [SExpr]                      -- Sequence
           | ImpactChange SExpr Int           -- Impact change
           | Change SExpr Float               -- Change
           | Scale SExpr Float                -- Scale
```

Figure 5.2: Example of declaring the data structure for semantic expressions.

5.2 Reducing semantic expressions

With data structures available for representing the syntax of the semantic expressions it is time to focus on reducing the expression using the semantic rules presented in Definition 3.4. This can be easily done in a functional language by specifying a *reduction function*, i.e. a function that recursively rewrites semantic expressions based on the rules presented in the definition. By using the *pattern matching* available in Haskell, each rule can be implemented in a one-to-one manner by a function declaration that only accepts the *pattern* of that rule. For instance Figure 5.3 shows the implementation of the (FC1), (SC) and (PC) rules. A small set of additional function declarations are needed to allow reduction inside a structure that itself cannot be reduced, and finally the *identity function* matches any pattern not captured by any of the other function declarations. Notice that η-reduction was not implemented, since this rule is merely a performance enhancing rule.

```
-- (FC1)
reduce (Change (Fun f j 0 ts) j') =
  Fun f (j + j') 0 $ map reduce ts

-- (SC)
reduce (Change (Seq ts) j') =
  Seq $ map (reduce . flip Change j') ts

-- (PC)
reduce (Change (Abs x t) j') =
  Abs x $ reduce $ Change t j'
```

Figure 5.3: Example of declaring the rules for semantic expressions.

5.3 Interacting with the C&C toolchain

When processing multiple texts the most effective way of interaction with the C&C toolchain is by running a server instance of it, since this allows the toolchain to only load the trained models once (which are quite large). The server can be interacted with through a *SOAP web service* [Box *et al.*, 2000]. It seem a somewhat strange choice that Clark and Curran choose to base the communication with the server on SOAP, since the only data structure ever exchanged is single strings (the text to parse as input, and the raw *Prolog style* string as output). It was chosen not to interact directly with the web service from Haskell, since no mature SOAP libraries are currently available natively for Haskell. Instead it was chosen to use a small client program distributed along with the C&C tools, allowing communication with the SOAP web service through the clients standard input/output.

The implemented parser for the Prolog style output yielded by the C&C toolchain, presented briefly in Section 4.2, uses the Parsec library for Haskell by Leijen [2001]. Parsec is a strong monadic parser combinator, that among other things allows fast and efficient parsing of LL[1] grammars, and can thus easily capture the subset of the Prolog language used by the C&C-toolchain. Parsec differs significantly from common Yacc approaches, since it describes the grammar *directly* in Haskell, without the need of some intermediate language or processing tools.

Figure 5.4 shows the actual raw output from the C&C-toolchain that is the basis the illustration in Figure 4.1 shown back in Section 4.2. The first section of the output represents the deduction tree, while the second represents the lexicon (obviously without semantic expressions).

```
ccg(1,
 ba('S[dcl]',
  fa('NP[nb]',
   lf(1,1,'NP[nb]/N'),
   lf(1,2,'N')),
  fa('S[dcl]\NP',
   lf(1,3,'(S[dcl]\NP)/(S[adj]\NP)'),
   lf(1,4,'S[adj]\NP')))).

w(1, 1, 'the', 'the', 'DT', 'I-NP', 'O', 'NP[nb]/N').
w(1, 2, 'service', 'service', 'NN', 'I-NP', 'O', 'N').
w(1, 3, 'was', 'be', 'VBD', 'I-VP', 'O', '(S[dcl]\NP)/(S[adj]\NP)').
w(1, 4, 'great', 'great', 'JJ', 'I-ADJP', 'O', 'S[adj]\NP').
```

Figure 5.4: Raw output from the C&C toolchain.

One of the most admirable features of Parsec is its *parser combinator library*, containing a verity of bundled auxiliary functions, which allows the declaration of advanced parsers by combining smaller parsing functions. To parse for instance the categories present in both of the sections one can build an *expression parser* simply by stating the *symbol*, *precedence* and *associativity* of the operators.

Figure 5.5 shows the parser for categories. The precedence of the operators are given by the outer list in the *operator table*, while operators within the same inner list have the same precedence, which is in the case for both of the categorial infix operators. Finally a category is declared as either compound (i.e. a category expression), or as one of the four primitive categories. Notice how the parser needs to first *try* to parse *noun phrases* (NP), and then *nouns* (N), since the parser otherwise could successfully parse a noun, and then meet an unexpected "P", which would cause a parser error.

The parsing of the lexicon is considered trivial, since its structure is flat with the exception of the category. The parser for the deduction proof is simply stated by two abstract rules, unary (e.g. for capturing type-raise) and binary (e.g. for capturing functional application), along with a top-rule for choosing specific instances of the

```
pCategoryExpr :: Parser Category
pCategoryExpr = buildExpressionParser pCategoryOpTable pCategory

pCategoryOpTable :: OperatorTable Char st Category
pCategoryOpTable = [ [ op "/"  (:/) AssocLeft,
                       op "\\" (:\) AssocLeft ] ]
                   where
                     op s f a = Infix ( string s >> return f ) a

pCategory :: Parser Category
pCategory =         pParens pCategoryExpr
            <|>     (pCategory' "S"     S)
            <|> try (pCategory' "NP"    NP)
            <|>     (pCategory' "N"     N)
            <|>     (pCategory' "PP"    PP)
            <?> "category"
```

Figure 5.5: Example of parsing categorial expression.

rules. Besides this there is several rules for handling C&C's relatively verbose rules for coordination, especially in connection with commas and punctuations.

5.4 WordNet interface and semantic networks

To lookup semantic concepts and relations in the WordNet data files an open source interface library by Daumé III [2008] was used as base. However the interface was not complete, and missed critical features. For instance the library could only calculate the closure of two semantic concepts, which of cause only is possible when the relation forms a partial order, e.g. as is the case with the *hyponym/hypernym* relation and the *holonym/meronym* relation. Therefore the library has undergone significant rewrite and cleanup in order to use it for the presented purpose.

To model semantic networks another open source library was used, namely the *Functional Graph Library* (FGL). The library implements efficient functional graph representation and algorithms presented by Erwig [2001]. However transforming the relational representation of WordNet into an actual graph in the sense of FGL is somewhat tricky. The reason for this is that intended usage of the WordNet data files do not exposes S, and neither r in the form of a subset of $S \times S$, which makes good sense since this representation does not scale well with $|S|$. Instead it is intended to query using the lookup function, M, which is indexed and allows logarithmic time lookup of lexical units; likewise a relation, $\hat{r}$, is a function from one semantic concept to a set of related concepts, i.e. $\hat{r} : S \to \mathcal{P}(S)$. This structure makes querying WordNet efficient, but also allows some optimization with respect to calculating the sentiment polarity value of lexical units. Recall from Section 4.3 that the approach is to select a set of respectively positive and negative seed concepts, and then measure the difference of the sum of distances from a lexical unit to these. However instead of regarding the entire graph (S, r) only a subgraph (S', r') is considered, namely the subgraph that constitutes the *connected component* that contains all semantic concepts that are reachable from the seed concepts using the relation function $\hat{r}$. This of cause assumes that r is symmetric, which is also the case for r_{similar} and $r_{\text{see-also}}$ cf. Section 4.3.

The construction of (S', r') for some set of positive and negative seed semantic concepts, respectively S_{pos} and S_{neg}, is denoted the *unfolding* of $S_{\text{pos}} \cup S_{\text{neg}}$ using $\hat{r}$. It is done using simple depth first search with the set of initially "unvisited" semantic concepts $S_{\text{pos}} \cup S_{\text{neg}}$. The FGL requires nodes to be assigned a unique index, and it is also clearly necessary to keep track of which semantic concepts has already been assigned a node in the graph, i.e. which nodes are considered visited. Lastly the set S' and r' should be build incrementally. In an imperative programming language maintaining such mutable state is straight forward, but in a purely functional programming language such as Haskell this require somewhat advanced techniques if it should be implemented efficiently. Launchbury and Peyton Jones [1994] presents a method to allow functional algorithms to update internal state, while still externally be purely functional algorithms with absolutely no side-effects. An implementation of Launchbury and Peyton Jones's method is available in Haskell through *strict state threads* and the *strict state-transformer monad.*

Figure 5.6 shows the Haskell code for unfolding semantic graphs. The actual unfolding are done by the function `unfoldST`, which yields a state transformer with computation result of `(Map a Node, Gr a Int)`, i.e. a map from the type of items to unfold, `a`, (e.g. semantic concepts) to nodes in the FGL graph and the actual FGL graph with nodes of type `a` and edges simply with integer weights (all edges have weight 1).

Since the `ST` structure conforms to the laws of monads *state transformation computations* can be chained into a pipeline, and thus even though the code for the algorithm might look almost imperatively, it is indeed purely functional. Initially three *references* to mutable state are constructed given each of the states an initial value. The `visit` function simply "visits" an item (e.g. semantic concept): if the item has already been visited, the unique index for its corresponding node in the FGL graph is simply returned; otherwise a new unique node index is allocated and outgoing relations from this item are visited by recursive calls. Mutable states can by modified by the `modifySTRef` function, which takes a reference to the state to modify, and a *transformation* function over the state. When the depth first search finishes, the map and FGL graph are returned as non-mutable structures as the result of the computation.

```
-- | Unfold the a graph using the given relation and seeds.
unfoldG :: (Ord a) => (a -> [a]) -> [a] -> (Map a Node, Gr a Int)
unfoldG r seeds = runST $ unfoldST r seeds

-- | State trasformer for unfolding graphs.
unfoldST :: (Ord a) => (a -> [a]) -> [a] -> ST s (Map a Node, Gr a Int)
unfoldST r seeds =
  do mapRef    <- newSTRef Map.empty    -- Map from Item to Node
     nodesRef  <- newSTRef []           -- List of Node/[Edge] pairs
     idRef     <- newSTRef 0            -- Counter for indexing nodes
     -- Recursively visits n
     let visit n =
           do -- Test if n has already been visited
              test <- (return . Map.lookup n =<< readSTRef mapRef)
              case test of
                Just v  -> return v
                Nothing ->
                  do -- Get next id for this item
                     i <- readSTRef idRef
                     modifySTRef idRef (+1)
                     -- Update item/node map
                     modifySTRef mapRef (Map.insert n i)
                     -- Recursively visit related items
                     ks <- mapM visit $ r n
                     let ns = ((i,n), [(i,k,1) | k <- ks])
                     modifySTRef nodesRef (ns:)
                     return i
     -- Visit seeds
     mapM visit seeds
     -- Read results and return map/graph-pair
     list <- readSTRef nodesRef
     nodeMap <- readSTRef mapRef
     let nodes = [n | (n, _) <- list]
     let edges = concat [es | (_, es) <- list]
     return (nodeMap, mkGraph nodes edges)
```

Figure 5.6: Example of usage of strict state threads.

5.5 Overall analysis and extraction algorithm

The implemented program loads a set of review texts from a plain text file. The texts are presumed to have been preprocessed by decent tokenization cf. Section 2.1, and each line in the file should constitute a sentence. The set of positive and negative seed concepts, as well as the *subject of interest* are simply defined directly in the source code for the program, since it is intended simply as a proof of concept system.

The program prints the result of the analysis (if any) for each of the input sentences. The implemented extraction algorithm also makes a very primitive pronoun resolution, namely that the third person pronouns *it*, *they* and *them* are simply identified as the subject of interest.

CHAPTER 6

Evaluation

This chapter evaluates the presented logical approach for sentiment analysis, specifically the *proof of concept* implementation presented in the previous section. In order to truly classify the capabilities of the system, real test-data are needed. In this chapter a test data set is introduced, and results from evaluation of the system on this set are presented. The results are explained, and it is considered whether the robustness and the correctness of the solution is significantly high enough for real applications.

6.1 Test data set

There are several *free to use* labeled review data set available (most of them consisting of movie reviews for some reason). However recall that most research have focused on classifying sentiment on document, sentence or simply on word level, but not on entity level cf. Section 1.4 on page 7. This mean that it unfortunately has not been possible to find any free data set there was labeled on entity level.

The test data set chosen for evaluation of the system is the *Opinosis data set* [Ganesan *et al.*, 2010]. The data set consists of approximately 7000 texts from actual user reviews on a number of different topics. The topics are ranging over different product and services, from consumer electronics (e.g. GPS navigation, music players, etc.) to hotels and restaurants. They are harvested from several online resellers and service providers, including i.a. *Amazon*[1] and *TripAdvisor*[2]. The data set is neither labeled on entity level (or any other level for that matter), since it originally was used for evaluating an *automatic summarization* project by Ganesan *et al.* However the source of the reviews was one of the main reasons to use *Opinosis Dataset* for the evaluation, since it was one of the original goals of the project, that a solution could process real data.

After a coarse review of the data set it is safe to argue that the data set chosen for evaluation indeed includes all of the problematics discussed in Section 2.2 on page 14. Since the data set is unlabeled it was chosen to label a small subset of it in order to measure the robustness and the correctness of the presented solution. To avoid biases toward how the proof of concept system analyzes text the labeling was performed independently by two individuals which had no knowledge of how the presented solution processes texts. As the example texts throughout this thesis might have hinted, the subset chosen was from the set of hotel and restaurant reviews. It was, of cause, a subset that had not previously been used to test the implementation during development, however fixing the evaluation on the domain of hotel and restaurant reviews allowed the choice of relevant seed concepts cf. Section 4.4. The *subject of interest* chosen for the analysis were *hotel rooms*, and the subset was thus randomly sampled from texts with high probability of containing this entity (i.e. containing any morphological form of the noun "room"). This approach were chosen, since there otherwise are no chance the proof of concept system can yield a result for the text.

The individuals were given a subset of 35 review texts, and should mark each text as either positive, negative or unknown *with respect to the given subject of interest.* Out of the 35 review text the two subject's positive/negative labeling agreed on 34 of them, while unknowns and disagreements were discarded. Thus the inter-human *concordance* for the test data set was 97.1%, which is very high, and would arguable drop if just a few more individuals were used for label annotation. The full subset samples, as well as each subjects marking is available in Table C.1 in Appendix C.

6.2 Test results

The test data set was processed by the proof of concept system and the system was able to yield a sentiment value for the "room" entity for just 38.2% of the test texts. An entity sentiment value is considered to *agree* with the human labeling, if had the correct sign (i.e. positive sentiment values agreed with positive labels, and negative values with negative labels). As mentioned it is hard to compare the results to any

[1]Amazon, http://www.amazon.com/
[2]TripAdvisor, http://www.tripadvisor.com/

baseline, since no published results has been obtainable for entity level sentiment analysis. The baseline presented here is thus a sentence-level baseline, calculated by using the Natural Language Toolkit (NLTK) for Python using a Naive Bayes Classifier (trained on movie reviews though rather than hotel reviews). The raw sentiment values calculated by the presented method are also available for each text in the test data set in Table C.1 in Appendix C. The *precision* and *recall* results for both the baseline, and the presented method are shown in Table 6.1. As seen the recall is somewhat low for the proof of concept system, which is addressed in the next section, while it is argued that precision of the system is indeed acceptable, since even humans will not reach a 100% agreement.

	Baseline	Presented method
Precision	71.5%	92.3%
Recall	44.1%	35.3%

Table 6.1: Precision and recall results for proof of concept system.

6.3 Understanding the results

To investigate the low recall, focus was turned to clarifying why the presented method only yields results for 38.2% of the test data set. The C&C toolchain was able to give a syntactic deduction proof for 94.4% of the test data set. However after closer inspection of the proofs constructed for texts in the test data set, it was discovered that approximately only half of these proofs were correct. This is of cause a major handicap for the presented method, since it is highly reliable on correct deduction proofs.

The reason the C&C toolchain behaves so inadequately is indeed expected, and thus a low racall was also expected, even though it is lower then hoped for. Recall from Section 4.1 that the C&C parser could recognize 98.4% of unseen data. However there is one major assumption if this promise should be met: the probability distribution of the input should of cause follow the same distribution as the training data that the C&C models was created from. The models were trained on *The CCGbank* [Hockenmaier and Steedman, 2007] and thereby follows the distribution of *The Penn Treebank* [Marcus *et al.*, 1993]. That this follows a different probability distribution than the Opinosis data set may not be that surprising, since the treebank consists mostly of well-written newspaper texts. To illustrate this consider Figure 6.1 which shows the probability distribution of sentence length (i.e. number of words) in: the set of hotels and restaurants reviews of Opinosis (2059 sentences); and respectively a subset of Wall Street Jounal (WSJ) corpus, which is a representative and *free to use* sample of The Penn Treebank (3914 sentences). This measure gives a clear indication that the two data sets indeed follows different probability distributions.

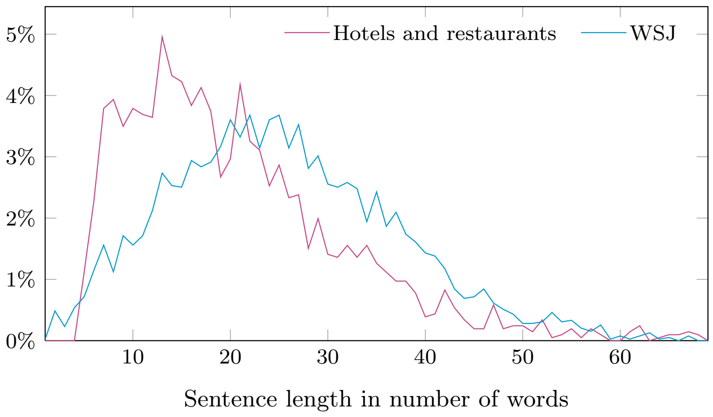

Figure 6.1: Sentence length distribution in representative subsets of Opinosis and The Penn Treebank.

At best this may however only explain half of missing results. To explain the rest focus needs to be turned to those sentences for which the C&C toolchain constructs proofs correctly, but does not yield any results. Consider Figure 6.2 which shows the deduction proof for test sentence #9, which most humans, like the two individuals used for labeling, would agree expressed a positive opinion about the rooms. However the sentiment extraction algorithm fails to capture this, even though it is worth noticing, that the semantic expression in the conclusion of the proof, i.e. $\text{furnish}^0_{95.0}(\text{room}_{0.0})$, indeed contains a positive sentiment value.

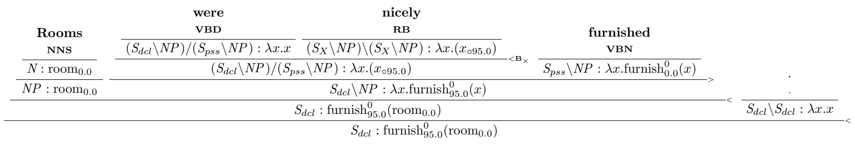
Rooms
NNS
were
VBD
nicely
RB
furnished
VBN
$N : \text{room}_{0.0}$
$NP : \text{room}_{0.0}$
$(S_{dcl}\backslash NP)/(S_{pss}\backslash NP) : \lambda x.x$
$(S_X\backslash NP)\backslash(S_X\backslash NP) : \lambda x.(x_{\circ 95.0})$
$<B_\times$
$(S_{dcl}\backslash NP)/(S_{pss}\backslash NP) : \lambda x.(x_{\circ 95.0})$
$S_{pss}\backslash NP : \lambda x.\text{furnish}^0_{0.0}(x)$
$>$
$S_{dcl}\backslash NP : \lambda x.\text{furnish}^0_{95.0}(x)$
$<$
$S_{dcl} : \text{furnish}^0_{95.0}(\text{room}_{0.0})$
$S_{dcl}\backslash S_{dcl} : \lambda x.x$
$<$
$S_{dcl} : \text{furnish}^0_{95.0}(\text{room}_{0.0})$

Figure 6.2: Sentence positive about *room*, but with no sentiment value on its functor.

In the next chapter solutions for these problems are proposed and discussed, and it is argued that even though the recall results for the test data set are unacceptable low, applications for the presented method are indeed possible given some additional effort.

Chapter 7

Discussion

The presented method for *entity level* sentiment analysis using deep sentence structure analysis has shown acceptable correction results, but inadequate robustness cf. the previous chapter.

The biggest issue for the demonstrated proof of concept system is the lack of correct syntactic tagging models. It is argued that models following a closer probability distribution of review texts than The Penn Treebank models would have improved the robustness of the system significantly. One might think, that if syntactic labeled target data are needed, then the presented logical method really suffers the same issue as machine learning approaches, i.e. *domain dependence*. However it is argued that exactly because the models needed are of *syntactic level*, and not of *sentiment level*, they really do not need to be *domain specific*, but only *genre specific*. This reduces the number of models needed, as a syntactic tagging model for reviews might cover several domains, and thus the *domain independence* of the presented method is intact.

To back this argument up, consider Figure 7.1 which shows the probability distribution of sentence lengths in two clearly different sentiment domains, namely *hotels and restaurants* (2059 samples) and *GPS navigation equipment* (583 samples). This measure gives strong indications, that a robust *syntactic level model* for either domain would also be fairly robust for the other. The same is intuitively not true for *sentiment level models*.

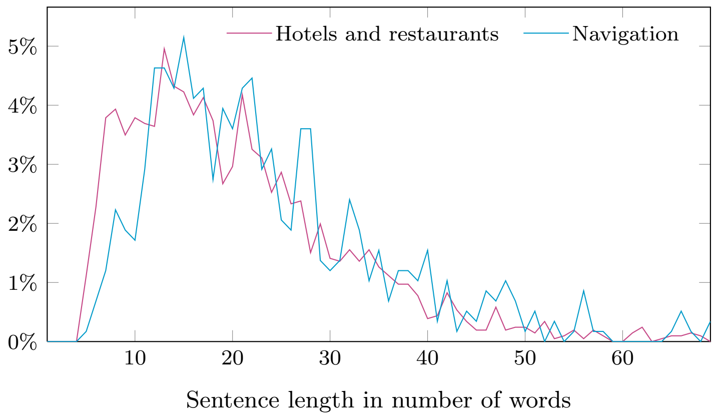

Figure 7.1: Sentence length distribution for different review topics.

An interesting experiment would have been to see how the presented method performed on such genre specific syntactic models. Building covering treebanks for each genre to train such models is an enormous task, and clearly has not been achievable in this project even for a single genre (The Penn Treebank took eight years to compile). However Søgaard [2012] presents methods for *cross-domain semi-supervised learning*, i.e. the combination of labeled (e.g. CCGBank) and unlabeled (e.g. review texts) data from different domains (e.g. syntactic genre). This allows the construction of models that utilizes the knowledge present in the labeled data, but also biases it toward the distribution of the unlabeled data. The learning accuracy is of cause not as significant as compared to learning with large amounts of labeled target data, but it can improve cases as the one presented in this thesis greatly. The reason why this was not performed in this project is partly due to it was not prioritized, and the fact that the method still assumes access to raw labeled data (e.g. CCGBank) which was not available.

The other issue identified when analyzing the low robustness was the failure of extracting sentiment values, even though they were actually present in the semantic expression yielded by the conclusion of the deduction proof. This clearly shows that the simple extraction algorithm given by Definition 4.3 is too constitutive, i.e. it turned out to be insufficient to only extract sentiment values at the atomic functor level for the *subject of interest*. However, since the knowledge is actually present, it is argued that more advanced extraction algorithms would be able to capture these cases.

The reason why more advanced extraction algorithms were not considered was that it would require more test data to validate that such advanced extraction strategies are well-behaved. Recall that it has not possible to find quality test data labeled on entity level, and it was considered too time consuming to manually construct large amounts of entity labeled data.

With these issue addressed, it is argued that the *proof of concept* system indeed shows at least the potential of the presented method, and further investment in labeled data, both syntactic tagging data, and labeled test data, would make the solution more robust.

7.1 Future work

Besides resolving the issue presented as the major cause of the low robustness, the presented method also leaves plenty of opportunities for expansion. This could include a more sophisticated pronoun resolution than the one presented in Section 5.5.

Likewise even more advanced extraction strategies could also include relating entities by the use of some of the abstract topological relations available in semantic networks, e.g. *hyponym*/*hypernym* and *holonym*/*meronym*. With such relations, a strong sentiment of the entity *room* might inflict the sentiment value of *hotel*, since *room* is a meronym of *building*, and *hotel* is a hyponym of *building*.

CHAPTER 8

Conclusion

This thesis has presented a *formal logical method* for *entity level* sentiment analysis, which utilizes *machine learning techniques* for efficient syntactic tagging. The method should be seen as an alternative to pure machine learning methods, which have been argued inadequate for capturing long distance dependencies between an entity and opinions, and of being highly dependent on the domain of the sentiment analysis.

The main aspects of method was presented in three stages:

- The *Combinatory Categorial Grammar* (CCG) formalism, presented in Chapter 3, is a modern and formal logical technique for processing of natural language texts. The semantics of the system was extended in order to apply it to the field of entity level sentiment analysis.
- In order to allow the presented method to work on a large vocabulary and a wide range of sentence structures, Chapter 4 described the usage of statistical models for syntactic tagging of the texts, after it had been argued that such an approach is the only reasonable. Algorithms for building semantic expressions from the syntactic information was presented, along with a formal method for reasoning about the sentiment expressed in natural language texts by the use of semantic networks.
- Chapter 5 presented essential details about the *proof of concept* system, which

has been fully implemented, using functional programming, in order to demonstrate and test the presented method.

Finally the presented method was evaluated against a small set of manually annotated data. The evaluation showed, that while the correctness of the presented method seem acceptably high, its robustness is currently inadequate for most real world applications as presented in Chapter 6. However it was argued in the previous chapter that it indeed is possible to improve the robustness significantly given further investment and development of the method.

Appendix A

A naive attempt for lexicon acquisition

This appendix describes the efforts that was initially made in order to acquire a CCG lexicon by *transforming* a tagged corpus, namely the *Brown Corpus*. The approach turned out to be very naive, and was dropped in favor for the C&C models trained on the CCGBank [Hockenmaier and Steedman, 2007], in turn The Penn Treebank [Marcus *et al.*, 1993].

The Brown Corpus

English is governed by convention rather than formal code, i.e. there is no regulating body like the Académie française. Instead, authoritative dictionaries, i.a. Oxford English Dictionary, describe usage rather than defining it. Thus in order to acquire a covering lexicon it is necessary to build it from large amount of English text.

The Brown Corpus was compiled by Francis and Kucera [1979] by collecting written works printed in United States during the year 1961. The corpus consists of just over one million words taken from 500 American English sample texts, with the intension of covering a highly representative variety of writing styles and sentence structures.

Notable drawbacks of the Brown Corpus include its age, i.e. there are evidently review topics where essential and recurring words used in present day writing was not coined yet or rarely used back 50 years ago. For instance does the Brown Corpus not recognize the word *internet*. However it is one of the only larger *free to use* tagged corpus available, and for this reason is was chosen for the attempt. Even early analysis showed that coverage would be disappointing, since the Brown Copus only contains 80.4% of the words of the *hotels and restaurants* subset of the *Opinosis data set* [Ganesan *et al.*, 2010] (3793 words). This mean that every 5th word would on average be a guess in the blind. However the approach was continued to see how many sentences would be possible to syntactic analyze with the lexicon.

The corpus is annotated with *part of speech* tags, but does not include the deep structure of a *treebank*. There is a total of 82 different tags, some examples are shown in Table A.1. As shown from the extract the tags include very limited information, and while some features can be extracted (e.g. tense, person) in some cases, the tagging gives no indication of the context.

Tag	Description
VB	verb, base form
VBD	verb, past tense
VBG	verb, present participle/gerund
VBN	verb, past participle
VBP	verb, non 3rd person, singular, present
VBZ	verb, 3rd. singular present

Table A.1: Extract from the Brown tagging set.

Without any contextual information the approach for translating this information into lexical categories becomes very coarse. For instance there are no way of determining whether a verb is *intransitive*, *transitive*, or *di-transitive*. The chosen method was simply to over-generate, i.e. for every verb, entries for all three types of verbs was added to the lexicon. In total 62 of such rules were defined, which produced a lexicon containing CCG categories for 84.5% of the 56 057 unique tokens present in the Brown Corpus.

Evaluating the lexicon

To evaluate the coverage of the acquired lexicon an *shift-reduce* parser was implemented in Haskell. It is not the most efficient parsing strategy, but was simple to implement and considered efficiently enough to test the lexicon. A representative sample of the *hotels and restaurants* subset of the *Opinosis data set* was selected (156 sentences). The result was that the parser only was able to parse 10.9% of the sentences. The result was very disappointing, and it was not even considered whether these even were correctly parsed. Instead it was recognized that building a CCG lexicon from only a tagged corpus is not a feasible approach. Further development of the approach was dropped and instead the component was replaced with the C&C tools [Clark and Curran, 2007].

Appendix B

Source code

Complete source code for the proof of concept solution presented can be downloaded from the url: http://www.student.dtu.dk/~s072466/msc.zip.

The following lists the essential Haskell code files for the implementation. `Main.hs` defines the extraction and analysis algorithms, i.e. $\mathcal{E}$ and $\mathcal{A}$, and also includes the main entry point for the application. The syntax and semantics for the semantic expressions are defined in `Lambda.hs`. The implementation of Combinatory Categorial Grammar (CCG) is defined by `CCG.hs`, and `Annotate.hs` defines both the generic annotation algorithm $\mathcal{U}_{\text{GEN}}$, as well as the special case annotation algorithms. Finally `Parser.hs` defines the parser for the output from the C&C toolchain. The following listing thus thereby not include the WordNet interface.

Main.hs

```haskell
{-# LANGUAGE ImplicitParams #-}

module Main where
import Data.Map (Map)
import qualified Data.Map as Map
import Data.Maybe
import Data.Char
import Control.Monad

import Parser
import CCG
import Annotate
import WordNet hiding (Word)

matchE :: String -> String -> Bool
matchE s s' = (map toLower s') == s      ||
              (map toLower s') == "it"   ||
              (map toLower s') == "they" ||
              (map toLower s') == "them"

extract :: String -> SExpr -> [Float]
extract s (Fun s' j _ es) | matchE s s' = j:(concat $ map (extract s) es)
                          | otherwise   = (concat $ map (extract s) es)
extract s (Seq es)                      = (concat $ map (extract s) es)
extract _ _                             = []

analyse :: (Word -> Word) -> String -> (String, Int) -> IO ((Int, Maybe Float))
analyse annotationAlgorithm subject (sentence, index) = do
  tree <- runCc annotationAlgorithm sentence
  if (isJust tree) then do
    let sexpr = nodeExpr $ fromJust tree
    let r = extract subject sexpr
    let m = (maximum r) + (minimum r)
    if (null r) then return (index, Nothing)
    else if (m > 0) then
      return $ (index, Just $ maximum r)
    else if (m < 0) then
      return $ (index, Just $ minimum r)
    else
      return (index, Nothing)
  else
    return (index, Nothing)



main :: IO ()
main = do wne <- initializeWordNetWithOptions Nothing Nothing

          -- Define positive concepts
          let adj_pos_list = [("good", 1), ("beautiful", 1), ("pleasant", 1),   ("clean",
               1), ("quiet", 1), ("friendly", 1),   ("cheap", 1), ("fast", 1), ("large",
               1),("nice",1)]
          let adj_neg_list = [("bad", 1),  ("hideous", 1),   ("unpleasant", 1), ("dirty",
               1), ("noisy", 1), ("unfriendly", 1), ("expensive", 1), ("slow", 1), ("small
               ", 1),("nasty",1)]

          -- Define intensifiers concepts
          let intensifiers = [("extreme", 1), ("much", 1), ("more", 1)]
          let qualifier = [("moderate", 1), ("little", 1), ("less", 1)]

          -- Load the synsets that corresponds to the words in the above lists.
          let s_pos = (let ?wne = wne in map (\(w, i) -> head $ search w Adj i)
               adj_pos_list)
          let s_neg = (let ?wne = wne in map (\(w, i) -> head $ search w Adj i)
               adj_neg_list)

          -- Load the synsets that corresponds to the words in the above lists.
          let intensifiers_ss = (let ?wne = wne in map (\(w, i) -> head $ search w Adj i)
               intensifiers)
          let qualifier_ss = (let ?wne = wne in map (\(w, i) -> head $ search w Adj i)
               qualifier)

          -- Print info about the seed concepts
          putStr "\n\n"

          putStr $ "Positive concepts:\n"
          putStr $ unlines $ map show s_pos
          putStr "\n"
          putStr $ "Negative concepts:\n"
          putStr $ unlines $ map show s_neg
          putStr "\n"
          putStr $ "Intensifying concepts:\n"
          putStr $ unlines $ map show intensifiers_ss
          putStr "\n"
          putStr $ "Qualifying concepts:\n"
          putStr $ unlines $ map show qualifier_ss

          putStr "\n\n"

          -- Build semantic networks and annotation environment
          putStr "Unfolding semantic networks...\n"
          let (adjMap, adjGraph) = (let ?wne = wne in unfoldG (\x -> (relatedBy Similar x
               ++ relatedBy SeeAlso x)) (s_pos ++ s_neg))
          let (scaleMap, scaleGraph) = (let ?wne = wne in unfoldG (relatedBy Similar) (
               intensifiers_ss ++ qualifier_ss))
          putStr $ "- Change network : " ++ (show $ Map.size adjMap) ++ " concepts.\n"
          putStr $ "- Scale network  : " ++ (show $ Map.size scaleMap) ++ " concepts.\n"
```

```
let adjPosRoots = map (fromJust . flip Map.lookup adjMap) s_pos
let adjNegRoots = map (fromJust . flip Map.lookup adjMap) s_neg
let adj_fun = pAdj adjGraph adjPosRoots adjNegRoots . catMaybes . map (flip Map.
     lookup adjMap)

let scaleIntensifiersRoots = map (fromJust . flip Map.lookup scaleMap)
     intensifiers_ss
let scaleQualifierRoots = map (fromJust . flip Map.lookup scaleMap) qualifier_ss
let scaleFun = pScale scaleGraph scaleIntensifiersRoots scaleQualifierRoots .
     catMaybes . map (flip Map.lookup scaleMap)

let env = AnnotationEnv {
  wnEnv = wne,
  adjFun = adj_fun,
  scaleFun = scaleFun
}

-- Load review data
reviewData <- liftM lines $ readFile "../Data/rooms_swissotel_chicago_a.txt"

-- Analyse
result <- mapM (analyse (annotateWord env) "room") $ zip reviewData [1..]

-- Print results
putStr "\n\n"
putStr "Results:\n"
putStr $ unlines (map (\(i, r) -> (show i) ++ ": " ++ (show r)) result)
putStr "\n\n"
putStr "Finshed.\n"
return ()
```

Lambda.hs

```
module Lambda where
import Data.List (nub,union,(\\));


-- | Data structure for semantic expressions
data SExpr = Var String                      -- Variable
           | Abs String SExpr                -- Lambda abstraction
           | App SExpr SExpr                 -- Lambda application
           | Fun String Float Int [SExpr]    -- Functor
           | Seq [SExpr]                     -- Sequence
           | ImpactChange SExpr Int          -- Impact change
           | Change SExpr Float              -- Change
           | Scale SExpr Float               -- Scale
           deriving (Eq)


-- | Returns the set of free variables in the given expression
free :: SExpr -> [String]
free (Var x)              = [x]
free (App e1 e2)          = (free e1) `union` (free e2)
free (Abs x e)            = (free e) \\ [x]
free (Fun _ _ _ es)       = nub $ concat $ map free es
free (ImpactChange e _) = (free e)
free (Seq es)             = nub $ concat $ map free es
free (Change e _)         = (free e)
free (Scale e _)          = (free e)


-- | Safe substitution of variables x' with e' in e
subst :: SExpr -> String -> SExpr -> SExpr
subst e@(Var x) x' e'      | x == x'    = e'
                           | otherwise = e
subst e@(App e1 e2) x' e'               = App (subst e1 x' e') (subst e2 x' e')
subst e@(Abs x e1) x' e'   | x == x'    =
                             -- x is bound in e, so do not continue
                             e
                           | x `elem` free e' =
                             -- x is in FV(e'), need alpha-conversion of x:
                             let x'' = head $ xVars \\ (free e1 `union` free e')
                             in  subst (Abs x'' $ subst e1 x (Var x'')) x' e'
                           | otherwise =
                             -- otherwise just continue
                             Abs x (subst e1 x' e')
subst e@(Fun f j k es) x' e'            = Fun f j k $ (map (\e1 -> subst e1 x' e' )) es
subst e@(Seq es) x' e'                  = Seq $ (map (\e1 -> subst e1 x' e' )) es
subst e@(ImpactChange e1 k') x' e'      = ImpactChange (subst e1 x' e') k'
subst e@(Change e1 j) x' e'             = Change (subst e1 x' e') j
subst e@(Scale e1 j) x' e'              = Scale (subst e1 x' e') j
```

```
-- | Reduces a semantic expression
reduce :: SExpr -> SExpr
-- -reduction
reduce (App (Abs x t) t')            = reduce $ subst (reduce t) x (reduce t')
reduce (App t1 t2)                   = if (t1 /= t1') then (reduce $ App t1' t2) else (App t1
      ' t2)
                                       where t1' = reduce t1
reduce (Abs x t)                     = Abs x $ reduce t
reduce (Fun f j k ts)                = Fun f j k $ map reduce ts
reduce (Seq ts)                      = Seq $ map reduce ts
-- FC1, FC2, SC and PC rules:
reduce (Change (Fun f j 0 ts) j')  = Fun f (j + j') 0 $ map reduce ts
reduce (Change (Fun f j k ts) j')  = Fun f j k $ map reduce $ (take (k - 1) ts) ++
                                         [Change (ts !! (k - 1)) j'] ++ (drop k ts)
reduce (Change (Seq ts) j')          = Seq $ map (reduce . flip Change j') ts
reduce (Change (Abs x t) j')         = Abs x $ reduce $ Change t j'
reduce (Change t j)                  = if (t /= t') then (reduce $ Change t' j) else (Change
      t' j)
                                       where t' = reduce t
-- FS1, FC2, SS and PS rules:
reduce (Scale (Fun f j 0 ts) j')    = Fun f (if j == 0 then j' else j * j') 0 $ map reduce
      ts
reduce (Scale (Fun f j k ts) j')    = Fun f j k $ map reduce $ (take (k - 1) ts) ++
                                         [Scale (ts !! (k - 1)) j'] ++ (drop k ts)
reduce (Scale (Seq ts) v)            = Seq $ map (reduce . flip Scale v) ts
reduce (Scale (Abs x t) v)           = Abs x $ reduce $ Scale t v
reduce (Scale t j)                   = if (t /= t') then (reduce $ Scale t' j) else (Scale t'
      j)
                                       where t' = reduce t
-- IC rule:
reduce (ImpactChange (Fun f j k ts) k') = Fun f j k' ts
reduce (ImpactChange t k')              = if (t /= t') then
                                            (reduce $ ImpactChange t' k')
                                          else
                                            (ImpactChange t k')
                                          where t' = reduce t
-- Otherwise
reduce x                                 = x

-- | Creates an infinite list of variables [x, x', x'', ...]
xVars :: [String]
xVars = iterate (++ "'") "x"

zVars :: [String]
zVars = iterate (++ "'") "z"

fVars :: [String]
fVars = iterate (++ "'") "f"

-- | Creates an infinite list of variables [x, y, z, x', y' z', x'', y'', z'', ...]
xyzVars :: [String]
xyzVars = [v ++ v' | v' <- (iterate (++ "'") ""), v <- ["x","y","z"]]

-- | Identity semantic expression
lid = Abs "x" $ Var "x"

-- | Determines if the expression complex, i.e. needs parenthesis
isComplexExpr :: SExpr -> Bool
isComplexExpr (App _ _)           = True
isComplexExpr (Seq _)             = True
isComplexExpr (ImpactChange _ _)  = True
isComplexExpr (Change _ _)        = True
isComplexExpr (Scale _ _)         = True
isComplexExpr _                   = False

-- | Pretty printing of data structures
instance Show SExpr where
  showsPrec d (Var x)          = (showString x)
  showsPrec d (Abs x t)        = (showString $ "\\" ++ x ++ ".") . (shows t)
  showsPrec d (App t1 t2)      = (showParen (isComplexExpr t1) (shows t1)) .
                                 (showString " ") .
                                 (showParen (isComplexExpr t2) (shows t2))
  showsPrec d (Fun f j _ [])   = (showString $ f ++ "'" ++ (show j))
  showsPrec d (Fun f j k ts)   = (showString $ f ++ "'" ++ (show j) ++ "(") .
                                 (showList' ts) . (showString ")")
                                 where showList' :: Show a => [a] -> ShowS
                                       showList' [] = showString ""
                                       showList' [a] = shows a
                                       showList' (a1:a2:as) = (shows a1) .
                                                              (showString ", ") .
                                                              (showList' (a2:as))
  showsPrec d (ImpactChange t k') = (shows t) . (showString "->") . (shows k')
  showsPrec d (Seq ts)         = (shows ts)
  showsPrec d (Change t1 v)    = (shows t1) . (showString "") . (shows v)
  showsPrec d (Scale t1 v)     = (shows t1) . (showString "") . (shows v)
```

CCG.hs

```haskell
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}
module CCG (
  module Unification,
  module Lambda,
  module CCG
) where
import Unification
import Lambda

type Token = String   -- Lexical entry
type Lemma = String   -- Lemma of lexical entry
type Pos   = String   -- Part of spearch

-- | Data structure for lexical units.
data Word = Word {
              token :: Token,
              lemma :: Lemma,
              pos :: Pos,
              category :: Category,
              expr :: SExpr
            }
            deriving (Eq)

type Lexicon = [Word]

infix 9 :/  -- Forward slash operator
infix 9 :\  -- Backward slash operator

type Agreement = [Feature]

data Category = S              { agreement :: Agreement } -- Sentence
              | N              { agreement :: Agreement } -- Noun
              | NP             { agreement :: Agreement } -- Noun Phrase
              | PP             { agreement :: Agreement } -- Preposision Phrase
              | CONJ           { agreement :: Agreement } -- Conjugation (temperary category)
              | Punctuation    { agreement :: Agreement } -- Punctation  (temperary category)
              | Comma          { agreement :: Agreement } -- Comma  (temperary category)
              | Category :/ Category                      -- Forward slash
              | Category :\ Category                      -- Backward slash
              deriving (Eq)

data Feature = FDcl   | FAdj | FEm  | FInv  -- Sentence
             | FTo    | FB   | FPt  | FPss  -- Verbs
             | FNg    | FNb  | FFor
             | FThr   | FFrg                -- Fragments
             | FQ     | FWq  | FQem         -- Questions
             | FVar String                  -- Variables
             | FUnknown String              -- Unknowns
             deriving (Eq,Show)

data PTree = PWord Word
           | PFwdApp    { nCategory :: Category, nExpr :: SExpr, n1 :: PTree, n2 :: PTree }
           | PBwdApp    { nCategory :: Category, nExpr :: SExpr, n1 :: PTree, n2 :: PTree }
           | PFwdComp   { nCategory :: Category, nExpr :: SExpr, n1 :: PTree, n2 :: PTree }
           | PBwdComp   { nCategory :: Category, nExpr :: SExpr, n1 :: PTree, n2 :: PTree }
           | PBwdXComp  { nCategory :: Category, nExpr :: SExpr, n1 :: PTree, n2 :: PTree }
           | PFwdTR     { nCategory :: Category, nExpr :: SExpr, n1 :: PTree }
           | PLexRaise  { nCategory :: Category, nExpr :: SExpr, n1 :: PTree }
           deriving (Eq,Show)

-- | Gets the category of a node in the deduction tree.
nodeCategory :: PTree -> Category
nodeCategory (PWord w) = category w
nodeCategory x         = nCategory x

-- | Gets the semantic expression of a node in the deduction tree.
nodeExpr :: PTree -> SExpr
nodeExpr (PWord w) = expr w
nodeExpr x         = nExpr x

instance Unifiable Category where
  S _             =? S _             = True
  N _             =? N _             = True
  NP _            =? NP _            = True
  PP _            =? PP _            = True
  CONJ _          =? CONJ _          = True
  Punctuation _   =? Punctuation _   = True
  Comma _         =? Comma _         = True
  (a :/ b)        =? (a' :/ b')      = a =? a' && b =? b'
  (a :\ b)        =? (a' :\ b')      = a =? a' && b =? b'
  _               =? _               = False

-- | Return if a category is compound.
```

```haskell
isComplex :: Category -> Bool
isComplex (_ :/ _) = True
isComplex (_ :\ _) = True
isComplex _        = False

-- | Return the argument of the type inflicted by a compound category.
arg :: Category -> Maybe Category
arg (_:\x) = Just x
arg (_:/x) = Just x
arg x      = Nothing

-- | Return the result of the type inflicted by a compound category.
res :: Category -> Maybe Category
res (x:\_) = Just x
res (x:/_) = Just x
res _      = Nothing

-- | Pretty printing of Word
instance Show Word where
  showsPrec d (Word { token = t, lemma = lemma, pos = p, category = c, expr = e }) =
    (showString t) . (showString "~") . (showString lemma) . (showString "/") .
    (showString p) . (showString "  ") . (shows c) . (showString " : ") .
    (shows e)

-- | Pretty printing of Category
instance Show Category where
  showsPrec d (S a)             = showString "S"    . (showString " ") . (shows a)
  showsPrec d (N a)             = showString "N"    . (showString " ") . (shows a)
  showsPrec d (NP a)            = showString "NP"   . (showString " ") . (shows a)
  showsPrec d (PP a)            = showString "PP"   . (showString " ") . (shows a)
  showsPrec d (CONJ a)          = showString "CONJ" . (showString " ") . (shows a)
  showsPrec d (Punctuation a)   = showString "."    . (showString " ") . (shows a)
  showsPrec d (Comma a)         = showString ","    . (showString " ") . (shows a)
  showsPrec d (a :/ b)  =
    ((showParen (isComplex a) (shows a)) .
    (showString "/") .
    (showParen (isComplex b) (shows b)))
  showsPrec d (a :\ b)  =
    ((showParen (isComplex a) (shows a)) .
    (showString "\\") .
    (showParen (isComplex b) (shows b)))
```

Annotate.hs

```haskell
{-# LANGUAGE ImplicitParams #-}

module Annotate where

import Control.Monad.ST
import Data.STRef
import Data.List (sort)
import Data.Map (Map)
import qualified Data.Map as Map
import Data.Maybe
import Data.Graph.Inductive

import CCG
import WordNet hiding (Word)

data AnnotationEnv = AnnotationEnv {
     wnEnv    :: WordNetEnv,
     adjFun   :: [SearchResult] -> Float,  -- rename to adjChange?
     scaleFun :: [SearchResult] -> Float -- rename to adjScale?
}

-- Parameters for the annotation
omega = 100
n = 5

-- | Unfold the graph using the given relation and seeds.
unfoldG :: (Ord a) => (a -> [a]) -> [a] -> (Map a Node, Gr a Int)
unfoldG r seeds = runST $ unfoldST r seeds

-- | State trasformer for unfolding graphs.
unfoldST :: (Ord a) => (a -> [a]) -> [a] -> ST s (Map a Node, Gr a Int)
unfoldST r seeds =
  do mapRef    <- newSTRef Map.empty    -- Map from Item to Node
     nodesRef  <- newSTRef []           -- List of Node/[Edge] pairs
     idRef     <- newSTRef 0            -- Counter for indexing nodes
     -- Recursively visits n
     let visit n =
          do -- Test if n has already been visited
             test <- (return . Map.lookup n =<< readSTRef mapRef)
             case test of
               Just v  -> return v
               Nothing ->
                 do -- Get next id for this item
                    i <- readSTRef idRef
                    modifySTRef idRef (+1)
                    -- Update item/node map
                    modifySTRef mapRef (Map.insert n i)
                    -- Recursively visit related items
                    ks <- mapM visit $ r n
                    let ns = ((i,n), [(i,k,1) | k <- ks])
                    modifySTRef nodesRef (ns:)
                    return i
      -- Visit seeds
      mapM visit seeds
      -- Read results and return map/graph-pair
      list <- readSTRef nodesRef
      nodeMap <- readSTRef mapRef
      let nodes = [n | (n, _) <- list]
      let edges = concat [es | (_, es) <- list]
      return (nodeMap, mkGraph nodes edges)

-- | Polarity value for adjective graphs
pAdj :: Real b => Gr a b -> [Node] -> [Node] -> [Node] -> Float
pAdj gr pns nns qns = (sum $ distAdj nns qns) - (sum $ distAdj pns qns)
                      where
                        distAdj :: [Node] -> [Node] -> [Float]
                        distAdj sns qns = take n $ sort [normAdj (length (sp sn qn gr) -
                             1) | sn <- sns, qn <- qns]

                        normAdj :: Int -> Float
                        normAdj x | x < 0 || x > 10 = omega / (fromIntegral n)
                                  | otherwise       = (fromIntegral x / 10) * (omega / (
                                      fromIntegral n))

-- | Polarity value for scale graphs
pScale :: Real b => Gr a b -> [Node] -> [Node] -> [Node] -> Float
pScale gr pns nns qns = 2**((sum $ distScale nns qns) - (sum $ distScale pns qns))
                        where
                          distScale :: [Node] -> [Node] -> [Float]
                          distScale sns qns = take n $ sort [normScale (length (sp sn qn
                               gr) - 1) | sn <- sns, qn <- qns]

                          normScale :: Int -> Float
                          normScale x | x < 0 || x > 10 = 1 / (fromIntegral n)
                                      | otherwise       = (fromIntegral x / 10) * (1 / (
                                          fromIntegral n))


concat2 :: [[a]] -> [a]
concat2 [] = []
concat2 x = let nonEmpty = filter (not . null) x
            in (map head nonEmpty) ++ concat2 (map tail nonEmpty)

shiftTerm :: SExpr -> Int -> SExpr
shiftTerm (Abs x t) k'       = Abs x (shiftTerm t k')
shiftTerm (Fun f j k ts) k' = Fun f j k' ts

isDet pos    = pos == "DT"
isAdj pos    = (take 2 pos) == "JJ"
isVerb pos   = (take 2 pos) == "VB"
isAdverb pos = (take 2 pos) == "RB"
isNoun pos   = (take 2 pos) == "NN" || pos == "PRP"
isP pos      = pos == "IN" || pos == "TO" || pos == "WDT"

annotateDet env w@(Word { category = c, token = t, lemma = l })
  | c =? NP [] :/ N [] =
    w { expr = lid }
  | otherwise =
    annotateAny env w

annotateNoun env w@(Word { category = c, token = t, lemma = l })
  | c =? N [] :/ N [] =
    w { expr = (Abs "x" $ Seq [Fun l 0 0 [], Var "x"]) } -- Part of multi lexical noun
  | otherwise         =
    annotateAny env w

annotateVerb env w@(Word { category = c, token = t, lemma = l })
  | c =? (S [FDcl] :\ NP []):/(S [FAdj] :\ NP []) =
    w { expr = lid } -- Linking verb
  | otherwise = annotateAny env w

annotateAdj env w@(Word { category = c, lemma = l })
  | (S [FAdj] :\ NP []) =? c || (NP [] :/ NP []) =? c || (N [] :/ N []) =? c =
    let query = (let ?wne = (wnEnv env) in search l Adj AllSenses)
```

```
        value = (adjFun env) query
    in  w { expr = (Abs "x" $ Change (Var "x") value) }
  | otherwise =
    annotateAny env w

annotateAdverb env w@(Word { category = c, lemma = l })
  | arg c =? res c =
    let -- Try to lookup adjective pertainyms
        query = (let ?wne = (wnEnv env) in (concat2 $ map (relatedBy Pertainym) (search l
             Adv AllSenses)) ++ (search l Adj AllSenses))
        scaleValue  = (scaleFun env) $ filter ((==) Adj . srPOS) $ query
        changeValue = (adjFun env) $ filter ((==) Adj . srPOS) $ query
    in  if scaleValue /= 1 then
          w { expr = (Abs "x" $ Scale (Var "x") $ scaleValue) }
        else
          w { expr = (Abs "x" $ Change (Var "x") $ changeValue) }
  | otherwise =
    annotateAny env w

annotateP env w@(Word { category = c })
  | (NP [] :\ NP []) :/ (S [] :/ NP []) =? c =
    w { expr = Abs "x" $ Abs "y" $ ImpactChange (App (Var "x") (Var "y")) 1 }
  | (NP [] :\ NP []) :/ (S [] :\ NP []) =? c =
    w { expr = Abs "x" $ Abs "y" $ ImpactChange (App (Var "x") (Var "y")) 2 }
  | (NP [] :\ NP []) :/ NP [] =? c =
    w { expr = shiftTerm (expr w') 2 }
  | otherwise =
    w'
  where w' = annotateAny env w

annotateAny _ w@(Word { token = t, category = c, lemma = l }) =
  w { expr = constructTerm xyzVars [] c }
  where
    constructTerm :: [String] -> [(SExpr, Category)] -> Category -> SExpr
    constructTerm vs ts c = case c of
      r :\ a -> constructAbstraction vs ts a r
      r :/ a -> constructAbstraction vs ts a r
      _      -> Fun l 0 0 $ reverse $ [v | (v, _) <- ts]

    constructAbstraction :: [String] -> [(SExpr, Category)] -> Category -> Category ->
         SExpr
    constructAbstraction (v:vs) ts a r = -- a = tau_alpha, r = tau_beta
      let t = Var v -- NP
          cond1 = any (\(_, t) -> (Just a) =? arg t) ts -- types where a migth be used as
               argument
          cond2 = any (\(_, t) -> arg a =? (Just t)) ts -- types where a migth be used as
               function
          term = if cond1 then map (\(t', c') -> if Just a =? arg c' then (App t' t,
               fromJust $ res c') else (t', c')) ts else
                 if cond2 then map (\(t', c') -> if arg a =? Just c' then (App t t',
                      fromJust $ res a) else (t', c')) ts else
                 (t,a):ts

      in
        Abs v (constructTerm vs (term) r)

-- | Special annotation for C&C conj-rule
annotateConj :: Category -> Word -> Word
annotateConj cat w@(Word { lemma = l }) =
  w { expr = Abs "x" $ Abs "y" $ constructTerm [] cat }
  where
    newVar used = head $ dropWhile (`elem` used) $ (iterate (++ "'") "z")
    constructTerm :: [String] -> Category -> SExpr
    constructTerm used c = case c of
      a :\ _ -> let v = newVar used in Abs v (constructTerm (v:used) a)
      a :/ _ -> let v = newVar used in Abs v (constructTerm (v:used) a)
      _      -> Seq $ map (\term -> foldr (flip App) term $ map Var used) [Var "x", Var "y
          "]

-- | Auxiliary function for annotation for C&C lp/rp-rule
flatten :: Category -> [Category]
flatten (a :\ b) = b:flatten a
flatten (a :/ b) = b:flatten a
flatten a        = [a]

-- | Special annotation for C&C lp/rp-rule
annotateConj' :: Word -> Word
annotateConj' w@(Word { lemma = l, category = c }) =
  let f = flatten c
      c1 = f !! 0
      c2 = f !! 1
      cr = f !! 2
  in
    case length f of
      1 -> error "Annotate Conj: We expect least t -> t."
      2 -> w { expr = lid }  -- Dummy conjection, just return identity, for instance , and
           . in "funny , and happy ."
        _ -> w { expr = Abs "x" $ Abs "y" $ constructTerm [] c2 }
  where
    newVar used = head $ dropWhile (`elem` used) $ zVars
    constructTerm :: [String] -> Category -> SExpr
    constructTerm used c = case c of
      a :\ _ -> let v = newVar used in Abs v (constructTerm (v:used) a)
      a :/ _ -> let v = newVar used in Abs v (constructTerm (v:used) a)
      _      -> Seq $ map (\term -> foldr (flip App) term $ reverse $ map Var used) [Var "
           x", Var "y"]

-- | Special annotation for C&C ltc-rule
annotateLtc :: Lexicon -> Word -> Word
annotateLtc _ w@(Word { token = t, category = c, lemma = l }) =
  w { expr = constructTerm xyzVars [] c }
  where
    constructTerm :: [String] -> [(SExpr, Category)] -> Category -> SExpr
    constructTerm vs ts c = case c of
      r :\ a -> constructAbstraction vs ts a r
      r :/ a -> constructAbstraction vs ts a r
      _      -> Seq $ reverse $ [v | (v, _) <- ts]

    constructAbstraction :: [String] -> [(SExpr, Category)] -> Category -> Category ->
         SExpr
    constructAbstraction (v:vs) ts a r = -- a = tau_alpha, r = tau_beta
      let t = Var v -- NP
          cond = any (\(_, t) -> (Just a) =? arg t) ts -- types where a migth be used as
               argument
          term = if cond then map (\(t', c') -> if Just a =? arg c' then (App t' t,
               fromJust $ res c') else (t', c')) ts else
                 (t,a):ts
      in
        Abs v (constructTerm vs (term) r)


annotateWord :: AnnotationEnv -> Word -> Word
annotateWord env w@(Word { pos = pos })
  | isDet pos     = annotateDet env w
  | isAdj pos     = annotateAdj env w
  | isVerb pos    = annotateVerb env w
  | isAdverb pos  = annotateAdverb env w
  | isNoun pos    = annotateNoun env w
  | isP pos       = annotateP env w
  | otherwise     = annotateAny env w
```

Parser.hs

```
{-# LANGUAGE ImplicitParams #-}
module Parser (
    parseLexicon,
    parseTree,
    runCc
  ) where

-- Misc.
import Control.Monad
import Data.Char
import Data.Maybe

-- Parsec
import Text.ParserCombinators.Parsec hiding (token)
import Text.ParserCombinators.Parsec.Expr

-- Proccess handling
import System.Process
import GHC.IO.Handle

-- Data Structures
import CCG
import Annotate

pLexicon :: Parser [Word]
pLexicon = pLexiconEntry `endBy` newline

pLexiconEntry :: Parser Word
pLexiconEntry = do (string "w(")
                   many1 digit
                   (string ", ")
                   many1 digit
                   (string ", '")
                   t <- pToken
                   (string "', '")
                   lemma <- pToken
                   (string "', '")
                   pos <- pToken
                   (string "', '")
                   pToken
                   (string "', '")
                   pToken
                   (string "', '")
                   c <- pCategoryExpr
                   (string "').")
                   return $ Word t lemma pos c (Var "?")

pTree :: Lexicon -> Parser PTree
pTree l = do (string "ccg(")
             many1 digit
             (string ",")
             t <- pSubtree l
             (string ").")
             return t

pSubtree :: Lexicon -> Parser PTree
pSubtree l = do     try (pBRule l "fa"  $ \c t1 t2 -> PFwdApp   c (reduce $ App (nodeExpr
     t1) (nodeExpr t2)) t1 t2 )
                <|> try (pBRule l "ba"  $ \c t1 t2 -> PBwdApp   c (reduce $ App (nodeExpr
                     t2) (nodeExpr t1)) t1 t2 )
                <|> try (pBRule l "fc"  $ \c t1 t2 -> PFwdComp  c (reduce $ Abs "x" $ App
                     (nodeExpr t1) (App (nodeExpr t2) (Var "x"))) t1 t2 )
                <|> try (pBRule l "bc"  $ \c t1 t2 -> PBwdComp  c (reduce $ Abs "x" $ App
                     (nodeExpr t2) (App (nodeExpr t1) (Var "x"))) t1 t2 )
                <|> try (pBRule l "bx"  $ \c t1 t2 -> PBwdXComp c (reduce $ Abs "x" $ App
                     (nodeExpr t2) (App (nodeExpr t1) (Var "x"))) t1 t2 )
                <|> try (pURule l "tr"  $ \c t     -> PFwdTR    c (reduce $ Abs "f" (App (
                     Var "f") $ nodeExpr t)) t )
                <|> try (pConj l)
                <|> try (pConj'' l)
                <|> try (pConj''' l)
                <|> try (pLex l)
                <|> try (pWord l)
                <?> "subtree"

pURule :: Lexicon -> String -> (Category -> PTree -> PTree) -> Parser PTree
pURule l f r = do (string f)
                  (string "('")
                  c <- pCategoryExpr
                  (string "',")
                  t <- pSubtree l
                  (string ")")
                  return $ r c t

pBRule :: Lexicon -> String -> (Category -> PTree -> PTree -> PTree) -> Parser PTree
pBRule l f r = do (string f)
                  (string "('")
                  c <- pCategoryExpr
                  (string "',")
                  t1 <- pSubtree l
                  (string ",")
                  t2 <- pSubtree l
                  (string ")")
                  return $ r c t1 t2

pConj :: Lexicon -> Parser PTree
pConj l = do r <- (try (string "conj") <|> (string "lp"))
             (string "('")
             t <- pToken
             (string "','")
             c1 <- pCategoryExpr
             (string "','")
             c2 <- pCategoryExpr
             (string "',")
             t1 <- pSubtree l
             (string ",")
             t2 <- pSubtree l
             (string ")")
             let t1' = case t1 of
                         PWord w   -> Just $ PWord $ annotateConj c1 $ w { category = (c1
                              :\ c1) :/ c1 }
                         otherwise -> Nothing
             if (isNothing t1') then
               unexpected ("Left child of a conjunction rule '" ++ r ++ "' should be a
                    word.")
             else
               return $ PFwdApp c2 (reduce $ App (nodeExpr (fromJust t1')) (nodeExpr t2))
                    (fromJust t1') t2

pConj''' :: Lexicon -> Parser PTree
pConj''' l = do r <- (try (string "lp") <|> (string "ltc"))
                (string "('")
                c <- pCategoryExpr
                (string "',")
                t1 <- pSubtree l
                (string ",")
                t2 <- pSubtree l
                (string ")")
                let t1' = case t1 of
                            PWord w   -> Just $ PWord $
                              case r of
                                "lp"  -> annotateConj' $ w { category = c :/ c }
                                "ltc" -> annotateLtc l (w { category = c :/ (nodeCategory
                                     t2) })
                            otherwise -> Nothing
                if (isNothing t1') then
                  unexpected $ "Left child of a conjunction rule '" ++ r ++ "' should be a
                        word."
                else
                  return $ PFwdApp c (reduce $ App (nodeExpr (fromJust t1')) (nodeExpr t2)
                       ) (fromJust t1') t2

pConj'' :: Lexicon -> Parser PTree
pConj'' l = do r <- (string "rp")
               (string "('")
               c1 <- pCategoryExpr
               (string "',")
               token <- optionMaybe (do { char '\''; t <- pToken; char '\''; char ',';
                    return t })
               c2 <- optionMaybe (do { char '\''; c <- pCategoryExpr; char '\''; char ',';
                     return c })
               t1 <- pSubtree l
               (string ",")
               t2 <- pSubtree l
               (string ")")
               let t2' = case t2 of
                           PWord w   -> Just $ PWord $ annotateConj' $ w { category = c1
                                :\ c1 }
                           otherwise -> Nothing
               if (isNothing t2') then
                 unexpected $ "Right child of a conjunction rule '" ++ r ++ "' should be a
                       word."
               else
                 return $ PBwdApp c1 (reduce $ App (nodeExpr (fromJust t2')) (nodeExpr t1)
                      ) t1 (fromJust t2')

pLex :: Lexicon -> Parser PTree
pLex l = do (string "lex('")
            c1 <- pCategoryExpr
            (string "','")
            c2 <- pCategoryExpr
            (string "',")
            t <- pSubtree l
            (string ")")
            return $ PLexRaise c2 (nodeExpr t) t

pWord :: Lexicon -> Parser PTree
pWord l = do (string "lf(")
             (many1 digit)
             (string ",")
             wordIndex <- (many1 digit)
             (string ",'")
             c <- pCategoryExpr
             (string "')")
             return $ PWord (l !! ((read wordIndex :: Int) - 1))

pToken :: Parser String
pToken = many1 $ upper <|> lower <|> digit <|> oneOf "_-$,.!?" <|> escaped <|> (char '' >>
      return '\'')

escaped = char '\\' >> choice (zipWith escapedChar codes replacements)
escapedChar code replacement = char code >> return replacement
codes        = ['\'', '"']
replacements = ['\'', '"']

pParens :: Parser a -> Parser a
pParens = between (char '(') (char ')')

pBrackets :: Parser a -> Parser a
pBrackets = between (char '[') (char ']')

pCategoryExpr :: Parser Category
pCategoryExpr = buildExpressionParser pCategoryOpTable pCategory

pCategoryOpTable :: OperatorTable Char st Category
pCategoryOpTable = [ [ op "/"  (:/) AssocLeft,
                       op "\\" (:\) AssocLeft ] ]
                   where
                     op s f a = Infix ( string s >> return f ) a
```

```
pCategory :: Parser Category
pCategory =        pParens pCategoryExpr
            <|>     (pCategory' "S"     S)
            <|> try (pCategory' "NP"    NP)
            <|>     (pCategory' "N"     N)
            <|>     (pCategory' "PP"    PP)
            <|>     (pCategory' "conj"  CONJ)
            <|>     (pCategory' "."     Punctuation)
            <|>     (pCategory' ","     Comma)
            <?> "category"

pCategory' :: String -> (Agreement -> Category) -> Parser Category
pCategory' s c = do string s
                    a <- pAgreement
                    return $ c a

pAgreement :: Parser Agreement
pAgreement = option [] (pBrackets $ pFeature `sepBy` (char ','))

pFeature :: Parser Feature
pFeature =      try (string "dcl"   >> return FDcl )
           <|> try (string "adj"   >> return FAdj )
           <|> try (string "pt"    >> return FPt  )
           <|> try (string "nb"    >> return FNb  )
           <|> try (string "ng"    >> return FNg  )
           <|> try (string "em"    >> return FEm  )
           <|> try (string "inv"   >> return FInv )
           <|> try (string "pss"   >> return FPss )
           <|> try (string "b"     >> return FB   )
           <|> try (string "to"    >> return FTo  )
           <|> try (string "thr"   >> return FThr )
           <|> try (string "frg"   >> return FFrg )
           <|> try (string "wq"    >> return FWq  )
           <|> try (string "qem"   >> return FQem )
           <|> try (string "q"     >> return FQ   )
           <|> try (string "for"   >> return FFor )
           <|> ( do { v <- many1 upper; return $ FVar     v } )
           <|> ( do { v <- many1 lower; return $ FUnknown v } )
           <?> "feature"

parseLexicon :: String -> Lexicon
parseLexicon str =
  case parse pLexicon "Parse error:" str of
    Left e  -> error $ show e
    Right r -> r

parseTree :: Lexicon -> String -> (Maybe PTree)
parseTree l str =
  case parse (pTree l) "Parse error:" str of
    Left e  -> Nothing -- error $ show e
    Right r -> Just r

getSection :: Handle -> String -> IO (Maybe String)
getSection h s =
    do -- hWaitForInput h (-1)
       eof <- hIsEOF h
       if eof then
         return Nothing
       else do
         inpStr <- hGetLine h
         if inpStr == "" then
           return $ Just s
         else do
           -- putStr ("Got line: " ++ inpStr ++ "\n")
           getSection h (s ++ inpStr ++ "\n")

runCc :: (Word -> Word) -> String -> IO (Maybe PTree)
runCc a s = do
  (Just inHandle, Just outHandle, _, processHandle) <-
     createProcess (proc "bin/soap_client_fix" [
         "--url", "http://localhost:9000"]) {
         std_in = CreatePipe,
         std_out = CreatePipe
     }
  hSetBuffering inHandle LineBuffering
  hSetBuffering outHandle LineBuffering

  putStr "\n"
  putStr "Parsing:\n"
  putStr s
  putStr "\n"
  hPutStr inHandle s
  hPutStr inHandle "\n"

  getSection outHandle ""            -- Discard comments
  getSection outHandle ""            -- Discard functor declarations
```

```haskell
tree    <- getSection outHandle "" -- Tree
lexicon <- getSection outHandle "" -- Lexicon

if (isJust tree) && (isJust lexicon) then
  do let l = map a $ parseLexicon $ fromJust lexicon
     let tree' = filter (not . isSpace) $ fromJust tree
     let t = parseTree l tree'
     return t
else
  return Nothing
```

Appendix C

Labeled test data

The following table consists of a random sample chosen from the "Swissotel Hotel" topic of the *Opinosis data set* [Ganesan *et al.*, 2010] which contain any morphological form of the *subject of interest: hotel rooms*. Each sentence in the data set (which may not constitute a complete review) has been labeled independently by two human individuals *with respect to the subject of interest: hotel rooms*. Furthermore the table contains results for the baseline (sentence level polarity value), and results for the presented method (entity level polarity value of subject of interest).

#	Review text	Human A	Human B	Presented method
1	The rooms are in pretty shabby condition , but they are clean .	Negative	Negative	Unknown
2	The rooms are spacious and have nice views, I was NOT impressed with the mattress and every, little, tiny thing costs money .	Unknown	Unknown	N/A
3	The rooms look like they were just remodled and upgraded, there was an HD TV and a nice iHome docking station to put my iPod so I could set the alarm to wake up with my music instead of the radio .	Positive	Positive	Unknown
4	The rooms were cleaned spic and span every day .	Positive	Positive	Unknown
5	When I got to the room , I thought the new rooms would have a plasma since the website implies the new rooms would have them , but I guess those come later .	Negative	Negative	Unknown
6	Very impressed with rooms and view !	Positive	Positive	Unknown
7	The rooms are not all that big .	Negative	Negative	Unknown
8	Expensive Parking but great rooms .	Positive	Positive	30.0
9	Rooms were nicely furnished .	Positive	Positive	Unknown
10	The rooms are very clean , comfortable and spacious and up-to-date .	Positive	Positive	52.0
11	I've olny ever stayed in the "standard" rooms in this property , all of which are spacious and airy , and function well for both business or leisure travellers .	Positive	Positive	Unknown
12	It does suffer , however , from a trend that I have been noticing that as rooms at business class hotels are upgraded , particularly with a patch panel for the big LCD , TV , drawer space becomes less and less .	Negative	Negative	Unknown
13	We even got upgraded to one of the corner rooms which also looked west toward Michigan Ave and the Wrigley building .	Positive	Positive	Unknown
14	The rooms were very clean , the service was polite and helpful , and it's near the heart of Chicago !	Positive	Positive	52.0
15	You can see downtown and or the Navy Pier from most of the rooms .	Positive	Positive	Unknown

#	Review text	Human A	Human B	Presented method
16	no more bathrobes in corner rooms suites , coffee service in room is parred way down , the buffet offered in the cafe is not as bountiful , although the cafe staff is inpeccable and extremely gracious and will bring you what you wish , check in staff not at all eager to upgrade you, even though you may be a frequent visitor .	Negative	Negative	Unknown
17	Our rooms were nice and didn't look worn or old .	Positive	Positive	Unknown
18	Rooms at the hotel are getting somewhat tired .	Negative	Negative	0.8
19	Great Location great rooms and bed but no help from desk personnel .	Positive	Positive	38.0
20	While the rooms are quite nice , I was dismayed by the snotty service I received at the Swissotel in Chicago .	Positive	Positive	72.0
21	Rooms are dated , our corner room's bathroom was shabby .	Negative	Negative	Unknown
22	The hotel was very nice , rooms were big , the pool hot tub area was very nice , and the location was great and easy to get to .	Positive	Positive	10.0
23	Rooms are good quality and clean , what you would expect from a four star business hotel .	Positive	Positive	46.0
24	The view from the rooms was fantastic , My daughters are allergic to feathers and all trace of them were removed from the room as soon as we advised housekeeping .	Positive	Positive	Unknown
25	The Swissotel is one of our favorite hotels in Chicago and the corner rooms have the most fantastic views in the city .	Positive	Positive	Unknown
26	Then again , the rooms are much larger and the view more than makes up for it .	Positive	Positive	26.0
27	Rooms in similar hotels would usually be about $250 , 300 .	Positive	Positive	Unknown
28	The actual hotel and rooms were very nice with amazing views , the staff was extremely rude .	Positive	Positive	8.0
29	The rooms were clean , and upscale for the low price we paid .	Positive	Positive	Unknown
30	Thanks to TravelZoo I was able to find an amazing deal , lakeside rooms for $129 night as part of a spring promotion .	Positive	Positive	Unknown
31	I recieved a great deal on the rooms here and it was wonderful .	Positive	Positive	8.0

#	Review text	Human A	Human B	Presented method
32	The room was huge as hotel rooms go .	Positive	Positive	26.0
33	Hotel was very clean and the rooms were comfy .	Positive	Positive	Unknown
34	word to the wise , avoid the rooms ending with 11 .	Negative	Negative	Unknown
35	The rooms are large and well , appointed , the staff was very professional and friendly , and the view was striking !	Positive	Positive	34.0

Table C.1: Labeled test data, subject of interest: Room

Bibliography

[Baccianella *et al.*, 2010] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. SentiWordNet 3.0: An Enhanced Lexical Resource for Sentiment Analysis and Opinion Mining. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*. European Language Resources Association (ELRA), 2010.

[Baldridge and Kruijff, 2003] Jason Baldridge and Geert-Jan M. Kruijff. Multi-Modal Combinatory Categorial Grammar. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1*, EACL '03, pages 211–218, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[Baldridge, 2002] Jason Baldridge. *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. PhD thesis, University of Edinburgh, 2002.

[Barendregt *et al.*, 2012] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types (Perspectives in Logic)*. Draft for unpublished book, http://www.cs.ru.nl/~henk/book.pdf, 2012.

[Blitzer *et al.*, 2007] John Blitzer, Mark Dredze, and Fernando Pereira. Biographies, Bollywood, Boomboxes and Blenders: Domain Adaptation for Sentiment Classification. In *In ACL*, pages 187–205, 2007.

[Box *et al.*, 2000] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. *Simple Object Access Protocol (SOAP) 1.1.* Url: http://www.w3.org/TR/soap/, 2000.

[Bresnan *et al.*, 1982] J. Bresnan, R. M. Kaplan, S. Peters, and A. Zaenen. Cross-Serial Dependencies in Dutch. *Linguistic Inquiry*, 13(fall):613–635+, 1982.

[Carreras *et al.*, 2002] Xavier Carreras, Lluís Màrquez, and Lluís Padró. Named Entity Extraction using AdaBoost. In *Proceedings of the 6th Conference on Natural Language Learning - Volume 20*, COLING-2002, pages 1–4, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[Clark and Curran, 2007] Stephen Clark and James R. Curran. Wide-Coverage Efficient Statistical Parsing with CCG and Log-linear Models. *Computational Linguistics*, 33(4):493–552, 12 2007.

[Clark, 2002] Stephen Clark. A Supertagger for Combinatory Categorial Grammar. In *Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6)*, pages 19–24, Venice, Italy, 2002.

[Daumé III, 2008] Hal Daumé III. *HWordNet - A Haskell Interface to WordNet*. Url: http://www.umiacs.umd.edu/~hal/, 2008.

[Erwig, 2001] Martin Erwig. Inductive Graphs and Functional Graph Algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[Esuli and Sebastiani, 2006] Andrea Esuli and Fabrizio Sebastiani. SentiWordNet: A Publicly Available Lexical Resource for Opinion Mining. In *Proceedings of the 5th Conference on Language Resources and Evaluation (LREC'06)*, pages 417–422. European Language Resources Association (ELRA), 2006.

[Eurostat, 2010] Eurostat. *Population on 1 January by Age and Sex*. Url: http://ec.europa.eu/eurostat, 2010.

[Fellbaum, 1998] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database (Language, Speech, and Communication)*. The MIT Press, illustrated edition edition, 1998.

[Francis and Kucera, 1979] W. Nelson Francis and Henry Kucera. Brown Corpus Manual. Technical report, Department of Linguistics, Brown University, Providence, Rhode Island, US, 1979.

[Ganesan *et al.*, 2010] Kavita Ganesan, Cheng Xiang Zhai, and Jiawei Han. Opinosis: A graph based approach to abstractive summarization of highly redundant opinions. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING '10)*, 2010.

[Guttman, 1949] Louis Guttman. *The Basis for Scalogram Analysis*, 1949.

[Hockenmaier and Steedman, 2007] Julia Hockenmaier and Mark Steedman. CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396, 2007.

[Hockenmaier *et al.*, 2004] Julia Hockenmaier, Gann Bierner, and Jason Baldridge. Extending the Coverage of a CCG System. *Journal of Language and Computation*, 2:165–208, 2004.

[Hockenmaier, 2003] Julia Hockenmaier. *Data and Models for Statistical Parsing with Combinatory Categorial Grammar.* PhD thesis, University of Edinburgh, 2003.

[Joshi *et al.*, 1975] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree Adjunct Grammars. *Journal of Computer and System Sciences*, 10(1):136–163, 1975.

[Joshi *et al.*, 1990] Aravind K. Joshi, K. Vijay Shanker, and David Weir. *The Convergence Of Mildly Context-Sensitive Grammar Formalisms*, 1990.

[Launchbury and Peyton Jones, 1994] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM.

[Leijen, 2001] Daan Leijen. *Parsec, A Fast Combinator Parser.* University of Utrecht, Department of Computer Science, PO.Box 80.089, 3508 TB Utrecht, The Netherlands, 2001.

[Likert, 1932] Rensis Likert. A Technique for The Measurement of Attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[Liu, 2007] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data.* Springer, 2007.

[Marcus *et al.*, 1993] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a Large Annotated Corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[Miller, 1995] George A. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[Montague, 1974] Richard Montague. *Formal Philosophy: Selected Papers of Richard Montague.* Yale University Press, 1974.

[Padró *et al.*, 2010] Lluís Padró, Samuel Reese, Eneko Agirre, and Aitor Soroa. Semantic Services in FreeLing 2.1: WordNet and UKB. In Pushpak Bhattacharyya, Christiane Fellbaum, and Piek Vossen, editors, *Principles, Construction, and Application of Multilingual Wordnets*, pages 99–105, Mumbai, India, February 2010. Global Wordnet Conference 2010, Narosa Publishing House.

[Pang and Lee, 2008] Bo Pang and Lillian Lee. Opinion Mining and Sentiment Analysis. *Foundations and Trends in Information Retrieval*, 2(1-2):1–135, 2008.

[Paul and Baker, 1992] Douglas B. Paul and Janet M. Baker. The design for the Wall Street Journal-based CSR corpus. In *Proceedings of the workshop on Speech and Natural Language*, pages 357–362, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics.

[Pingdom, 2010] Pingdom. *Study: Ages of Social Network Users.* Url: http://pingdom.com/, 2010.

[Pollard, 1984] Carl Pollard. *Generalized Context-Free Grammars, Head Grammars and Natural Language.* PhD thesis, Stanford University, 1984.

[Russell and Norvig, 2009] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3rd edition, 2009.

[Shieber, 1985] Stuart M. Shieber. Evidence Against the Context-Freeness of Natural Language. *Linguistics and Philosophy*, 8(3):333–343, 1985.

[Simančík and Lee, 2009] František Simančík and Mark Lee. A CCG-based System for Valence Shifting for Sentiment Analysis. *Research in Computing Science*, 41:99–108, 2009.

[Steedman, 1998] Mark Steedman. *Categorial Grammar*, 1998.

[Steedman, 2000] Mark Steedman. *The Syntactic Process.* The MIT Press, 2000.

[Steedman, 2011] Mark Steedman. *Taking Scope: The Natural Semantics of Quantifiers.* The MIT Press, 2011.

[Søgaard, 2012] Anders Søgaard. *Semi-supervised Learning.* ESSLLI-2012 Lecture, 2012.

[Tan *et al.*, 2011] Luke Kien-Weng Tan, Jin-Cheon Na, Yin-Leng Theng, and Kuiyu Chang. Sentence-level Sentiment Polarity Classification using a Linguistic Approach. In *Proceedings of the 13th International Conference on Asia-pacific Digital Libraries: For Cultural Heritage, Knowledge Dissemination, and Future Creation*, ICADL'11, pages 77–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[van Eijck and Unger, 2010] Jan van Eijck and Christina Unger. *Computational Semantics with Functional Programming.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[Vijay-Shanker and Weir, 1994] K. Vijay-Shanker and David J. Weir. The Equivalence Of Four Extensions Of Context-Free Grammars. *Mathematical Systems Theory*, 27:27–511, 1994.

[Webster and Kit, 1992] Jonathan J. Webster and Chunyu Kit. Tokenization as the Initial Phase in NLP. In *Proceedings of the 14th Conference on Computational Linguistics - Volume 4*, COLING-1992, pages 1106–1110, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics.