IMPLEMENTING THE TYPE-RAISING ALGORITHM BY GRAMMAR COMPILING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

OĞUZHAN DEMİR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COGNITIVE SCIENCE

SEPTEMBER 2019

IMPLEMENTING THE TYPE-RAISING ALGORITHM BY GRAMMAR COMPILING

submitted by **OĞUZHAN DEMİR** in partial fulfillment of the requirements for the degree of **Master of Science in Cognitive Science Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin _____
Dean, **Graduate School of Informatics Institute, METU**

Prof. Dr. Cem Bozşahin _____
Head of Department, **Cognitive Science, METU**

Prof. Dr. Cem Bozşahin _____
Supervisor, **Cognitive Science, METU**

Examining Committee Members:

Assist. Prof. Dr. Umut Özge
Cognitive Science, METU _____

Prof. Dr. Cem Bozşahin
Cognitive Science, METU _____

Assist. Prof. Dr. Burcu Can
Computer Engineering, Hacettepe University _____

Date: **05 September 2019**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: OĞUZHAN DEMİR

Signature :

ABSTRACT

IMPLEMENTING THE TYPE-RAISING ALGORITHM BY GRAMMAR COMPILING

Demir, Oğuzhan

M.S., Department of Cognitive Science

Supervisor : Prof. Dr. Cem Bozşahin

September 2019, 36 pages

Type-raising is part of theory of Combinatory Categorial Grammar, by which all arguments including complements are type-raised. Generating type-raising rules in an automatic manner in the compile-time via a simple tool would make experimenting with Combinatory Categorial Grammar faster, allowing control on each run. In this study, created tool is tested with various grammars including large scale Eve database, giving results in O(N) where N is the number of verbs in the grammar.

Keywords: Combinatory Categorial Grammar, Type-Raising

ÖZ

DİLBİLGİSİ DERLEME YÖNTEMİYLE TÜR-YÜKSELTME ALGORİTMASININ GELİŞTİRİLMESİ

Demir, Oğuzhan

Yüksek Lisans, Bilişsel Bilimler Bölümü

Tez Yöneticisi : Prof. Dr. Cem Bozşahin

Eylül 2019 , 36 sayfa

Tür-yükseltme, Bileşimsel Ulamsal Dilbilgisi teorisinin bir parçasıdır. Her tümlece ve argümana bu teori dahilinde Tür-yükseltmesi uygulanır. Tür-yükseltme kurallarını kendi kendine derleme zamanında oluşturabilen basit bir araç Bileşimsel Ulamsal Dilbilgisi ile oynamada zaman kazandıracaktır ve her program çalıştırılışında işletmeye izin verecektir. Bu çalışmada yaratılan araç bunların içinde büyük ölçekli veritabanı Eve de olan çeşitli dilbilgisi ile de test edildi ve dilbilgisindeki yüklem sayısının N olduğu durumda O(N) karmaşıklığında sonuç verdiği görüldü.

Anahtar Kelimeler: Bileşimsel Ulamsal Dilbilgisi, Tür-yükseltme

This page is like a tattoo. Once it is written it is never deleted. I don't have a tattoo and may never have it but I am grateful to some people in my life and would like to express it.

To my father and mother that taught me to be honest and responsible. To my sister that taught me to believe in myself. I am happy to make them proud and all my life I will try not to change that.

ACKNOWLEDGMENTS

I would like to express my appreciation to Prof. Cem Bozşahin for taking his time to help me each and every week along my research while keeping his patience to tell the mistakes I make, explaining the parts that I had hard time understanding. I would also like to thank my company and colleagues for allowing me to do my research using the resources of the company while providing an almost home-office environment on the campus of my university. This work could not have been completed without my family's support. Thanks mom, dad and sis. My friends also supported me morally throughout the thesis period, thanks for being a part of this valuable work. Special thanks to random strangers that shared funny contents and made genious comments on social media, those kept me going as well.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CCG	Combinatory Categorial Grammar
CFG	Context Free Grammar
CNF	Chomsky Normal Form
DCCG	Discourse CCG
POS	Part of Speech

CHAPTER 1

INTRODUCTION

Type-raising is Combinatory Categorial Grammar's capture of grammatical case in all its aspects. In this chapter, we define what type-raising is and why we need it, together with its relation to Categorial Type Transparency and Surface Structure. After that, in the following chapters, Combinatory Categorial Grammar (CCG) (Steedman, 1996, 2000) and CCGlab (Bozsahin, 2015) are taken up in detail.

In this study, we type-raise all the verbs' arguments and create type-raising rules out of them. Then these rules are given to CCGlab parser to bring more control on the parsing of sentences. Unlike the already present parsers that use specifically selected type-raising rules, this tool allows us to create these rules according to the provided grammar. The algorithm to generate type-raising rules' details are discussed in chapter 3. Other tools that use CCG to parse sentences are examined in chapter 3 as well. Chapter 4 gives some statistical information by providing tables on parsing sentences using this tool.

1.1 The Need for Type-Raising

The type-raising mechanism is considered to be universal and it allows an additional non-standard "surface-structure" by making the subjects a function over predicates, facilitating their combination. We shall see that not only subjects but all arguments must be type-raised. The action referred to as type-raising will be elaborated both in this section and in the rest of the thesis.

Type-raising is necessary when an argument wants to compose with a verb that seek such argument so that they can be a part of a coordination. A motive behind using type-raising as a facilitator would be that when a category is type-raised, it creates a constituent that becomes a clue that can be used to combine the rest of the sentence with the type-raised portion. A crucial feature of type-raising is that it is intended to give the same semantic interpretation when a sentence is parsed without type-raising. This is because the functional composition rules of combinatory categorial grammars (CCG) are procedurally neutral (Pareschi and Steedman, 1987) meaning there is no more than one way to combine any given two substantive categories. Eventually the type-raising rules can be understood as a tool to enable composition

where a basic category of CCG would not allow composition.

The other combinatory rules aside, type-raising can be thought as a lexical or morphological level process and the previous CCG parsers (easyCCG, openCCG, Clark and Curran parser) included it in the derivation via a unary rule. We generate these rules automatically by looking at the grammar. This gives more power over deriving sentences than selecting specific type-raising rules and hoping them to work in all of the cases. However, rule generation is a computationally challenging process especially when identical rules are generated more than one time. As a result, the number of derivations may become too many to deal with because of using the same rules over and over. A solution to this problem is mentioned in the Results and Discussions chapter.

1.2 Transparency and Surface Structure

Transparency is a property of surface structure. The surface structure of a sentence is the final stage in the syntactic representation of a sentence, which provides the input to the phonological component of the grammar, and which thus most closely corresponds to the structure of the sentence that is articulated and heard.

Semantic transparency in a word is the degree to which the meaning of a compound word or an idiom can be inferred from its parts (or morphemes) (Nordquist, 2019). Transparency in Categorial Grammars, on the other hand, depends on the principle Categorial Type Transparency. Each syntactic category is connected to a semantic logical form. Although some predicates have the exact same category, their logical form is different. CCG has this type of transparent projection that holds the meaning of a predicate in the logical form. The Categorial Type Transparency says that the reduction application made in the semantic logical form of a category follows its syntactic type. This is exemplified in the Background chapter, under the title CCG.

People tend to compute syntax by looking at a number of cues as they try to understand the sentence and the surface structure gives clear information about the syntactic representation of a sentence. The researchers Bever (1970), Fodor and Garrett (1967) proposed a method of parsing sentences using syntactic cues. They explained that when a determiner like 'the' or 'a' is heard, it is supposable that a noun phrase has started. Another example of this dynamic would be word order. Although it is prone to change when the structure of a sentence is made passive, in most sentences, people expect the first noun to be the subject and the second to be the object. (Harley, 2014)

Having made such points about cues in language and parsing, it can now be understood where lexicalized grammars obtain their power from. Each set of structural definitions includes deep structures, surface structures, a phonetic representation and a semantic representation. If a grammar is able to provide the information that a structural definition has, then the information can be used to parse sentences or create new ones in accordance with the syntax and

semantics of the sentence. As a lexicalized grammar, CCG together with the type-raising rules, adds a surface structure by enabling a sentence reach its final form in lock-step with semantics.

CHAPTER 2

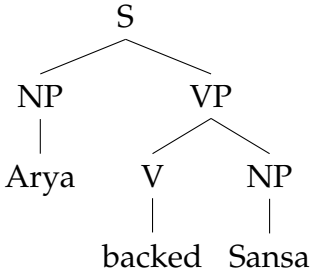
BACKGROUND

2.1 Combinatory Categorial Grammar (CCG)

Combinatory Categorial Grammar (Steedman, 1996, 2000) is a way to express a grammar by classifying the units as categories. It is an approach that expresses a sentence both syntactically and semantically. This can be a sentence in a natural or synthethic language. Units' categories are divided into two; Basic and complex. CCG is a type of lexicalized grammar. While generative grammars parse strings top-down as shown in Figure 2.1, CCG parses strings bottom-up.

Figure 2.1 Generative grammar and top-down parse example

(a)
S $\rightarrow$ NP VP
VP $\rightarrow$ V NP
NP $\rightarrow$ Sansa, Arya
V $\rightarrow$ backed
(b)



CCG has both the syntactic and semantic rules as lexical correspondence. Every CCG grammar entry has a syntactic category and a logical form representing the semantic type.

The syntactic category assignment system works as a function taking arguments and returning a result. After assigning each unit a category, all the units are parsed together to form a meaningful sentence. Basic categories would be **N, NP, S** etc.(noun, nounphrase and sentence, respectively), while complex categories are the combination of these basic categories. The notation to combine them is α / β or $\alpha \setminus \beta$.

Here the argument is β and the result is α. The slash in between indicates the side where the function needs to get its argument to give the desired output. In a category like **(S / NP)**, α is S, and β is NP. The slash is leaning forward, meaning that a word that has aforementioned category expects an NP to its right to become an S, with NP being the argument and S being the output of this function. It can also be **(S \ NP)**. This time it expects the NP to its left to become an S.

(S / NP) by itself may be an argument or a result. For example in a category like **S \ (S / NP)**, β is **(S / NP)** and if **(S / NP)** is provided to the left of the holder of this category, it will give an S. **(S / NP) / S** will give **(S / NP)** as a result if we provide S to its right. These categories are assumed to be left associative so we do not need to use parenthesis in the category **(S / NP) / S**, because it equals **S / NP / S**.

After the category assignment is settled, units can be joined by a set of combinatory rules. These include application, type raising (T), composition (B), and substitution (S) combinators. Out of these the most common are forward and backward functional application rules and they are shown in Figure 2.2.

Logical form for the semantic representation of words is defined using lambda calculus. Lambda calculus (also written as λ-calculus) is denoted as a mathematical logic to represent computation which employs function abstraction and function application methods using variable binding and substitution. It is utilized as a universal model of computation.

Lambda calculus includes constructing lambda terms and performing reduction operations on them. In the simplest form of lambda calculus, terms are built using only the following rules, shown in Table 2.1:

Table 2.1: Lambda calculus

SYNTAX	NAME	DESCRIPTION
x	Variable	A character or string representing a parameter or mathematical/logical value
(λ x.M)	Abstraction	Function definition (M is a lambda term). The variable x becomes bound in the expression.
(M N)	Application	Applying a function to an argument. M and N are lambda terms.

Figure 2.2 Functional application rules

(a) $X/Y : f \quad Y : a \quad \Rightarrow \quad X : fa$	(Forward Application : >)
(b) $Y : a \quad X\backslash Y : f \quad \Rightarrow \quad X : fa$	(Backward Application : <)

An example grammar is provided in Figure 2.3

Figure 2.3 Example grammar of CCG

Sansa := NP : *Sansa′*
Leg := N : *Leg′*
Days := NP\N : $\lambda x.\ days'x$
skips := S\NP/NP : $\lambda x \lambda y.\ skips'xy$

Figure 2.4 shows how the combinatory rules are applied to the grammar in Figure 2.3.

Figure 2.4 CCG bottom-up parse example

$$\dfrac{\dfrac{\dfrac{\text{Sansa}}{NP : Sansa'} \quad \dfrac{\text{skips}}{S\backslash NP/NP : \lambda x \lambda y.skips'\,x\,y} \quad \dfrac{\dfrac{\text{leg}}{N : leg'} \quad \dfrac{\text{days}}{NP\backslash N : \lambda x \lambda y.days'\,x\,y}}{NP : leg'\ days'}<}{S\backslash NP : \lambda y.skips'\ leg'\ days'\ y}>}{S : skips'\ leg'\ days'\ Sansa'}<$$

Another rule used in CCG is composition. In Figure 2.5 two of the composition rules are shown.

Figure 2.5 Basic composition rules

(a) $X/Y : f \quad Y/Z : g \quad \Rightarrow_{\mathbf{B}} \quad X/Z : \lambda x.f(gx)$ (Forward Composition : >**B**)
(b) $Y\backslash Z : f \quad X\backslash Y : g \quad \Rightarrow_{\mathbf{B}} \quad X\backslash Z : \lambda x.f(gx)$ (Backward Composition : <**B**)

2.2 Type-Raising (TR)

In order for a sentence like "Arya does but Sansa skips leg days" to compose, arguments must seek a functor to make themselves attach to the rest of the sentence. Type-raising allows this. When an argument gets type-raised, it becomes a function by itself so that any other category near it can combine with it. Figure 2.6 is an example to both forward type-raising and subject type raising of the sentence mentioned in the beginning. NP's *Arya* and *Sansa* need to use type-raising rules to combine with their respective verbs *does* and *skips*. Objects and complements also need type-raising when a sentence is composed. In Figure 2.7, it is shown that objects *dagger* and *sword* need TR to combine.

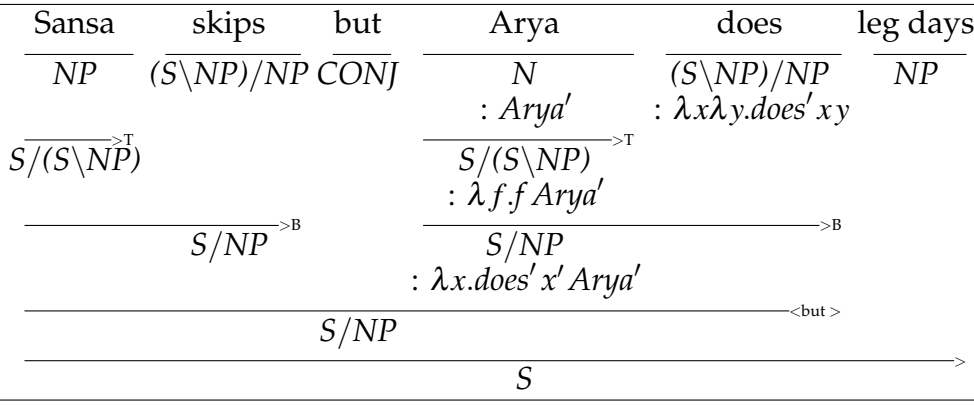
Type raising is universal. Every language needs TR in their derivations. An example TR derivation in Turkish is shown in Figure 2.10. Below are the type-raising rules:

Forward Type-raising: $\mathrm{NP} : \mathrm{a} \rightarrow \mathrm{T} \ / \ (\mathrm{T}\backslash\mathrm{NP}) : \boldsymbol{\lambda}\mathrm{f} : \mathrm{f\ a}\ (>\mathrm{T})$

Backward Type-raising: NP : a $\rightarrow$ T $\setminus$ (T/NP) : λf : f a ($<$T)

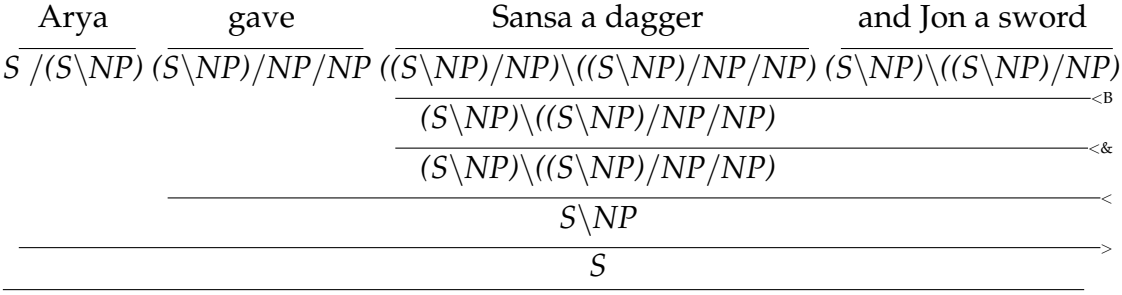
1. Subjects must be type-raised:

Figure 2.6 CCG forward subject type-raising example



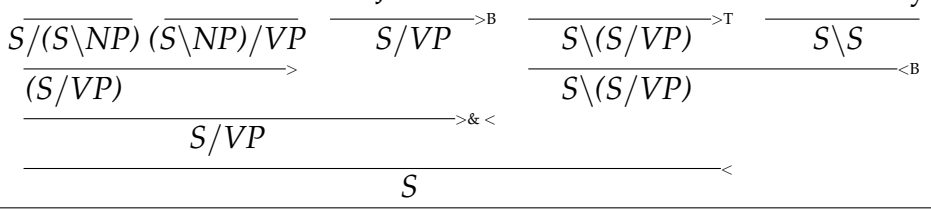
2. Objects (both direct and indirect) must be type-raised:

Figure 2.7 CCG object type-raising example



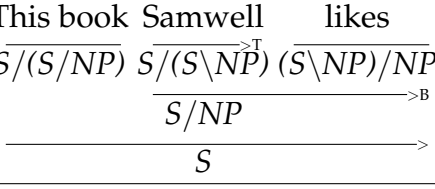
3. Complements must be type-raised too:

Figure 2.8 CCG complement type-raising example



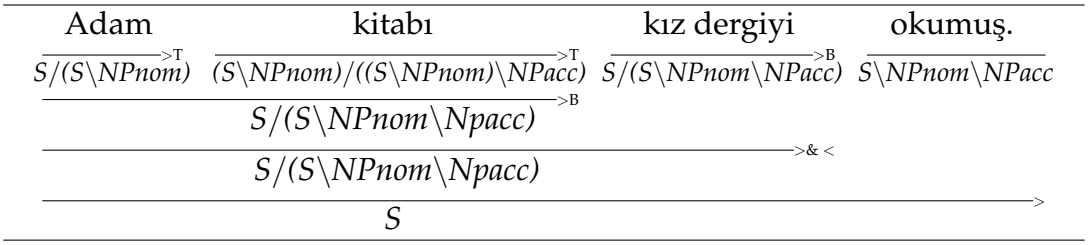
4. Type-raising is not a construction-specific requirement:

Figure 2.9 Type-raising is not construction specific



5. Type-raising is required in all languages:

Figure 2.10 Type-raising is universal



The subject type-raising rule provided in Figure 2.6 keeps the order of the combining words because it allows the argument NP (in the given rule's case) to seek an argument to its right, keeping the subject and predicate order linear. The semantic interpretation of all combinatory rules is fully determined by the Principle of Type Transparency; All syntactic categories reflect the semantic type of the associated logical form and all syntactic combinatory rules are type-transparent versions of one of a small number of semantic operations over functions including application, composition and type-raising. That's why a type-raised subject NP will have a correct way of being composed with predicates to be able to combine with the object *leg days* to form *does' leg days' arya'* in Figure 2.6. We know that true linking verbs like *become, seem* and *suppose* can compose with *Arya does* as in (a), meaning it may end in infinite right-node raising. Or in an example like (b), it may culminate in an infinite number of leftward extractions.

(a) [Sansa skips] but [you suppose Arya does]

(b) Leg days which [you suppose Arya does]

In English, type-raising and composition are used to successfully extract wh-words and to do right-node raising and argument cluster coordination. It is necessary to note that when there is type-raising, there will be a need for a composition because they work in pairs (Hockenmaier and Bisk, 2010).

2.3 What is the Problem with TR

Type-raising may cause misunderstandings especially regarding when to type-raise and when to combine. In order to avoid a structure like "Sansa speaks because Valyrian, she enjoys Old Valyria.", some restrictions should be made when combining. In this sentence, *because* has the category **S/S/S** (forward crossing composition >B) which shall not be composed with backward type-raising <T *Valyrian* **S \ (S/NP)**, vice-versa is also disallowed, <B and >T (Hockenmaier and Bisk, 2010).

The problems regarding type-raising are:

a. Categorial ambiguity:

There may be more than one usable category usable. e.g., **S,NP**.

b. Spurious ambiguity:

Categories may lead to more than one derivations that end up having the same semantics (Wittenburg, 1987). *Mary saw Ayla* has a derivation of both left and right branching S which has a semantics equal to *saw Ayla Mary*. A solution to this may seem obvious; pick one of the derivations doing a, for example, reducing the first strategy, but then there arises another problem. The constituent we would have in hand may not find an argument to combine with itself, making all the analysis obligatory again. To find a genuine distinct reading, it seems that all the spurious analyses must be discovered (Pareschi and Steedman, 1987). Parsing solutions to this problem are stated in chapter 3. Steedman shows that these "spurious" readings are phonologically non-redundant.

c. Genuine ambiguity:

(i) Lexico-semantic ambiguity: A lexical category may cause more than one semantic relation.

(ii) Attachment ambiguity: More than one derivations of the same type of category can have different semantics, e.g. PP attachment.

People that are working with CCG have tried to find ways to solve these ambiguities. The section 'Methods of TR' in the Materials and Methods chapter lists the methods proposed to solve the problems arising with TR.

2.4 CCGlab

CCGlab is a tool for experimenting with CCG. CCGlab grammars are written in paper-style and the results are almost in paper format (Bozsahin, 2019). The tool is coded in COMMON LISP and it implements all the combinators of CCG (application, combination, substitution).

Source code and the manual are at Github page of Bozsahin (2015).

2.5 CHILDES Database and Brown Corpus

The CHILDES database is a collection corpora of daily speech between children and their parents (Macwhinney, 2000). It was created in 1973 and includes part-of-speech tags. Because the tool used as a CCGlab plug-in needs to be tested with both artificial and natural language, this corpus is of great value in its ability to simulate and work with spoken language. Among the corpora, Eve corpus was selected (Brown, 1973). It has both the child's and parent's utterances. An extract from the transcription of a random session is shown in Figure 2.11.

Creating a lisp ready object out of this corpus was done manually by Sakirogullari (2019). Grammar was written as shown in the chapter Materials and Methods. Derivations were gathered using this database's converted grammar.

Figure 2.11 Extract from original Eve Corpus

@loc: /Brown/Eve/011100b.cha
@PID: 11312/c-00034754-1
@Begin
@Languages: eng
@Participants: CHI Eve Target_ Child , MOT Sue Mother , FAT David Father , COL Colin Investigator
@ID: eng|Brown|CHI|1;11.00|female|||Target_ Child|
@ID: eng|Brown|MOT||female|||Mother|||
@ID: eng|Brown|FAT||male|||Father|||
@ID: eng|Brown|COL|||Investigator|
@Date: 25-MAR-1963
@Time Duration: 11:15-12:15
*CHI: Fraser go sit right there .
%mor: n:prop|Fraser v|go v|sit adv|right adv|there .
%gra: 1|2|SUBJ 2|0|ROOT 3|2|OBJ 4|5|JCT 5|2|JCT 6|2|PUNCT
%add: MOT
%gpx: indicating couch
*MOT: alright . [+ IMIT]
%mor: co|alright .
%gra: 1|0|INCROOT 2|1|PUNCT
*CHI: I go get a pencil .
%mor: pro:sub|I v|go v|get det:art|a n|pencil .
%gra: 1|2|SUBJ 2|0|ROOT 3|2|COMP 4|5|DET 5|3|OBJ 6|2|PUNCT
*MOT: what're you gonna do ?
%mor: pro:int|whatãux|be&PRES pro:per|you part|go-PRESP̃inf|to v|do ?
%gra: 1|4|SUBJ 2|4|AUX 3|4|SUBJ 4|0|ROOT 5|6|INF 6|4|COMP 7|4|PUNCT
*CHI: get a pencil . [+ RES]
%mor: v|get det:art|a n|pencil .
%gra: 1|0|ROOT 2|3|DET 3|1|OBJ 4|1|PUNCT
*CHI: Fraser (.) I get some .
%mor: n:prop|Fraser pro:sub|I v|get qn|some .
%gra: 1|3|LINK 2|3|SUBJ 3|0|ROOT 4|3|OBJ 5|3|PUNCT
%act: coming back with one pencil
*CHI: Fraser (.) I get xxx a pencil .
%mor: n:prop|Fraser pro:sub|I v|get det:art|a n|pencil .
%gra: 1|3|LINK 2|3|SUBJ 3|0|ROOT 4|5|DET 5|3|OBJ 6|3|PUNCT
*COL: what's that you're writing ?
%mor: pro:int|whatc̃op|be&3S pro:rel|that pro:per|youãux|be&PRES part|write-PRESP ?
%gra: 1|2|SUBJ 2|0|ROOT 3|6|LINK 4|6|SUBJ 5|6|AUX 6|2|CPRED 7|2|PUNCT
*CHI: that . [+ RES]
%mor: comp|that .
%gra: 1|0|INCROOT 2|1|PUNCT

CHAPTER 3

MATERIALS AND METHODS

3.1 Methods of TR

There are various ways to type-raise, which are listed below.

3.1.1 Memoization

In computer science, memoization is a dynamic programming technique to keep the outputs of an expensive function call mostly in a hash table, in order to get it back again when that function is invoked again. In the literature Nakatsu and White (2010) referred to their method of parsing a grammar as DCCG and they use this technique in openCCG parser.

3.1.2 Chart Based Parsing

The processes followed in this parsing technique are first extracting a lexicon, i.e. a mapping from words to sets of lexical categories, and then manually defining the combinatory rule schemas, such as functional application and composition, which combine the categories together. The derivations in the treebank are then used to provide training data for the statistical disambiguation model (Hockenmaier and Steedman, 2005). This is the method used in the C&C parser tool.

When selecting the type-raising rules in their parser C&C, Stephen Clark and James Curran checked the most used type-raising categories on CCGbank and utilized those as the type-raising rule set (Hockenmaier and Steedman, 2005). The categories are presented in Table 3.1.

3.1.3 Normal Form Parsing

Normal form parsing is not a type-raising method but it depends on type-raising. A word in a grammar can have more than one category leading to

categorial ambiguity. According to the sentence to be derived, a specific category can be selected and this ambiguity can be surpassed. All that is left would be spurious ambiguities. Eisner (1996) presents a normal-form parsing algorithm for CCG that has constraints in combining words to solve spurious ambiguity. His solution does not have grammatical type-raising in its rule set (It can still have categories like S/(S\ NP), but there is no system to change derived NP to e.g. S/(S\ NP) like our plug-in does) and he states that all of the spurious ambiguities are eliminated except the chain categories like A/B B/C C/D. Hockenmaier and Bisk (2010) introduced a new normal form for CCG by adding a couple of more constraints. By using standart CKY parsing, they eliminated chained category's spurious ambiguity as well.

3.1.4 Proliferation of TR

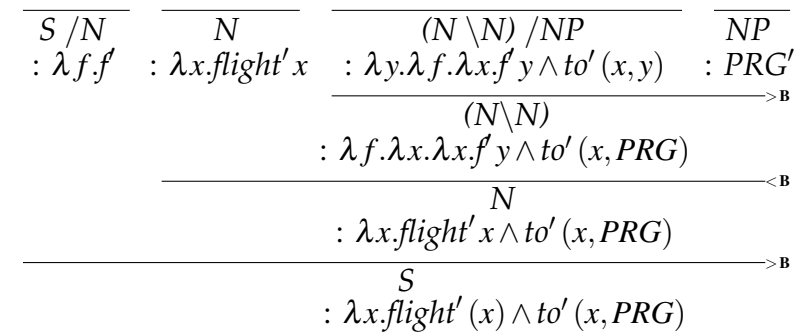
The dynamically type-raising of categories is possible. The use of polymorphic forms has the advantage of coming with very specific rules and thus preventing the proliferation of categories.

3.1.5 Online Learning

In their paper Online Learning of Relaxed CCG Grammars for Parsing to Logical Form, Zettlemoyer et all (2005) introduced a new mapping to logical form from sentences. When a training example like "Show me flights to Prague" is given, the mapped output is; λ x.flight(x) $\wedge$ to(x, PRG).

Logical connectors: conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$)

Figure 3.1 Composition example



A challenging problem with this is that derivations are not annotated. Their approach is to extend what was proposed earlier, which is learning a lexicon and the parameters for a weighted CCG (Zettlemoyer and Collins, 2005).

For a mapping like above, learning CCG works well for complex, grammatical sentences. For example;

Input: Show me flights from Newark and New York to San Francisco or Oakland that are nonstop.

Output: λx.flight(x) $\wedge$ nonstop(x) $\wedge$ (from(x,PRG) $\vee$ from(x,NYC)) $\wedge$ (to(x,SFO) $\vee$ to(x,OAK))

But what if the sentence is written in a way that only editing could correct its grammar. For example;

Input: Boston to Prague the latest on Friday.

Output: argmax(λ x.from(x,BOS) $\wedge$ to(x,PRG) $\wedge$ day(x,FRI), λy.time(y))

Given a log-linear model with a CCG lexicon A, a feature vector f, and weights w, the best parse is:

y^* = argmax $w.f(x,y)$

where we consider all possible parses y for the sentence x given the lexicon A.

It generates all the possible substrings and matches them across categories that trigger on the logical form. The normally given categories in Figure 3.2

Figure 3.2 Categories given for *Show me the latest flight from Boston to Prague*

Show	me	the	latest	flight	from	Boston	to	Prague	on	Friday
$\overline{S/NP}$	$\overline{NP/N}$	$\overline{N}$	$\overline{N\backslash N}$		$\overline{N\backslash N}$		$\overline{N\backslash N}$			

will not parse the sentence in Figure 3.3.

Figure 3.3 Categories for *Boston to Prague the latest on Friday*

Boston	to	Prague	the	latest	on	Friday
$\overline{NP}$	$\overline{N\backslash N}$	$\overline{NP/N}$	$\overline{N\backslash N}$			

They add two rules to relax the combination of categories. By reversing the direction of the principal categories and inserting the missing semantic context, as well as bypassing missing nouns, it goes to a complete parse. The algorithm gives points to those which are grammatical, and takes points from those which are using relaxed parsing rules. It is able to select a subset of the generated possible substrings and categories, and processes the data set one example at a time. Their TR is map-like type-shifting, however. It always has extra semantics than $\lambda\ f.fa$

Training and checking the correctness from the formula y^* = argmax $w.f(x,y)$, it sees if lexicon A is a subset of the generated strings and logical form match, sets this lexicon as the main lexicon A, and updates its parameters (weight w, y' and $f(xi,y)$). At the end, the output is the remaining lexicon A and parameters w (Zettlemoyer and Collins, 2007).

3.2 Tools to work with CCG

3.2.1 OpenCCG

OpenCCG is an open source natural language processing library written in Java, which provides parsing and realization services based on Mark Steedman's Combinatory Categorial Grammar (CCG) formalism (Steedman, 2000). The library makes use of the multi-modal extensions to CCG devised by Jason Baldridge in his dissertation (Baldridge, 2002) and in a joint EACL-03 paper with Bozsahin et al. (2006)

This tool uses some of the rules by default and following text is copied directly from openCCG's Github page (White, 2012) which gives a clear idea on used rules.

" *Application, composition and crossed composition (forward and backward in each case), as well as forward type-raising from NP to S/(S\NP) and backward type-raising from NP to S$1\(S$1/NP).*

A backward type-raising rule from PP to S$1\(S$1/PP). The $ causes a dollar-sign raise category to be created, as shown; without it, we'd just get S\(S/PP). Type-raise - $: PP => S; "

As they stated in the code file on their Github page **ccg-format-grammars/ tiny/tiny.ccg**, their type-raising rules are hard-coded and are not extended beyond the stated ones.

3.2.2 Clark & Curran Parser

The C&C, CCG parser and supertagger form part of the language processing tools developed by Clark and Curran (2007). The tools are written in C++ and have been designed to be efficient enough for large-scale NLP tasks. Combinatory rules as well as type-raising rules are selected by looking at the occurrence frequency of the type-raising rules on CCGbank. Disadvantages of this approach is discussed in Chapter 4.

Table 3.1: Set of Unary Rules used by Clark and Curran

Category	Type-raising rule
NP	S/(S\NP)
NP	(S\NP)\((S\NP)/NP)
NP	((S\NP)/NP)\(((S\NP)/NP)/NP)
NP	((S\NP)/(S[to]\NP))\(((S\NP)/(S[to]\NP))/NP)
NP	((S\NP)/PP)\(((S\NP)/PP)/NP)
NP	((S\NP)/(S[adj]\NP))\(((S\NP)/(S[adj]\NP))/NP)
PP	(S\NP)\((S\NP)/PP)
S[adj]\NP	(S\NP)\((S\NP)/(S[adj]\NP))

3.2.3 easyCCG

EasyCCG is a CCG parser created by Mike Lewis and the source code is available at Github page of Lewis (2014). It uses factored lexical categories and the parsing is then just a deterministic search and followed by the selection of the highest probability category sequence that supports a CCG derivation.

The parser uses Steedman (2000)'s combinatory rules *(forward application, backward application, forward composition, backward crossed composition, generalized forward composition, generalized backward crossed composition)* because those are put forward as linguistically universal. As type-raising rules, only the ones shown in Table 3.2 are used (Lewis and Steedman, 2014).

Table 3.2: Set of Unary Rules used by easyCCG

Initial	Result	Usage
NP	S/(S\NP)	Type-Raising
NP	(S\NP)/((S\NP)/NP)	
PP	(S\NP)/((S\NP)/PP)	

3.3 Procedure and Algorithm

As stated in the previous chapters, CCG gives words the opportunity to combine and eventually compose sentences using combinatory rules and type-raising. This tool generates and adds the type-raising rules above the already given grammar.

Inside our tool, a function takes a path as an argument to a file (the file types that the tool works with are explained below in the Data Structures section). It uses the predefined rules to generate new type-raised categories. The workflow for this generation process as well as a description of how to use the new generated rules are also provided.

3.3.1 Data Structures

The input to the algorithm is a ccglab `.ded` file format. CCGlab file formats for a file named *P* are as follows;

– `P.ccg` : This is the grammar provided to ccglab written by the user themself. The user specifies the categories of the units in accordance with the rules in the ccglab manual.

– `P.ded` : Short for deduction, this is what the program uses as an input. It has a nested association list data structure that helps load into Lisp directly. An association list is a list of pairs, and each pair is a key value association. This helps the tool use listing comprehensions, pop and push the items to the

rule set while not changing the original copy. To be able to interpret what the `.ded` file's contents are, two tables 3.3 and 3.4 are good references to look at.

– `P.ind` : Short for induction, the same grammar but in a model-trained state. This is to use the generated rules and derivations as parse ranking and has the same format as `P.ded`.

– `P.lisp` : This is our code file and will be further explained in the Workflow section.

Having described the file formats, the next step is to identify the code parts and see what those parts do.

The lisp file is a plug-in for CCGlab. Its role is to enable type-raising on selected verb morphemes. Work flow is explained in section 3.3.2 and the algorithm is explained in section 3.3.3. An abstract data type approach has been implemented to put the algorithm to use.

3.3.2 Workflow

The examples below are taken directly from the CCGlab manual (Bozsahin, 2019). A `.ccg` file which is a raw input to CCGlab consists of the elements shown after this paragraph. The top 3 are called lexical items, and the bottom one is called a unary rule. Only unary rules care about our order of specifications while the whitespacing is not important at all. As it parses, the latter parsed unary rules can reach the former parsed rules. Also the same category does not apply again once it already has.

John n := np[agr=3s] : !john ;
likes v := (s\np[agr=3s])/ ^ np : \x\y. !like x y;
and x := (@X*@X)/*@X : \p\q\x. !and (p x)(q x);
(L1) np[agr=?x] : lf –> s/(s\np[agr=?x]) :\lf \p. p lf ;

We mentioned unary rules because type raising is unary.

Having written the grammar and saved it as a my_grammar.ccg file, we run CCGlab and type

*(load-grammar "my_grammar" :make t)

which creates my_grammar.ded and loads it onto Lisp.

OR

* (load-model "my_grammar" :make t)

which creates my_grammar.ind and loads it onto Lisp.

Let us assume that the upper option was typed and now `my_grammar.ded` is ready. Since both options have the same workflow, there is no need to divide

them.

(1) The first step to use the compile-tr.lisp is to load it by doing;

* (load "compile-tr.lisp")

(2) The main method to type-raise categories gets two parameters. First argument is the path to our `my_grammar.ded` and the second argument to it is the Lisp list of "to be type-raised" categories, henceforth it will be mentioned as *"morphemes' list"*. If the file `'my_grammar.ded'` is in the path /home/docs and desired morphemes are V, V3, Ving then it should be typed;

* (compile-tr "/home/docs/my_grammar.ded" '(V V3 Ving))
File created at doc/raised-lex-rules.ded
The rules also set to the global variable *RAISED-LEX-RULES*

It prints two instructive messages. The above statement says that in the path where `'compile-tr.lisp'` is a doc folder was created and a file `raised-lex-rules.ded` was generated. The below statement says that in case we want to manipulate the rules generated by the `compile-tr` method, we have an option to do so by editing or using the variable `*RAISED-LEX-RULES*`. Type-raising rule generation is achieved with these two simple function calls.

(3) The lisp translation of the rules are kept in the global variable `*ccg-grammar*`. When we do `(load-grammar)` in CCGlab, it initializes and fills `*ccg-grammar*` (global variables and their descriptions are in Table 3.5). Because of this, when the function `compile-tr` is called, `*ccg-grammar*` is initialized and filled with the rules using loaded grammar. `compile-tr` function makes use of this variable to extract the category information and to start generating type-raising rules. However, it does not alter `*ccg-grammar*`. There is a method to do this. It is called `(add-tr-to-grammar)` and all it does is push the new set of rules kept in `*RAISED-LEX-RULES*` to the end of `*ccg-grammar*`.

* (add-tr-to-grammar)
Type-raising rules added at the end of *ccg-grammar*

Table 3.3: Lex Item Features

LEX ITEM FEATURES	
KEY	unique key for an entry
PHON	phonological form
MORPH	POS tag
SYN	syntactic type
SEM	logical form
INDEX	a unique value

(4) After pushing the newly generated rules at the end of `*ccg-grammar*`, what should be done is to save the created grammar. Let us call the new grammar `grammar_withTR.ded`.

* (save-grammar "grammar_withTR.ded")

(5) Loading the saved grammar again to make sure the grammar that is dealt with is the one that is created just now. Be aware of the naming since `load-grammar` does not use extensions when writing the file name, while `save-grammar` in step (4) needs an extension.

* (load-grammar "grammar_withTR")

The rules are ready to be used to parse new sentences.

3.3.3 Algorithm to type-raise syntactic categories

The algorithm is provided by Cem Bozsahin. Figure 3.4 shows a representation of the provided algorithm and source code is at Github page of Demir (2019).

Figure 3.4 Algorithm to type-raise syntactic categories

input: a CCG grammar, Part-of-speech(POS) categories for verbs.
output: list of unary rules for type-raising all arguments of the verbs in the input.
method:

```
for each verb morpheme do
    for each argument in the verb do
        create a new item consisting of (category without the argument)
        (opposed direction of the argument) (entire category)
        store the new item
        trim the argument from the category
    end for
end for
```

The algorithm generates all the rules and works in all the arbitrary argument inputs. This is for completeness. If there is no such rule, it will not add anything to the end of the grammar. It also gives the argument type-raising correctly. We tested the system with various grammars, including the large scale Eve database which has 4054 already defined rules.

3.3.4 Detailed code analysis and complexity discussion

Here is the main function's step by step detailed execution plan that explains how the type-raising is accomplished. An example `*ccg-grammar*` is provided in Figure 3.5 to show a snippet.

(1) At the start of the program, `*ccg-grammar*` is loaded as an input together with the user provided *morphemes' list*.

(2) Once the input is loaded, the verb morpheme finding function is called. It is a straightforward searching function that goes through all the entries- **(KEY 1), (KEY 2) ... (KEY N)**- on `*ccg-grammar*`, then in each one of them, it checks if any of the morphemes given in the user provided list (*morphemes' list*) equals that of `*ccg-grammar*`. Therefore, it has O(N) complexity. This function stores the resulting items in `*VERBS-IN-GRAMMAR*`.

(3) A function gets the biggest key id and stores it in `*last-key-id*` for later use. Iterates to the end of the list, so the complexity of this loop is O(N).

(4) After storing the biggest key id of `*ccg-grammar*`, a nested loop (shown in 3.4) that seeks the syntactic part to be type-raised - namely **(SYN X)** pair- is executed. The outer loop goes through the list that is initialized in the step 2 (named `*VERBS-IN-GRAMMAR*`) and the inner loop goes through the arguments of current verb morpheme.

Since the inner loop's iteration takes much less than the outer loop (a table that shows the relation between the max number of arguments versus the number of categories provided at chapter 4), inner loop's complexity can be assumed constant O(1). The outer loop takes O(N) time to execute, so this step becomes O(1)*O(N)*O(complexity for the type raising operation). Inner loop is where the type-raising function is called. Type-raising function's complexity is constant O(1), because what it does is simple list comprehension.

This brings us to the conclusion that the total complexity of this nested loop is O(1)*O(N)*O(1) = O(N)

The whole program takes O(N) + O(N) + O(1)*O(N)*O(1) $\approx$ O(N)

3.3.5 Type-raising function implementation details

Having talked about the complexity of the type-raising algorithm function, here is how the function itself is implemented;

When the syntactic category is obtained-**(SYN X)** pair where **SYN** is key and **X** is the value-, `(type-raise category)` function starts the recursive type raising process.

(a) As the base case of a recursive function: it checks if the category is an atomic category. If it is atomic (N, NP, S, P etc..), there is no argument left, meaning type-raising must stop.

(b) If the category is complex ((S/NP),(S\NP)/V), etc.,) the direction in which the argument is expected is noted.

(c) The argument to be type-raised is 'popped' from the category and the new category is formed according to the direction (category without the argument) (opposed direction of the argument) (entire category)

(d) The new category and the last type-raised argument are saved in variables.

(e) The type-raising function is called again but this time without the last type-raised argument. The process restarts from (a) until it hits the base case.

After the type raising function reaches the base case and type raising stops, the resulting values are saved. A new syntactic category and the semantics of type-raised morpheme are set and an index is assigned to the new category. This entry is pushed to the `*RAISED-LEX-RULES*`.

When the iterations are over, the resulting `*RAISED-LEX-RULES*` is saved as a file.

As an example, what happens to a category when the type-raising function gets called using the example.ded file in Figure 3.5 is shown here;

Assume `compile-tr` is called with the argument '(V) meaning the tool gets to type-raise only the rules that have the morpheme V (MORPH V).

```
* (compile-tr "doc/example.ded" '(V))
```

Grammar is loaded, `*ccg-grammar*` is initialized. Since there is only one key entry matching the item in our file - that is **(KEY 4)** with the (MORPH V)-, that item would be the only one to be inserted into the list `*verbs-in-grammar*`.

```
* *verbs-in-grammar*
```

(**(KEY 4)** (PHON ANSWER) (MORPH V) (SYN (((((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL))) (DIR BS) (MODAL ALL) ((BCAT AUX) (FEATS ((TYPE TEMP))))) (DIR FS) (MODAL ALL) ((BCAT NP) (FEATS NIL))))
(SEM (LAM X (LAM Y (LAM Z ((("SIMP" Y) "ANSWER") X) Z))))) (PARAM 1.0))

Then

`(type-raise` '(((((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL)))
(DIR BS) (MODAL ALL) ((BCAT AUX) (FEATS ((TYPE TEMP)))))
(DIR FS) (MODAL ALL) ((BCAT NP) (FEATS NIL))))

is called.

As a base case of a recursive function, `(type-raise)` checks if the category provided is complex or atomic. To do this, it checks if there is a **DIR** tag in the category.

((BCAT S) (FEATS ((TYPE INF)))) **(DIR BS)** (MODAL ALL) ...

Having found a **DIR** tag, it interprets this as there being at least one argument

to type-raise. The argument is the part that comes after the latest 1st level **DIR**. 1st level here means that it should reside in the outermost part of the nested parenthesis.

Here is what the category in `*verbs-in-grammar*` should look like on paper;

((S\NP)\AUX) / **NP**

The outermost **DIR** is the rightest forward slash. The argument is **NP** whose lisp representation is ((BCAT NP) (FEATS NIL)).

After type-raising this argument, the result should look like this;

((S\NP)\AUX) \ ((S\NP)\AUX) / **NP**

This is a new rule and will be stored in the `*raised-lex-rules*` variable. With the remaining rule (the last argument is dropped from the category), `(type-raise category)` gets called again.

[Stack 1] `(type-raise` '((((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL)))
(DIR BS) (MODAL ALL) ((BCAT AUX) (FEATS ((TYPE TEMP))))))

First, it again checks if it is a complex cat. It finds a **DIR** tag and detects the argument by checking the outermost level **DIR** which is being shown here

(DIR BS) (MODAL ALL) ((BCAT AUX) (FEATS ((TYPE TEMP))))

Then it starts type-raising.

(S\NP) \ **AUX**

is our category. **AUX** is the argument.

After the operation, it becomes;

(S\NP) / ((S\NP) \ **AUX**)

This will be pushed back to the `*raised-lex-rules*`. Remaining (S\NP) is given to the recursive type-raise function.

[Stack 2] `(type-raise` '(((BCAT S) (FEATS ((TYPE INF))))
(DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL)))

It still is a complex category since there is a **DIR**
(DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL))

Therefore it starts the operation on,

S**NP**

NP is the argument. When it finishes the list comprehension, the output is:

S / (S\NP)

which is added to `*raised-lex-rules*`. After adding the rule, the next line is a call to `(type-raise category)`

[Stack 3] `(type-raise` '((BCAT S) (FEATS ((TYPE INF)))))

This will hit the base case, since it will not be able to find any **DIR** tag. Execution will be stopped and the function that is 3 stacks deep will return.

[Stack 2] Execution ends,there is no more line left to execute because last line was a call to `(type-raise)`.

[Stack 1] Execution ends, no more line left to execute.

```
* (compile-tr path category)
```

Since all recursive levels returned, the main function will stop and all that is left is `*raised-lex-rules*` with its content filled with new type-raising rules.

```
* *raised-lex-rules*
```

(((**KEY 6)** (INSYN ((BCAT NP) (FEATS NIL))) (INSEM LF) (OUTSYN (((BCAT S) (FEATS ((TYPE INF)))) (DIR FS) (MODAL ALL) (((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL))))) (OUTSEM (LAM LF (LAM P (P LF)))) (INDEX # :|AUTO-TR398|) (PARAM 1.0))

((**KEY 7)** (INSYN ((BCAT AUX) (FEATS ((TYPE TEMP))))) (INSEM LF) (OUTSYN ((((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL))) (DIR FS) (MODAL ALL) ((((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL))) (DIR BS) (MODAL ALL) ((BCAT AUX) (FEATS ((TYPE TEMP))))))) (OUTSEM (LAM LF (LAM P (P LF)))) (INDEX # :|AUTO-TR399|) (PARAM 1.0))

((**KEY 8)** (INSYN ((BCAT NP) (FEATS NIL))) (INSEM LF) (OUTSYN (((((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL))) (DIR BS) (MODAL ALL) ((BCAT AUX) (FEATS ((TYPE TEMP))))) (DIR BS) (MODAL ALL) (((((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL))) (DIR BS) (MODAL ALL) ((BCAT AUX) (FEATS ((TYPE TEMP))))) (DIR FS) (MODAL ALL) ((BCAT NP) (FEATS NIL))))) (OUTSEM (LAM LF (LAM P (P LF)))) (INDEX # :|AUTO-TR400|) (PARAM 1.0)))

Next is to see how these rules are used in the derivation of sentences. This will be explained in the Chapter 4.

3.3.6 Algorithm to type-raise semantic categories

Semantic categories' type raising is more straightforward. Any semantic type, when it is type-raised, becomes

$X : a \quad \Rightarrow_{\mathbf{T}} : \lambda f.fa$

The grammar that Figure 3.5 represents is shown in another Figure 3.6

Figure 3.5 Example *ccg-grammar* variable contents

(DEFPARAMETER *CCG-GRAMMAR*

'(((**KEY 1**) (PHON ABC) (MORPH N)
(SYN ((BCAT N) (FEATS ((TYPE COUNT) (AGR 3) (COUNT SG)))))
(SEM "ABC") (PARAM 1.0))

((**KEY 2**) (PHON ABCS) (MORPH PLN)
(SYN ((BCAT NP) (FEATS ((TYPE COUNT) (AGR 3) (COUNT PL)))))
(SEM ("PL" "ABC")) (PARAM 1.0))

((**KEY 3**) (PHON ANY) (MORPH DET)
(SYN (((BCAT NP) (FEATS ((TYPE ?X) (AGR ?Y) (COUNT ?Z)))) (DIR FS) (MODAL ALL) ((BCAT NP) (FEATS ((TYPE ?X) (AGR ?Y) (COUNT ?Z))))))
(SEM (LAM X ("ANY" X))) (PARAM 1.0))

((**KEY 4**) (PHON ANSWER) (MORPH V)
(SYN (((((BCAT S) (FEATS ((TYPE INF)))) (DIR BS) (MODAL ALL) ((BCAT NP) (FEATS NIL))) (DIR BS) (MODAL ALL) ((BCAT AUX) (FEATS ((TYPE TEMP))))) (DIR FS) (MODAL ALL) ((BCAT NP) (FEATS NIL))))
(SEM (LAM X (LAM Y (LAM Z ((("SIMP" Y) "ANSWER") X) Z)))) (PARAM 1.0))

((**KEY 5**) (PHON ABOUT) (MORPH PRE)
(SYN ((((BCAT S) (FEATS NIL)) (DIR FS) (MODAL ALL) ((BCAT S) (FEATS NIL))) (DIR FS) (MODAL ALL) ((BCAT NP) (FEATS ((TYPE TIME))))))
(SEM (LAM X (LAM Y (("TIME" Y) ("ABOUT" X))))) (PARAM 1.0))

((**KEY 6**) (PHON ABOUT) (MORPH PRE)
(SYN ((((BCAT S) (FEATS NIL)) (DIR BS) (MODAL ALL) ((BCAT S) (FEATS NIL))) (DIR FS) (MODAL ALL) ((BCAT NP) (FEATS ((TYPE TIME))))))
(SEM (LAM X (LAM Y (("TIME" Y) ("ABOUT" X))))) (PARAM 1.0)))

Figure 3.6 Example grammar written for CCGlab

```
abc n := n[type=count,agr=3,count=sg] : !abc ;
abcs pln := np[type=count,agr=3,count=pl] : !pl !abc ;
any det := np[type=?x,agr=?y,count=?z]/np[type=?x,agr=?y,count=?z] :
\x.!any x;
answer v := s[type=inf]\np\aux[type=temp]/np : \x\y\z.!simp y !answer x z;
about pre := (s/s)/np[type=time] : \x\y.!time y (!about x);
about pre := (s\s)/np[type=time] : \x\y.!time y (!about x);
```

This is why in the compile-tr.lisp plug-in, the semantic type-raising is hard coded. All semantic categories are turned into (LAM LF (LAM P (P LF))) in the ded file when `(compile-tr path morphs)` function is used. When `(debug-tr path morphs)` is used, however, semantic representation is filled with the argument that has been type-raised in that step.

Table 3.4: Lex Rule Features

LEX RULE FEATURES	
KEY	unique key for an entry
INSYN	input category
INSEM	input logical form
OUTSYN	output syntactic category rule
OUTSEM	output logical category rule
INDEX	a unique value

Table 3.5: Compile-TR global variables

Global variables	
`*ccg-grammar*`	all the lexical entries after loading a grammar to CCGlab
`*verbs-in-grammar*`	selected list of lexical entries according to *"morphemes' list"*
`*syns*`	type-raised SYN entries of a morph
`*args*`	list of arguments of categories that have already type-raised
`*last-key-id*`	highest KEY id in *ccg-grammar*
`*raised-lex-rules*`	type-raised lexical rules including INSYN OUTSYN INSEM OUTSEM

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Results and Examples

For the very first experiment, a simple grammar can be picked. This is to get an idea of what is trying to be achieved and what the resulting derivations would look like when the type-raising plug-in is used. A grammar called corner.ccg is provided here;

```
the d := np/*n: \x. x;
man n := n: !man;
in p := (n\n)/*np: \x\y.!in x y;
corner n := n: !corner;
hits v := (s\np)/np: \x\y.!hits x y;
```

After loading the grammar to CCGlab

* (load-grammar "corner")

Next is to type-raise the only verb "hits".

* (load "compile-tr")

* (compile-tr "corner.ded" '(V))

* (add-tr-to-grammar)

Grammar added at the end of *ccg-grammar*

*(save-grammar "corner-tr.ded")

Now that it saved our grammar with the newly type-raised rules added at the end of `*ccg-grammar*`.

The grammar had 5 words in total, and for the 2 arguments, it generated 2 different rules. The added items have the new type raised rules created from the arguments of the verb "hits".

This one below is generated from the argument written in italic $(S\backslash NP)/\mathit{NP}$.

a. (S \ NP) \ ((S \ NP) / NP)

This next one is generated from the argument written in italic (S*NP*)/NP.

b. S / (S \ NP)

Those arguments were both NP's but they had different functors, so the rules turned out to be different. Let us parse using a simple sentence to test it out.

* (p '(the man hits the corner))

This command returned true, which means there are derivations available for this sentence.

* (ders)

Figure 4.1 Derivation 1

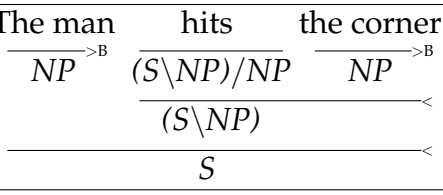


Figure 4.2 Derivation 2

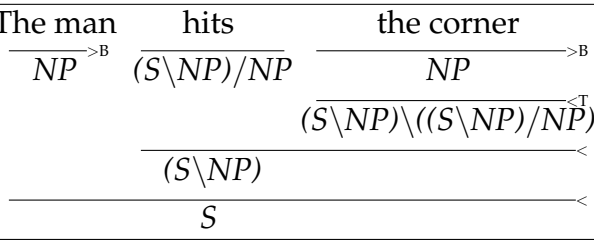


Figure 4.3 Derivation 3

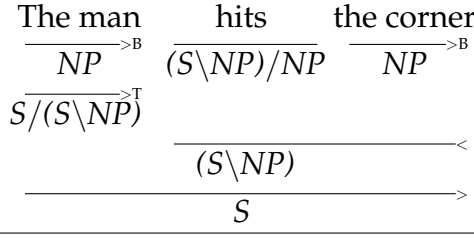
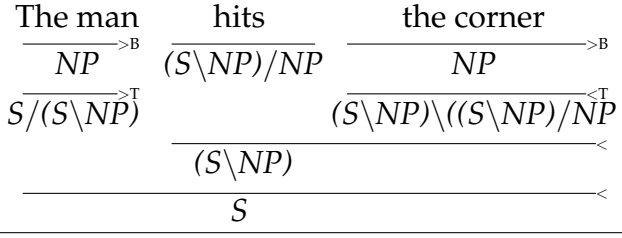


Figure 4.4 Derivation 4



The tool generates all the possible type-raising rules. Here is another example from the large scale database Eve:

If the parser is allowed to type-raise with basic categories like NP, PP, N, then the number of derivations would be more than the number of derivations when we disallow the type-raising of basic categories.

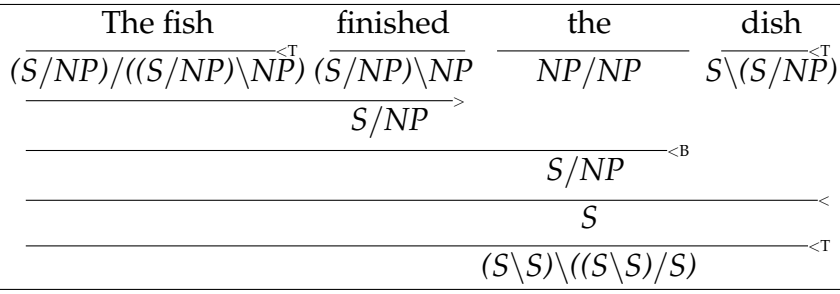
*(type-raise-off)

* (p '(the fish finished the dish))

* (ders)

This shows 4395 derivations. A random derivation is at 4.5

Figure 4.5 Derivation including basic category type-raising



However, if we do:

* (type-raise-targets '(NP PP N))

(NP PP N)

* (p '(the fish finished the dish))

* (ders)

Now the parsing without basic categories generates only 98 derivations. When the tool is not allowed to use lower types like N, NP or PP in derivations, the amount of derivations generated is drastically reduced because it basically omits the lower types and proceeds with the complex categories only.

If a sentence that is going to be parsed is longer, then as one would expect, number of derivations would become higher. This is because the number of methods to combine categories is increasing thanks to type-raising. Imagine combining two sentences with a 'but'. For example, the sentence "the fish finished the dish but did not finish the cake" has 3958 derivations. Statistics are shown in Table 4.2.

4.2 Discussions

Unlike other parsers, our tool generates type-raising rules by looking at the provided grammar. This has two advantages over other parsers. The first one is that we would have all the type-raising rules we need so that instead of waiting for our predefined rules to work and parse sentences, we would have the right ones to decide and tell if there is a derivation. The second one is that while other parsers give limited amount of type-raising rules and only one or two of them have features in their arguments, the rule generation algorithm used in this tool do not drop the features of the categories. For that reason we can also have the rules that have features in its arguments. Clark and Curran parser has only [to] and [adj] as features on their type-raising rule set. OpenCCG uses dollar sign notation to represent all the NP and PP's in the lexicon, however, arguments are not limited to only those two.

Our tool type-raises all the arguments. However, it does not check if more than one identical type-raising rule is present. Therefore, if subsumption is made, the number of rules would decrease remarkably and that would make the computation even easier. As can be seen in the Table 4.1, the tool has generated 7419 rules but if the subsumption is made, number of rules generated reduces to 523 in the Eve database (Subsumption has been designed and implemented by Cem Bozşahin). These include 2471 different verbs' arguments with their features kept.

Table 4.1: Eve and PFTL database TR Rule Generation results

	# of categories	**# of TR Rules Generated**	**# of Rules if Subsumed**
Eve	4054	7419	523
	# of Verbs	**max # of arguments**	
	2471	6	
	# of categories	**# of TR Rules Generated**	**# of Rules if Subsumed**
PFTL	251	148	47
	# of Verbs	**max # of arguments**	
	92	4	

In Table 4.2, it is shown the number of derivations when type-raising with atomic types allowed and when it is disallowed. Disabling the type-raising on atomic types decreases the number of derivations as expected.

Table 4.2: # of Derivations when TR on Atomic Types enabled / disabled

Sentence	TR w/ Atomic Types	TR w/o Atomic Types
The fish finished the dish	4395	98
The fish finished the dish but did not finish the cake	*more than allowed memory*	3958

CHAPTER 5

CONCLUSION AND FUTURE DIRECTIONS

5.1 Conclusion

Language modeling tools like CCG can be trained, and can be taught to choose the correct derivation of words using both syntactic and semantic information. The syntactic interpretation of combinatory rules are considered type - transparent. All syntactic categories reflect the semantic type of the associated logical form and all syntactic combinatory rules are type-transparent versions of one of a small number of semantic operations over functions including application, composition and type-raising. To be able to parse sentences and find different derivations from categories, these functional operations are used. Type-raising is one of these operations and it is thought to be universal. When a sentence is parsed, type-raising simulates the case where arguments like noun phrases become functions and seek other arguments like verb phrases or auxiliary verbs near them to compose and become a sentence. Type-raising is not limited to subjects, objects or complements, however. It is also used in non-complete sentences thanks to its being not a constructive requirement. Additionally, type-raising is not language specific. The sentences can be parsed with type-raising very often, such that, as can be seen in the Results and Discussions chapter, a simple example gives 4 derivations and 3 of them are generated by type-raising rules. The plug-in integrated to CCGlab was created in order to experience such types of cases. Instead of using specific type-raising rules, it generates all the rules with the features included. This is what separates it from all the other tools. Having an intermediate step filled using this plug-in, CCGlab can be used to examine the sentences derived by type-raising rules faster and more efficient.

5.2 Future Directions

The implemented tool correctly generates all the type-raised categories using all the arguments of verbs as intended. However, there are arguments with the same features and the same type of functors that when they are type-raised, they give the exact same rule. These can be eliminated and a subsumption can be found to reduce the number of rules. This would help

reduce the number of derivations. A result was reported in 4.2.

Bibliography

Baldridge, J. (2002). *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. PhD thesis, University of Edinburgh.

Bever, T. (1970). *The Cognitive Basis for Linguistic Structures*. Wiley.

Bozsahin, C. (2015). Ccglab. `https://github.com/bozsahin/ccglab`.

Bozsahin, C. (2019). CCGlab manual. `"https://github.com/bozsahin/ccglab/blob/master/docs/CCGlab-manual.pdf`.

Bozsahin, C., Kruij, G.-J., and White, M. (2006). Specifying grammars for openccg: A rough guide.

Brown, R. (1973). *A First language: the early stages.* Cambridge, MA: Harvard University Press.

Clark, S. and Curran, J. (2007). Wide-coverage efficient statistical parsing with ccg and log-linear models. *Computational Linguistics*, 33:493–552.

Demir, O. (2019). Type-raising for ccglab. `https://github.com/karavana/cogs`.

Eisner, J. (1996). Efficient normal-form parsing for combinatory categorial grammar. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 79–86, Santa Cruz, California, USA. Association for Computational Linguistics.

Fodor, J. A. and Garrett, M. (1967). Some syntactic determinants of sentential complexity. *Perception and Psychophysics*, 7:289–296.

Harley, T. (2014). *The Psychology of Language: From Data to Theory*. Psychology Press.

Hockenmaier, J. and Bisk, Y. (2010). Normal-form parsing for combinatory categorial grammars with generalized composition and type-raising. *COLING '10 Proceedings of the 23rd International Conference on Computational Linguistics*, pages 465–473.

Hockenmaier, J. and Steedman, M. (2005). Ccgbank. `http://groups.inf.ed.ac.uk/ccg/ccgbank.html`.

Lewis, M. (2014). Easyccg. `https://github.com/mikelewis0/easyccg`.

Lewis, M. and Steedman, M. (2014). A ccg parsing with a supertag-factored model. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Macwhinney, B. (2000). The CHILDES project: tools for analyzing talk. *Child Language Teaching and Therapy*, 8.

Nakatsu, C. and White, M. (2010). Generating with discourse combinatory categorial grammar. *Linguistic Issues in Language Technology*.

Nordquist, R. (2019). What is semantic transparency? `https://www.thoughtco.com/semantic-transparency-1691939`.

Pareschi, R. and Steedman, M. (1987). A lazy way to chart-parse with categorial grammars. *ACL '87: Proceedings of the 25th annual meeting on Association for Computational Linguistics*.

Sakirogullari, C. (2019). Measuring empirical bias toward ergativity and accusativity. Master's thesis, Middle East Technical University.

Steedman, M. (1996). *Surface structure and interpretation*. MIT press.

Steedman, M. (2000). *The syntactic process*, volume 24. MIT press.

White, M. (2012). Openccg. `https://github.com/OpenCCG/openccg/`.

Wittenburg, K. (1987). Predictive combinators: a method for efficient processing of combinatory categorial grammars. *ACL '87: Proceedings of the 25th annual meeting on Association for Computational Linguistics*.

Zettlemoyer, L. S. and Collins, M. (2005). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence (UAI2005)*, pages 658–666.

Zettlemoyer, L. S. and Collins, M. (2007). Online learning of relaxed ccg grammars for parsing to logical form. *In Proceedings EMNLP-CoNLL*, pages 678––787.