博士論文

Incremental Parsing in Combinatory Categorial Grammar

谷口　雅弥

主指導教員　東条　敏

北陸先端科学技術大学院大学
先端科学技術専攻
［情報科学］

令和５年３月

Abstract

Combinatory Categorial Grammar (CCG) includes combinators in addition to Categorial Grammar (CG) to accommodate linguistic phenomena. For example, the type-raising rule is realized by a combinator in CCG to exchange the argument-functor relation, and such a rule is generalized as the Continuation-Passing Style (CPS) transformation. However, there is a concern that CPS may excessively accept ungrammatical sentences. In this thesis, we investigate the expanded grammar rules of CCG in terms of Lambek Calculus (LC). First, we show that Barker's CPS transformation is provable in LC, but Plotkin's CPS transformation is not. Second, we show a provable subset of Plotkin's CPS transformations. Due to the complexity of proving unprovability, we formalize the proof in Isabelle/HOL and verify it. We show that this subset is a grammatical class represented in LC and call it type-restricted CPS transformation.

The grammar defined in the formal system, the notion of normative grammar, is helpful for tagging sentences of a large corpus, i.e., to annotate each word by a part of speech (POS). In this research, we aim to obtain categorial grammar rules, where the category is a generalized notion of POS. However, finding a proper set of grammar rules is computationally exponential regarding the length of the sentence, and thus, a reliable but exhaustive search method is in demand. Here, we present a support system for annotation by the CCG parser based on the bidirectionality and non-determinism in logic programming. Contrary to the common usage of the parser, we extract a set of grammar rules from a syntax tree and retrieve all the probable readings.

In grammar extraction, the problem is keeping a head-dependency relation in a treebank, which is a collection of trees representing sentence constituency and dependency relation. We are motivated to extract grammar rules from the treebank, i.e., to decompose the tree data structure and to find grammar rules. After the extraction, we need to validate the adequacy of the grammar so that we inspect the generative power of the obtained grammar. In this phase, the syntactic head is a significant feature; however, the head information is missing in the obtained grammar. Hence, we propose supplementing the lost head information with the type-raising rule of the categorial grammar (CG). We extend the same issue to Combinatory Categorial Grammar (CCG) and solve it using generalized type-raising. Furthermore, we verify our grammar by the formal proof written in the proof assistant system, Isabelle/HOL.

Finally, we show the application to parse a sentence using our grammar system. Although a simple grammar system is well-inspected in computational linguistics, the extension of CG is not so concerning the computational order problem and the proof-theoretic properties. In this thesis, we reveal the computational complexity of CG, its variants, and the form of the parsing tree as a proof tree in substructural logic, especially Lambek Calculus. Our contribution is three-fold: (a) we showed the new grammar rule named type-restricted CPS transformation; (b) we gave the new grammar extraction system keeping the head-dependency relation; (c) we showed constructive (partial) proof of the left-branching derivation in categorial grammar and Lambek Calculus.

Keywords: Combinatory Categorial Grammar, Lambek Calculus, Continuation-Passing Style Transformation, Incremental Parsing, Theorem Proving

Contents

List of Figures

List of Tables

Listings

Notations

In this thesis, we employ the notion of categorial grammar extended with the part of speech (POS) such as S/NP, and the notion of lambda calculus extended with word constants such as apple.

NP	**Part of speech** and category in categorial grammar
α	Variable in categorial grammar
S/NP, $\alpha\backslash\beta$	Functional category in categorial grammar
Γ	Sequence of categories
τ	Atomic type in lambda calculus
υ	Type variable in lambda calculus
$a \rightarrow x$	Functional type in lambda calculus
word	**Word** and constant in lambda calculus
t	Constant and variable in lambda calculus

Acknowledgements

My advisor, Professor Satoshi Tojo, taught me the fundamentals of logic, linguistics, and information science, which form the basis of this project, as well as how to behave as a researcher. With his guidance, I was able to complete this project. I would like to express my gratitude to him. My secondary advisor, Professor Nguyen Le Minh, provided advice in the field of natural language processing research and introduced me to excellent workshops. My advisor for my minor research project, Professor Mizuhito Ogawa, guided me through the concepts that gave me the idea for this research, including mathematics, theorem-proving support systems, mathematical logic, and programming language theory. Professor Koji Mineshima, who was also the advisor of my minor research project, provided research guidance as an expert in formal semantics, which is the main focus of this research. Assistant professor, Hitoshi Omori, taught me how to carry out my research with comments and enthusiasm as an expert in logic. Lecturer Teeradaj Racharak, also an assistant professor, helped me throughout the project as an expert in artificial intelligence, someone who has worked in industry, a young researcher of my generation, and a graduate of the same laboratory. Professor Takako Nemoto, as an expert in mathematical logic, has been very helpful in discussing the logical aspects of the research and the careers of doctoral students. Dr. Hiroakira Ono, Professor Emeritus, provided advances to me on the foundation theory of algebraic semantics and substructural logic over a period of three years. Mrs. Hiromi Inada, who has been in charge of administrative work in Tojo's and Nguyen's laboratories, helped us with the research funding procedures and assisted us a lot in our research. I would not have accomplished this research without the help of these experts, and I would like to express my gratitude to them. No less significantly, I wish to thank Dr. Kentaro Inui, Dr. Nao Hirokawa, and Dr. Naoya Inoue for their exceptional kindness in offering their authoritative comments. This research project is supported by the JAIST DRF program and JSPS DC2 program. These financial supports helped me in my life and studies during the COVID-19 era, and I am grateful for them. I would like to thank the seniors and juniors in Tojo's and Nguyen's laboratories who have worked with me for the past five years. I really enjoyed doing research with them, and they helped me many times in my research life. During my doctoral program, I had the opportunity to volunteer as a learning assistant at the Seirei Aijien (Children's Home of the Holy Spirit). The experience in elementary education certainly improved my research, and I would like to thank the children and staff. Finally, I would like to thank my family, my father Susumu, my mother Yoko, my sister Kotomi, and all my relatives for their moral support of my research.

1. Introduction: Formal Linguistics Background

1.1. Incremental Reading

Incremental processing is the essential component of a natural language, which originates as a spoken language; that is, humans speak and listen to a sentence in the temporal order. They consume words in the sentence from the beginning to the end on-the-fly. Then, they could understand the meaning of the sentence even if the sentence is in-progress. After the invention of the writing language, they still consume the words in the sentence from the beginning to the end. According to the human eye-tracking experiment [13], they read the sentence from left to right, which means that humans read it incrementally. For example, humans read Sentence (1) word by word. We could obtain the meaning of the sentence even if the sentence is in-progress.

(1) High above the city, on a tall column, stood a statue of the Happy Prince.[1]

In addition, incremental processing is essential even if we assume that humans can process enough words at once, they use their memory to keep the several words to process the sentence. In generative grammar, the sentence is generated by some grammar rules. Generally, the sentence might have an arbitrary number of words. However, the human brain physically keeps only a finite number of words in the memory. Hence, they should chop the sentence into chunks by adequate size. Then, they should process the chunks incrementally. This is the same as the incremental processing by words. For example, the human chops Sentence (1) into chunks: 'High above the city,' 'on a tall column,' 'stood a statue,' and 'of the Happy Prince.' For every chunk processed by the reader, they could understand the meaning of the sentence incrementally.

In natural language processing, incremental processing is also a significant problem because the target of the software is the sentence written by the human who writes for the human. Then, we expect that the software should be able to understand the meaning of the sentence incrementally. In the case of machine translation, we would like to obtain the sentence on-the-fly. In the case of the question-answering system, we would like to process a large document more than a computer could not keep on disk space. Because of these reasons, we should consider the incremental processing problem in practice.

From the viewpoint of Linguistics, incremental processing is also important. The garden-path sentence is a sentence that humans often fail to understand. The following

[1] Oscar Wilde, The Happy Prince and Other Tales, 1888.

sentences are garden-path sentences in English and Japanese.

(2) The horse raced past the barn fell.

(3) Taro-ha Hanako-no shashin-wo totta Jiro-wo hometa.
(Taro complimented Jiro who takes a picture of Hanako.)

In Sentence (2), the human misunderstands the meaning of the sentence until the last word; that is, the human obtains the meaning: the horse is running in front of the barn In this situation, the word 'raced' is a verb. However, once the human reads the word 'fell,' then the word 'fell' becomes a verb and the word 'raced' becomes an adjective. In Sentence (3), the human also so does until the word 'Jiro-wo hometa', that is, the human obtains the meaning: Taro takes a picture of Hanako. In this situation, the word 'totta' is a verb. However, once the human read the word 'Jiro-wo hometa', then the word 'hometa' is a verb and the word 'totta' becomes an adjective. Further, we obtained additional information: Taro did not takes a picture of Hanako. This phenomenon depends on the incremental processing of the human brain. Therefore, we should consider incremental processing in natural language processing to simulate human recognition.

The current formal grammar theory contains two kinds of incremental processing problems. The first problem is the availability of the grammar rules: it is unclear with what rules we could parse the sentence incrementally. The second problem is the existence of incremental parsing: It is required to verify whether we can parse all the sentences incrementally or not. In this thesis, we aim to solve these problems with categorial grammar and its variants. Moreover, we also aim to develop the incremental parsing algorithm and its implementation in software to demonstrate the proof of concept. Further, we finally showthe formal grammar theory verification system in the proof assistant system, Isabelle/ HOL.

In addition, we show the computational problem of incremental parsing. From the computational model perspective, there are two kinds of incremental processing. By two approaches, we could solve the above incremental processing problem. The first option is the parallel processing shown in Figure 1.1. When the word is consumed, they produce several states by multiple rules in parallel and fork the process, and finally, they obtain a result by one branch in some condition. The second option is the mutable processing shown in Figure 1.2. When the word is consumed, they update the state by the rule and continue the processing. They will retry in some step past if the state is rejected.

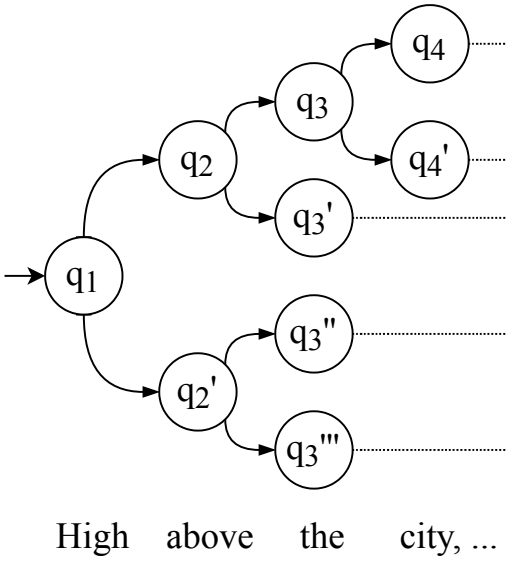


Figure 1.1.: Parallel incremental processing

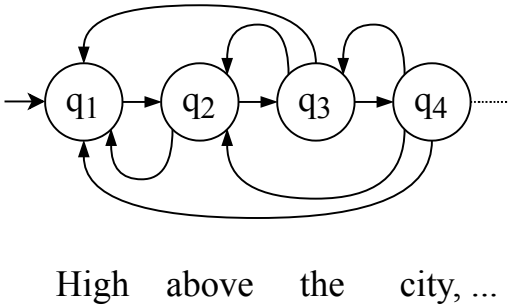


Figure 1.2.: Mutable incremental processing

In the common generative grammar setting, the number of rules and the number of words are finite. Hence, these options are able to simulate each other. To simulate the first option with the second one, they choose only one branch to process and push other branches in the stack memory for each step. If the branch is not closed in the given condition, they pop the stack and return to the previous state. Then, they process the next branch. To simulate the second option by the first one, for each step, they just produce all the possible branches and states by the rule without any retry processes. The following is the research problem for incremental processing.

Incremental Processing Problem

- **Availability of the grammar rules:** It is unclear with what rules we could parse the sentence incrementally.
- **Existence of incremental parsing:** It is required to verify whether we could parse all the sentences incrementally or not.
- **Computational order:** It is unclear in what order we should process the sentence incrementally.

In this thesis, we focus on the first option, which is parallel processing. We do not focus on the efficiency of the parsing algorithm and the memory usage. We show the existence/ availability of incremental processing in formal grammar.

1.2. Heads and Dependencies

For incremental parsing, it is not enough to parse a sentence from left to right in word-by-word order. We should consider keeping the linguistic information of the sentence, which is the heads and the dependencies according to Chomsky's X-bar theory [7].

In this thesis, our grammar formalism is based on phrase structure grammar (PSG), which builds a tree from grammar. We call the tree a constituency tree reflecting an important syntactic structure of natural language. We attach a label to each node by the part of speech (POS) tag, and each leaves by lexical items. We list all the POS tags on Table 1.1, which is majorly used in Penn treebank [28, pp.6–7][5, pp.35–46]. For example, we parse Sentence (4) as Figure 1.3.

(4) I really love you.

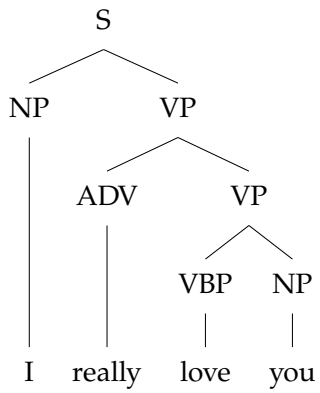


Figure 1.3.: Constituency tree

The tree represents the hierarchical structure of the sentence. The clause level node S presents the whole sentence. The phrase level node VP presents the verb phrases: 'really love you' and 'love you'. The phrase level node NP presents the noun phrases: 'I' and 'you'. The word level node VBP presents the verb: 'love'. The word level node ADV presents the adverb: 'really'. The lowest level node is the actual word: 'I', 'you', 'really', and 'love'.

Each node except the lowest level node contains at least one lower or equal level node. If the node contains two or more child nodes, we rank the child nodes by the head-dependency relation. The head node is the most important in the phrase. The dependency node is the other node in the phrase. There are well-known head-dependency rules such as that VP should be the head, then NP is the dependency in the phrase. In categorial grammar, the head-dependency relation is discussed in the literature [3]. The functional category VP takes the atomic category NP. Then, the head is VP and the dependency is NP.

However, the head-dependency relation sometimes is ambiguous and hidden/ implicit in the grammatical structures. Further, the relation is not consistent in a grammar theory, especially, categorial grammar and its variants. For instance, we expect the functional category is the head in the phrase, but it may not be true because the functional category is given independence from the relation. In the variation of categorial grammar, the functional category also takes a functional category as a composition. Thus, the functional category is not always the head in the phrase. Hence, we should not persist in the head-dependency relation only in the functional category.

This mismatch between the head-dependency relation and the grammatical structure is a problem of the current formal grammar theory. In the grammar extraction technique, we need to extract the head-dependency relation from the constituency tree but it is just the functional category, not the head-dependency relation. By using a dataset extracted with this method, we never deal with the correct head-dependency relation. Moreover, we sometimes make labels of two nodes as heads in the constituency tree by grammar extraction. We call this problem *losing a head* in grammar extraction. It breaks the head-dependency relation by assigning two heads in the phrase. Then, we cannot compare the significance between two nodes in a phrase. The following is the research problem for the head-dependency relation.

Losing-a-head Problem

- **Mismatching relation**: Every function-arguments relation is not the head-dependency relation.
- **Missing relation**: The composition of functional categories is not the head-dependency relation.
- **Recovering relation**: The type-raising rule recovers the head-dependency relation only if the function-argument relation is in categorial grammar.

In this thesis, we aim to solve these problems by categorial grammar and its variants.

The type-raising rule is the key to solving these problems. It is the rule changing the head-dependency relation dynamically in the phrase. However, the type-raising rule is only effective in a case: the application of a functional category and another category. To deal with the composition of two functional categories, we need to introduce the generalized type-raising rule, called the continuation-passing style (CPS) translation rule. In a later section, we show the technical details of the CPS translation rule.

Table 1.1.: POS tags

POS	**Description**[2]
Clause Level	
S	Simple declarative clause, i.e. one that is not introduced by a (possibly empty) subordinating conjunction or a wh-word and that does not exhibit subject-verb inversion.
SBAR	Clause introduced by a (possibly empty) subordinating conjunction.
SBARQ	Direct question introduced by a wh-word or a wh-phrase. Indirect questions and relative clauses should be bracketed as SBAR, not SBARQ.
SINV	Inverted declarative sentence, i.e. one in which the subject follows the tensed verb or modal.
SQ	Inverted yes/no question, or main clause of a wh-question, following the wh-phrase in SBARQ.
Phrase Level	
ADJP	Adjective Phrase.
ADVP	Adverb Phrase.
CONJP	Conjunction Phrase.
FRAG	Fragment.
INTJ	Interjection. Corresponds approximately to the part-of-speech tag UH.
LST	List marker. Includes surrounding punctuation.
NAC	Not a Constituent; used to show the scope of certain prenominal modifiers within an NP.
NP	Noun Phrase.
NX	Used within certain complex NPs to mark the head of the NP. Corresponds very roughly to N-bar level but is used quite differently.
PP	Prepositional Phrase.
PRN	Parenthetical.
PRT	Particle. Category for words that should be tagged RP.
QP	Quantifier Phrase (i.e. complex measure/amount phrase); used within NP.
RRC	Reduced Relative Clause.
UCP	Unlike Coordinated Phrase.
VP	Vereb Phrase.

WHADJP	Wh-adjective Phrase. Adjectival phrase containing a wh-adverb, as in how hot.
WHAVP	Wh-adverb Phrase. Introduces a clause with an NP gap. May be null (containing the 0 complementizers) or lexical, containing a wh-adverb such as how or why.
WHNP	Wh-noun Phrase. Introduces a clause with an NP gap. May be null (containing the 0 complementizers) or lexical, containing some wh-word, *e.g.* who, which book, whose daughter, none of which, or how many leopards.
WHPP	Wh-prepositional Phrase. Prepositional phrase containing a wh-noun phrase (such as of which or by whose authority) that either introduces a PP gap or is contained by a WHNP.
X	Unknown, uncertain, or unbracketable. X is often used for bracketing typos and in bracketing the...the-constructions.
Word level	
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun (prolog version PRP − S)
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense

VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun (prolog version WP − S)
WRB	Wh-adverb

1.3. Continuations in Natural Language

To keep the head-dependency relation, we need to employ the generalized type-raising rule, called continuation-passing style (CPS) transformation. The main concern of the eager building of a tree in categorial grammar is that the tree is built before the whole head-dependency relation is determined. This is not problem in non-incremental parsing because we can know the whole sentence at once. For incremental processing, however, the eager parsing strategy is failed by the garden-path sentence shown in Section 3.1. Thus, we need to employ CPS-transformation to postpone the computation, which is building a tree, to the appropriate time.

Continuation is a fundamental concept in computer science, especially, lambda calculus. This concept represents the flow of control in a program. In a computational context, we always have a previous computation q_{i-1} and a next computation q_{i+1} in a temporal order. We call the next computation the continuation of the current computation as shown in Figure 1.4. Continuations are general concept and implicitly used in many computational models. For example, assume that the current computation is the multiplication 2×3 in computation of $1 + 2 \times 3$, then the next computation is the addition $1 + \square$ and it is the continuation.

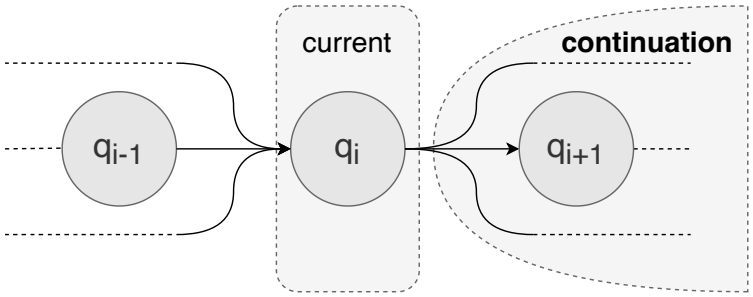


Figure 1.4.: Continuation in the computation

The continuation-passing style (CPS) is a notation for representing programs in which the continuation is explicitly passed as an additional argument. For instance, the CPS

[2]The description of the POS tags is from [28, pp.6–7][5, pp.35–46].

version of $1+2\times3$ is shown as $(\lambda k.k(2\times3))(\lambda x.1+x)$. The continuation $1+\sqcup$ is explicitly passed as an additional argument of 2×3. In this sense, we call the non-CPS form as the direct style (DS). Comparing DS and CPS, the functional form in DS is transformed into the argument in CPS. In fact, $1+\sqcup$ is a functional form in DS and it is transformed into the argument in CPS.

In natural language, continuation is also used implicitly. As the settings, we consider that the sentence is parsed incrementally. The current computation is the parsing of the current word and the next computation is the parsing of the next word. The continuation is the parsing of a sentence with the rest of the words appearing after the current word as shown in Figure 1.5. In the previous section, we assume the head-dependency relation for each phrase. The dependency is the current words and the head-part is the rest of the words. Then, the head-dependency relation is reversed in CPS form by taking the continuation as an argument.

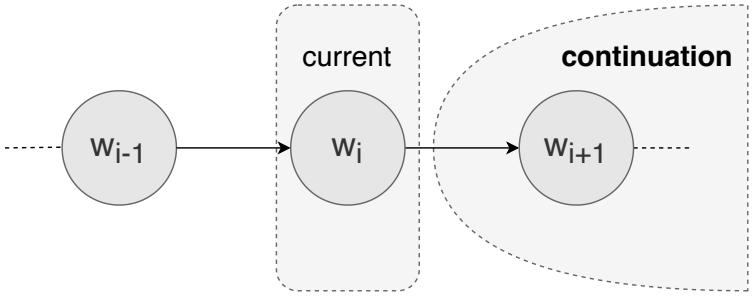


Figure 1.5.: Continuation in the sentence

The transformation from the DS to CPS was initially proposed by Plotkin [26, p.147]. We show Plotkin's transformation in Section 2.3. In the 1990s' such transformation was reinvestigated as the type-raising in linguistics. In linguistics, the alternative transformation was proposed by Barker [1, p.4]. Furthermore, there are variants of transformations. Plotkin's original transformation drastically changes the scope-taking of variables. In addition, the transformation also changes the evaluation strategy from eager evaluation to lazy evaluation in the lambda calculus sense. It is useful for the generalized version of type-raising. However, it is required to take the universe of the scope regarding a given context. It is a strict condition for linguistics usage. Hence, Barker proposed restricted transformation. Barker's transformation is rather simple than the original one but it is potentially hard to deal with the deep scope.

In this thesis, we inspect the proof-theoretic properties of CPS-transformation defined in simply-typed lambda calculus, of which type system is formalized in the propositional logic. Moreover, we extend the propositional logic with the directional implications $/, \backslash$. Furthermore, the extension is used to represent the categorial grammar and variants including Lambek Calculus. Barker's transformation is provable in Lambek calculus but the Plotkin's transformation is not. They are both useful but they have the following issues.

CPS Transformation Problem

- **Provability:** Plotkin's transformation is unprovable in an intuitionistic system.
- **Expressiveness:** Barker's transformation deals without the deep scope.

To solve these problems, we give the successor of Plotkin's transformation to be the moderate between them. Our proposal is restricted to CPS transformation but it can deal with the deep scope such as λ-abstraction in lambda calculus as shown in Table 1.2. The detail of the transformation is shown in Section 2.3. To consider the proof theory, we employ the theorem-proving technique to test the provability and to develop the automatic deducing algorithm.

The type system of simply-typed lambda calculus is formalized in propositional logic. We consider the extension of the propositional logic, Lambek Calculus (LC). In the generative grammar theory, LC is corresponding to context-free grammar (CFG). The following comparison table shows the expressiveness of the transformations in lambda calculus, their provability in LC, and their generative power in the generative grammar theory.

Table 1.2.: Comparison of CPS transformations

	Provability in LC	Generative power	Expressiveness
Plotkin [26]	Unprovable	Over CFG	Any term
Barker [1]	Provable	CFG	non-λ abstraction
Tanguchi [35]	Provable	CFG	Any term [3]

1.4. Theorem proving for Linguistics

Our approach is to connect proving a theorem to parsing a sentence. We first formalize parsing a sentence into a tree and grammar rules. The generation of a tree with given grammar rules is called parsing. The grammar rules can be represented as a set of implications. For example, the verb phrase is a noun phrase followed by a verb. It means that the verb is 'the noun phrase implies the verb phrase'. Further, the generation of a sentence is achieved by applying an implication with modus ponens and reaching a sentence clause S. Thus, we can represent the tree as a proof in propositional logic concluding S.

Here, we employ the theorem-proving method to search for proof, which is a tree generated from a set of grammar rules corresponding to propositional logic. Isabelle/HOL is a proof assistant system based on higher-order logic (HOL). To search for proof in Isabelle/HOL, its developer employs a term rewriting system (RTS), which rewrites a goal into a subgoal, and checks whether the subgoal is a given assumption or not.

[3] It excepts the higher-order lambda abstraction

There are many other theorem provers, such as Coq, Lean, and Agda. They are built on the different aspects of proof search, which is called Curry-Howard correspondence.

The goal is to write in Isar, which is a programming language to write proofs in Isabelle/HOL. Isar is mainly constructed by four parts: data types, functions, theorems, and proofs. We first define a data type to represent the target of the problem. `datatype` is a keyword to define a new algebraic data type, which is a constructor of data as follows:

Listing 1.1: Boolean

```
1 datatype Boolean = T | F
```

Next, we define a function to handle the data type. `fun` is a keyword to define a new function as follows:

Listing 1.2: Negation

```
1 fun bNot :: "Boolean ⇒ Boolean" where
2   "bNot T = F"
3 | "bNot F = T"
```

The function `bNot` takes a Boolean value and returns the opposite value. Further, we define a theorem to prove a property of the function. `theorem` is a keyword to define a new theorem as follows:

Listing 1.3: Double negation

```
1 theorem bNot_bNot: "bNot (bNot b) = b"
```

The theorem `bNot_bNot` states that the double negation of a Boolean value is the same as the original value. Finally, we prove the theorem by proving the subgoals. In this part, we declare rewriting commands. In this example, we use `by simp` command to rewrite the goal into a subgoal. `simp` is a built-in rewriting strategy defined in Isabelle/HOL as rewriting it based on equality `=`. The common rewriting strategies are `simp`, `auto`, `blast`, and `fastforce`, in which the right strategy is more aggressive than the left one. Note that the aggressive rewriting strategy may cause an infinite loop. Hence, we need to be careful to use the rewriting strategy, that is, it is not enough to use the aggressive strategy.

Isabelle/ HOL also provides the automatic theorem prover (ATP) to search for proof. `sledgehammer` is a command to search for proof by using the ATP. As the default settings, it uses some ATPs: E, Z3, Vampire, SPASS, CVC4, and zipperposition. The command launches the ATPs in parallel and returns the proof if it finds proof. However, it is not guaranteed to find proof. The search space depends on the computational power

of the machine: a number of cores, memory, and so on. Commonly, ATPs return the proof with a special rewriting strategy called `metis`, which is a built-in rewriting strategy with given hints and theorems. It was investigated by Hurd [16] mentioned in Isabelle/HOL source code [4].

We use the command `sledgehammer` in two situations. First, we use it to search for proof for a given theorem. To prove the grammatical properties as theorems, we use the command to search for proof. Note that it does not appear in the proof script because it is replaced by the proof found by the ATP. In this thesis, we show the proof without `sledgehammer` command because all these commands are replaced by the proof found by the ATP. Second, we use it to search for a tree that is a parsing result of a sentence encoded as a theorem in Isabelle/ HOL. Once we define enough grammar rules and lemmas, we can search for a tree by using the command. Thus, we employ ATPs for the formal language parsing problem. Here, we state that proof as a parsing tree, and a prover is a parser.

In this thesis, we contribute to the formal language parsing problem by using the theorem prover. We prove the grammatical properties in Isabelle/ HOL. For instance, it is not easy to check the generative power of complex grammar rules in the manual. The theorem prover checks the generative power of grammar rules automatically. Thus, it is verified to the correctness of existing Linguistic work. Next, we use the theorem prover to search for a parsing tree. This is a new approach to the formal language parsing problem, which is positioned as a successor of the definite clause grammar (DCG) [25]. In DCG, an interpreter of Logic programming parses a sentence by given grammar. In our work, the ATP parses a sentence by giving grammar rules and lemmas.

Theorem Proving Problem

- **Decidability:** Theorem prover as a parser is not decidable in general.
- **Complexity:** It is complex that we inspect the state-of-art grammar manually.

Compared to the classical work, our work can be expected semi-decidability regarding the proof search strategy. The unification algorithm in Prolog is commonly a depth-first search. Thus, it may drop into the infinite loop in some cases. The ATP is expected to run as a breadth-first search in simple cases. Then, it must halt when the algorithm finds the proof/ parsing tree. Using ATP for the formal parsing problem contributes to the decidability problem. For practical reasons such as the memory size of computer resources, note that it may fail to prove/ parse them. In Appendix B, we define the propositional logic and soundness in Isabelle/HOL and show the completeness of the foundation of the categorial grammar.

[4] Makarius Wenzel (Last updated at: 2022/10/26), File <~~/src/Tools/Metis/metis.ML>, `https://isabelle.in.tum.de/library/HOL/HOL/ISABELLE_HOME/src/Tools/Metis/metis.ML.html`, checked at 2022/11/17

2. Preliminaries: Logic, Language, and Computation

2.1. Compbinatory Categorial Grammar

The state-of-the-art techniques on natural language processing (NLP) rely on a corpus that is a collection of sentences, each word of which is annotated by a part-of-speech (POS). Since these POSs are generalized to categories of combinatory categorial grammar (CCG) [32], we aim at fixing the categories. Since the candidates of categories increase in proportion to the length of the sentence, finding a proper set of grammar rules is computationally explosive and an exhaustive search is in demand.

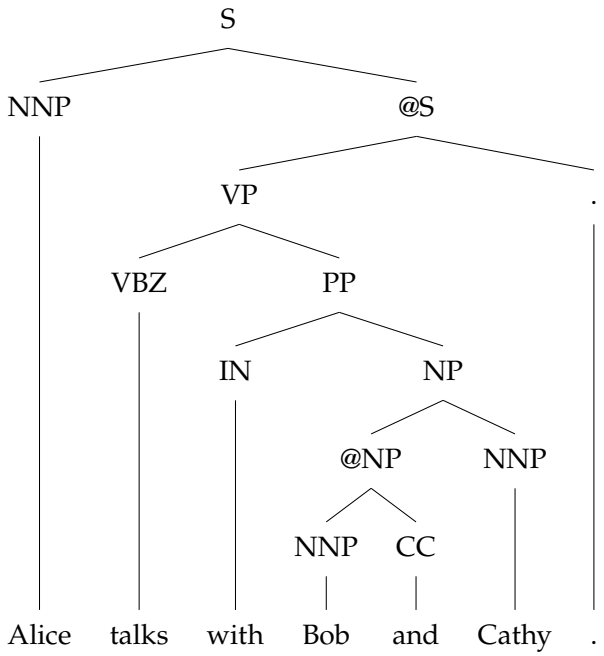


Figure 2.1.: Constituency Tree

Our target formalism, CCG, is a kind of phrase structure grammar as opposed to dependency grammar.

Definition 1 (Category)**.** Category is syntactic information of the word and is repre-

sented by the combination of POS and /, \. The following is the Backus-Naur form (BNF) of CCG category for τ.

$$\tau ::= \sigma \mid \tau/\tau \mid \tau\backslash\tau \qquad\qquad \sigma ::= \mathrm{S} \mid \mathrm{NNP} \mid \cdots \quad \text{(POS)}$$

Definition 2 (Syntactic term and semantic term)**.** CCG is the formalism to parse the sentence syntactically and it simonteously generates the semantic information; the syntactic term is the combination of categories and the semantic term is the lambda expression and the reduction of it. In this grammar, it is denoted as the $\mathtt{syntactic\_term} : \mathtt{semantic\_term}$. Optionally, we drop the semantic term and focus on the syntactic term.

The reduction of semantic part is based on the beta-reduction of lambda calculus. The whole reduction is defined in Theorem 26.

The CCG parser generates a constituency tree from an input sentence, given a set of grammar rules; *e.g.*, the constituency tree in Figure 2.1 is generated from Sentence (5) with the set of grammar rules in Figure 2.2.

(5) Alice talks with Bob and Cathy.

$$\begin{aligned}
\text{Alice} &\rightarrow \mathrm{NNP} : \mathsf{Alice} \\
\text{talk} &\rightarrow \mathrm{VP/PP} : \mathsf{talk} \\
\text{Cathy} &\rightarrow \mathrm{NNP} : \mathsf{Cathy} \\
\text{with} &\rightarrow \mathrm{PP/NP} : \mathsf{with} \\
\text{.} &\rightarrow \mathrm{VP\backslash(NNP\backslash S)} : \lambda f.\lambda x.fx \\
\text{Bob} &\rightarrow \mathrm{NNP} : \mathsf{Bob} \\
\text{and} &\rightarrow \mathrm{NNP\backslash(NP/NNP)} : \mathsf{and}
\end{aligned}$$

Figure 2.2.: Lexicon of English Fragment

Compared to other parsers of phrase structure grammar, the CCG parser is advantageous in that we can directly connect a category to a POS and we can simultaneously obtain its semantics representation, such as in Figure 2.3. Hence, CCG was employed in the semantic parser to improve parsing accuracy and to process the logical inference [21, 17]. However, there often exist multiple assignments of category for a lexical item due to the ambiguity of the head-dependency relation for each branching in a tree. Thus, the candidates of categories regarding an entire sentence increase exponentially in accordance with the length of the sentence. For instance, there can be 265 sets of probable grammar rules over Sentence (5) and the tree in Figure 2.1. It is hard for us to manually fix a category in CCG.

CCG is a successor of categorial grammar (CG) [31], which is equivalent to context-free grammar (CFG). The elements of CG are categories that are atomic categories and functional categories. We commonly use a POS as an atomic category, and a functional type is denoted with two category constructors, / and \, which take the next (previous,

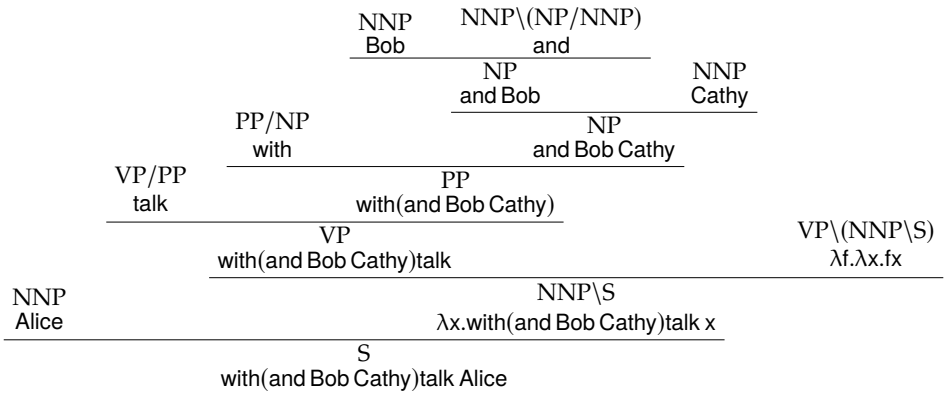


Figure 2.3.: Derivation of grammar

respectively) category and return another category, *e.g.*, a verb phrase S/NP takes a noun phrase NP and returns a sentence S.

In this formalism, we call an association from a lexical item called *lexeme* to a category, a grammar rule, and call a set of grammar rules a lexicon[1]. Further, we correspond a parsing tree to a derivation chaining applications of functional categories, where the chain is restricted to taking a next or a previous category only, *i.e.*, the word order is fixed by the occurrence of a word in a sentence. Furthermore, we attach a semantic representation to the functional category. The representation is similar to lambda calculus, as shown in the following transformation for the bottom derivation in Figure 2.3.

$$\begin{aligned}&\mathsf{NNP} : \mathsf{Alice} \quad \mathsf{VP} : \mathsf{with(...)talk} \quad \mathsf{VP\backslash(NNP\backslash S)} : \lambda f.\lambda x.fx\\ &\Rightarrow \mathsf{NNP} : \mathsf{Alice} \quad \mathsf{NNP\backslash S} : \lambda x.\mathsf{with(...)talk}\, x\\ &\Rightarrow \mathsf{S} : \mathsf{with(...)talk\ Alice}\end{aligned}$$

We explain processing a given sentence in CCG. Through the processing, we retrieve a derivation from categories to a category as proof of natural deduction in logic. The lexicon ℓ is a map from a word to a category in Figure 2.2. We process Sentence (5), First, we pick up a grammar rule from the lexicon for each lexeme, *e.g.*, $\ell(\mathsf{Alice}) = \mathsf{NNP} : \mathit{Alice}$. Then, we get a sequence of grammar rules holding the order by the original sentence. Next, we construct a derivation from the sequence. The manner of derivation is almost the same as a proof of natural deduction using only an implication instead of the two functional categories. Corresponding to the modus ponens (MP) in logic, we employ the following application rules < and >.

$$\frac{\alpha/\beta : f \qquad \beta : y}{\alpha : fy} > \qquad \frac{\beta : y \qquad \beta\backslash\alpha : f}{\alpha : fy} <$$

As for two lexemes, "Bob and" with the application rule <, we obtain the following derivation.

[1]lexicon means a set of lexeme in linguistics.

$$\frac{\text{NNP} : \text{Bob} \qquad \text{NNP}\backslash(\text{NP}/\text{NNP}) : \text{and}}{\text{NP}/\text{NNP} : \text{and Bob}} <$$

Further, we combine two leximata of the same category with conjunction and the deduction rule CONJ, then we combine the two noun phrases "Bob and Cathy" as follows.

$$\frac{\text{NNP} : \text{Bob} \qquad \text{CONJ} : \text{and} \qquad \text{NNP} : \text{Cahty}}{\text{NP} : \text{and Bob Cahty}} \text{CONJ}$$

In addition to CG, Steedman [32] introduced combinators as deduction rules in CCG. The major combinator B combines two functional categories as follows.

$$\frac{\alpha/\beta : f \qquad \beta/\gamma : g}{\alpha/\gamma : \lambda z.f(g(z))} > \text{B} \qquad \frac{\alpha\backslash\beta : f \qquad \beta\backslash\gamma : g}{\alpha\backslash\gamma : \lambda x.g(f(x))} < \text{B}$$

With this combinator rule $>$ B, we show the derivation of "talk with" as follows.

$$\frac{\text{VP}/\text{PP} : \text{talk} \qquad \text{PP}/\text{NP} : \lambda y.\lambda x.\text{with}(x)y}{\text{VP}/\text{NP} : \lambda x.\text{with}(x)\text{talk}} < \text{B}$$

Next, we get the derivation of the verb phrase, which consists of two parts, "talks with" and "Bob and Cathy." using the application rule $<$.

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \text{VP}/\text{NP} : \lambda x.\text{with}(x)\text{talk} & \text{NP} : \text{and Bob Cathy}\end{array}}{\text{VP} : \text{with(and Bob Cathy)talk}} <$$

After the application of rule $<$, we obtain the sentence from three parts "Alice", "talk with Bob and Cathy", and "." However, the derivation is not only this one. In fact, this derivation is different from Figure 2.3 for the verb phrase. Moreover, we can find another derivation with other combinator rules such as the type-raising rules as follows.

$$\frac{\alpha : x}{(\beta/\alpha)\backslash\beta : \lambda f.fx} > \text{T} \qquad \frac{\alpha : x}{\beta/(\alpha\backslash\beta) : \lambda f.fx} < \text{T}$$

2.2. Lambek Calculus

In this thesis, we write α/β and $\beta\backslash\alpha$ as the implication from β to α to deal with the propositional logic system. This is the mathematical formulation of the sentence structure introduced by Lambek [18]. The original Lambek Calculus is defined as a sequent calculus with tree connectives, the implications $/$, $\backslash$, and the product $\bullet$. However, in this thesis, we do not employ the product in its definition for the sake of simplicity.

Definition 3 (Lambek Calculus (LC) [18])**.** Lambek calculus is defined as a sequent calculus, which consists of atomic terms, right functional terms $\cdot\backslash\cdot$, and left functional terms $\cdot/\cdot$. The following are initial sequent and deduction rules. A lower Greek letter is a term for a sequent, and a capital Greek letter is a sequence of terms.

$$\frac{}{\alpha \vdash \alpha}\,\mathrm{Id} \qquad \frac{\Sigma \vdash \alpha \qquad \Gamma, \alpha, \Delta \vdash \beta}{\Gamma, \Sigma, \Delta \vdash \beta}\,\mathrm{Cut} \qquad \frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \beta/\alpha}\,\mathrm{I}/$$

$$\frac{\alpha, \Gamma \vdash \beta}{\Gamma \vdash \alpha\backslash\beta}\,\mathrm{I}\backslash \qquad \frac{\Sigma \vdash \alpha \qquad \Gamma, \beta, \Delta \vdash \gamma}{\Gamma, \Sigma, \alpha\backslash\beta, \Delta \vdash \gamma}\,\backslash\mathrm{I} \qquad \frac{\Gamma, \beta, \Delta \vdash \gamma \qquad \Sigma \vdash \alpha}{\Gamma, \beta/\alpha, \Sigma, \Delta \vdash \gamma}\,/\mathrm{I}$$

LC is the mathematical formalization of categorial grammar (CG). Hence, some of the CCG rules are provable in this system. For instance, the application is provable.

$$\frac{\alpha \vdash \alpha \qquad \beta \vdash \beta}{\alpha, \alpha\backslash\beta \vdash \beta}\,\backslash\mathrm{I}$$

Furthermore, the simple sentence structure is also provable in LC. In the following proof, we show that I love you is a sentence because NP, NP\(S/NP), NP is reduced into S.

$$\frac{\mathrm{NP} \vdash \mathrm{NP} \qquad \dfrac{\mathrm{S} \vdash \mathrm{S} \qquad \mathrm{NP} \vdash \mathrm{NP}}{\mathrm{S/NP}, \mathrm{NP} \vdash \mathrm{S}}\,/\mathrm{I}}{\mathrm{NP}, \mathrm{NP}\backslash(\mathrm{S/NP}), \mathrm{NP} \vdash \mathrm{S}}\,\backslash\mathrm{I}$$

In Definition 3, we have included Cut in the basic rules, however, it is known that we can exclude Cut from the basic rules.

Theorem 4 (Cut elimination in LC [18])**.** *If* $\Sigma \vdash \alpha$ *and* $\Gamma, \alpha, \Delta \vdash \beta$ *are both provable without* Cut*, then* $\Gamma, \Sigma, \Delta \vdash \beta$ *is also provable.*

Proof. This proof is based on the proof of the cut elimination in the original Lambek calculus [18]. Let $d(\alpha)$ be a degree which counts the number of connectives / and \ in a given formula α. Generally, $d^*(\Gamma)$ is a sum of $d(\alpha)$ for each α in Γ. For example, the degree of Cut in Definition 3 is $d^*(\Sigma, \Gamma, \Delta, \alpha, \beta)$. Let the last step of a given proof is derived by the rule Cut as follows.

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash \alpha & \Sigma, \alpha, \Delta \vdash \beta\end{array}}{\Sigma, \Gamma, \Delta \vdash \beta}\,\mathrm{Cut}$$

Then, we show the proof of $\Sigma, \Gamma, \Delta \vdash \beta$ from premises proved by the rules with/ without Cut, whose degrees are less than $d^*(\Sigma, \Gamma, \Delta, \alpha, \beta)$. We apply this method inductively from the root to the leaves to prove $\Sigma, \Gamma, \Delta \vdash \beta$ without Cut.
case I: $\Gamma \vdash \alpha$ is an instance of $\alpha \vdash \alpha$. Then, the proof of $\Sigma, \alpha, \Delta \vdash \beta$ is given by the other premises.
case II: $\Sigma, \alpha, \Delta \vdash \beta$ is an instance of $\alpha \vdash \alpha$. Then, Σ and Δ is empty. Thus, the proof of $\Gamma \vdash \beta$ is given by the other premises.
case III: The last step in the proof of $\Gamma \vdash \alpha$ is derived by one of the rules /I and \I. Then, the premises of the proof of $\Gamma \vdash \alpha$ are $\Gamma'' \vdash \gamma$ and $\Gamma' \vdash \alpha$ where $d^*(\Gamma') < d^*(\Gamma)$. Here, we have a proof of $\Sigma, \Gamma', \Delta \vdash \beta$ with Cut as follows. The degree of this proof is $d^*(\Sigma, \Gamma', \Delta, \alpha, \beta)$ which is less than $d^*(\Sigma, \Gamma, \Delta, \alpha, \beta)$. Further we have a proof $\Sigma, \Gamma, \Delta \vdash \beta$ with the above condition.

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ \Gamma' \vdash \alpha & \Sigma, \alpha, \Delta \vdash \beta\end{array}}{\Sigma, \Gamma', \Delta \vdash \beta}\,\mathrm{Cut} \qquad \begin{array}{c}\vdots\\ \Gamma'' \vdash \gamma\end{array}}{\Sigma, \Gamma, \Delta \vdash \beta}\,/\mathrm{I} \text{ or } \backslash\mathrm{I}$$

case IV: The last step in the proof of $\Sigma, \alpha, \Delta \vdash \beta$ is defined by one of the rules $\mathrm{I}/$, $\mathrm{I}\backslash$, $/\mathrm{I}$, and $\backslash\mathrm{I}$ where the main connective of α is not introduced. Then, the premises of the proof of $\Sigma, \alpha, \Delta \vdash \beta$ are $\Sigma', \alpha, \Delta' \vdash \beta'$ and $\Omega \vdash \gamma'$ where $d^*(\Sigma', \beta', \Delta') < d^*(\Sigma, \beta, \Delta)$. Here, we have a proof of $\Sigma', \Gamma, \Delta' \vdash \beta'$ with Cut as follows. The degree of this proof is $d^*(\Sigma', \Gamma, \Delta', \alpha, \beta')$ which is less than $d^*(\Sigma, \Gamma, \Delta, \alpha, \beta)$. Further we have a proof $\Sigma, \Gamma, \Delta \vdash \beta$ with the above condition.

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ \Gamma \vdash \alpha & \Sigma', \alpha, \Delta' \vdash \beta'\end{array}}{\Sigma', \Gamma, \Delta' \vdash \beta'}\,\mathrm{Cut} \qquad \begin{array}{c}\vdots\\ \Omega \vdash \gamma'\end{array}}{\Sigma, \Gamma, \Delta \vdash \beta}\,\mathrm{I}/, \mathrm{I}\backslash, /\mathrm{I}, \text{ or } \backslash\mathrm{I}$$

case V: The last step in the proofs of both premises introduce the main connective of $\alpha = \gamma/\delta$. Then the proof is as follows. The degree of this proof is $d^*(\Gamma, \gamma/\delta, \Sigma, \Delta', \Delta'', \beta)$.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, \delta \vdash \gamma\end{array}}{\Gamma \vdash \gamma/\delta}\,\mathrm{I}/ \qquad \dfrac{\begin{array}{cc}\vdots & \vdots\\ \Delta' \vdash \delta & \Sigma, \gamma, \Delta'' \vdash \beta\end{array}}{\Sigma, \gamma/\delta, \Delta', \Delta'' \vdash \beta}\,/\mathrm{I}}{\Sigma, \Gamma, \Delta', \Delta'' \vdash \beta}\,\mathrm{Cut}$$

The above proof is rewritten as follows. The degree of the Cut in premises is $d^*(\Gamma, \delta, \gamma, \Sigma, , \Delta'', \beta)$, which is less than $d^*(\Gamma, \gamma/\delta, \Sigma, \Delta', \Delta'', \beta)$. Further, the degreee of the Cut in the last step is $d^*(\Delta', \delta, \Sigma, \Gamma, \Delta'', \beta)$, which is also less than $d^*(\Gamma, \gamma/\delta, \Sigma, \Delta', \Delta'', \beta)$. Thus, we have a proof $\Sigma, \Gamma, \Delta', \Delta'' \vdash \beta$ with the above condition.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Delta' \vdash \delta\end{array} \qquad \dfrac{\begin{array}{cc}\vdots & \vdots\\ \Gamma, \delta \vdash \gamma & \Sigma, \gamma, \Delta'' \vdash \beta\end{array}}{\Sigma, \Gamma, \delta, \Delta'' \vdash \beta}\,\mathrm{Cut}}{\Sigma, \Gamma, \Delta', \Delta'' \vdash \beta}\,\mathrm{Cut}$$

case VI: The last step in the proofs of both premises introduce the main connective of $\alpha = \gamma\backslash\delta$. The degree of this proof is $d^*(\Gamma, \gamma\backslash\delta, \Sigma, \Delta', \Delta'', \beta)$. This is treated similarly to case V. □

Hereafter, we omit Cut from LC for the sake of brevity. Since the Cut rule produces a category that does not appear in the conclusion, the cut-free proof is important to show that a given sequent is unprovable in LC. For instance, we show two lemmas in both cases that are provable and unprovable in LC.

Lemma 5 (Provability of type-raising in LC)**.** NP $\Rightarrow$ S/(NP\S) *is a type-raising rule, which switches the biting relation from* NP NP\S *to* S/(NP\S) NP\S*. Then, the sequent* $\alpha \vdash \beta/(\alpha\backslash\beta)$ *and* $\alpha \vdash (\beta/\alpha)\backslash\beta$ *are provable in LC.*

Proof. The proof of these sequents is given as follows.

$$\dfrac{\dfrac{\alpha \vdash \alpha \qquad \beta \vdash \beta}{\alpha, \alpha\backslash\beta \vdash \beta}\backslash\mathrm{I}}{\alpha \vdash \beta/(\alpha\backslash\beta)}\mathrm{I}/ \qquad \dfrac{\dfrac{\alpha \vdash \alpha \qquad \beta \vdash \beta}{\alpha/\beta, \beta \vdash \beta}/\mathrm{I}}{\alpha \vdash (\beta/\alpha)\backslash\beta}\mathrm{I}\backslash$$

□

2.3. CPS-Transformation in Lambda Calculus

Based on the simply typed lambda calculus, we define the concept of computation in lambda calculus. This is called *continuation*, which was initially investigated in the 1960s to represent what calculation we should run next in a given context [27], which is similar to the relationship `goto` – `label` in the imperative programming language. We use the `goto` command to jump into the next program execution declared by the `label` command; that is, we deal with the `goto` command to continue to the next execution. We call this operation the continuation. The continuations appear everywhere in the calculations because the program is the chain of the execution command and there implicitly exists the continuations among each execution.

We, hereafter, use the lambda calculus notation because the infix notation is not usable to denote the CPS form. Note that '$\mathsf{add}\,x\,y$' is $x + y$. Example 6 shows all continuations in an arithmetic program.

Example 6 (Continuations in 1 + 2)**.** The call-by-value interpreter of the arithmetic program evaluates '$\mathsf{add}\,1\,2$' as follows. The continuation of Step 1 is Step 2–5. Generally, the continuation of Step i is Step $(i + 1)$–5.

1. Evaluate the integer 2
2. Evaluate the integer 1.
3. Evaluate the operator add.

$$\underbrace{\mathsf{add}}_{3}\ \underbrace{1}_{2}\ \underbrace{2}_{1}$$

First, we work with untyped and simply-typed lambda calculus, as defined in [15]. A usual computer program implicitly executes a code sequentially, step by step. However, in some cases, the order of execution may change. The remained schedule after the preempted execution is called a *continuation*. Then, a computer program with an explicitly-mentioned continuation is said to be in continuation-passing style (CPS).

Such as the arithmetic program, the form in which the continuation implicitly appears is called the *direct style* (DS) [9]. On the other hand, we introduce the form in which the

continuation explicitly appears called the *continuation-passing style* (CPS) [27, 26]. The following is the CPS form of 1 + 2, where add′ is also the CPS form of add.

$$\lambda k.(\lambda l.(\lambda s.s\ \mathsf{add}')(\lambda h.(\lambda t.t1)(\lambda i.hil)))(\lambda m.(\lambda u.u2)(\lambda n.mnk))$$

1. $\lambda n.mnk$ is the continuation of 2.
2. $\lambda i.hil$ is the continuation of 1.
3. $\lambda h.(\lambda t.t1)(\lambda i.hil)$ is the continuation of add′.

The argument k represents the global continuation, resolved after all the reductions are done. Note that the call-by-value interpreter runs the CPS form in the DS form's reverse order (3–2–1) because each term is deferred by the lambda abstraction. This inversion is introduced by Plotkin as follows.

Definition 7 (Plotkin's CPS transformation [26])**.** $[\![\cdot]\!]$ is recursively defined as a map from a DS lambda term to a CPS lambda term, where x is a variable, and M and N are meta-variables. Further, m, n, k represent the continuations of each term.

$$[\![x]\!] \equiv \lambda k.kx \qquad [\![\lambda x.M]\!] \equiv \lambda k.k(\lambda x.[\![M]\!]) \qquad [\![MN]\!] \equiv \lambda k.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnk)) \tag{2.1}$$

The original CPS transformation in Definition 7 is defined in untyped-lambda calculus. Here, we consider the type of transformation in simply-typed lambda calculus as follows.

Definition 8 (Type of Plotkin's CPS transformation [4])**.** $\langle\!\langle\cdot\rangle\!\rangle$ is mutually defined with $\langle\cdot\rangle$, where capital letters are types of simply-typed lambda calculus. Further, A is the answer type uniquely determined in the global context. $\langle\!\langle\cdot\rangle\!\rangle$ is corresponding to $[\![\cdot]\!]$ in Definition 7.

$$\langle\!\langle\alpha\rangle\!\rangle_\delta = (\langle\alpha\rangle_\delta \to \delta) \to \delta \qquad \langle\alpha \to \beta\rangle_\delta = \langle\alpha\rangle_\delta \to \langle\!\langle\beta\rangle\!\rangle_\delta \qquad \langle\gamma\rangle_\delta = \gamma\ (\gamma \text{ is atomic})$$

In combinatory categorial grammar (CCG), we say that a grammar rule is CPS transformation if the category in the rule corresponds to Definition 8. In addition to Plotkin's work, the following is another transformation motivated by linguistic observation.

Definition 9 (Barker's CPS transformation [2])**.** $[\![\![\cdot]\!]\!]$ is recursively defined as a map from a DS lambda term to a CPS lambda term, where a is an arbitrary constant, and M and N are meta-variables. Further, m, n, k represent the continuations of each term.

$$[\![\![a]\!]\!] \equiv \lambda k.ka \qquad [\![\![MN]\!]\!] \equiv \lambda k.[\![\![M]\!]\!](\lambda m.[\![\![N]\!]\!](\lambda n.kmn)) \tag{2.2}$$

In Definition 9, the lambda abstraction from the CPS transformation is omitted, and also the continuation of N is rearranged from $\lambda n.mnk$ to $\lambda n.kmn$. As a result, types in the transformation are simplified as follows.

Definition 10 (Type of Barker's CPS transformation [2])**.** $\langle\!\langle\!\langle\cdot\rangle\!\rangle\!\rangle$ is defined as follows, where capital letters are types of simply-typed lambda calculus. Further, A is the answer type uniquely determined in the global context. $\langle\!\langle\!\langle\cdot\rangle\!\rangle\!\rangle$ is corresponding to $[\![\![\cdot]\!]\!]$ in Definition 9.

$$\langle\!\langle\!\langle\alpha\rangle\!\rangle\!\rangle_\delta = (\alpha \to \delta) \to \delta$$

Barker's CPS transformation type is the same as the type-raising rule. Thus, the transformation is the generalized version of CCG.

3. Incremental Parsing in Categorial Grammar

3.1. Left-branching Derivation in CG

In categorial grammar (CG) and its variants, there are incremental parsing algorithms [11, 14, 30], but not all grammatical sentences could be parsed by these algorithms. It is empirically known that all the grammatical sentences are parsed incrementally by CG with combinator rules. As the previous researches were software simulations of incremental parsing, we could not obtain that those algorithms work on a sentence of any length. Thus far, we have empirically shown that any grammatical sentences could be incrementally parsed [37], but no proof of this is known to us.

A naïve structure generated by incremental parsing is a left-branching tree, which has binary nodes only on the left branches. We obtain the incremental parsing by our transformation from a given arbitrary tree to the left-branching tree.

First, we define CG and its variants. CG is a proof system A consists of three axiom schemata with a cut rule, where the lower letters are arbitrary in any category of CG and the capital Greek letters are sequences of categories.

$$\frac{}{\alpha \vdash \alpha}\ \textsc{Id} \qquad \frac{}{\alpha/\beta, \beta \vdash \alpha}\ \mathsf{A}> \qquad \frac{}{\alpha, \alpha\backslash\beta \vdash \beta}\ \mathsf{A}< \qquad \frac{\Gamma \vdash \alpha \quad \Sigma, \alpha, \Delta \vdash \beta}{\Sigma, \Gamma, \Delta \vdash \beta}\ \textsc{Cut}$$

For example, (I) "I love you" is parsed as $\mathrm{NP}, (\mathrm{NP}\backslash\mathrm{S})/\mathrm{NP}, \mathrm{NP} \vdash \mathrm{S}$ with the following derivation.

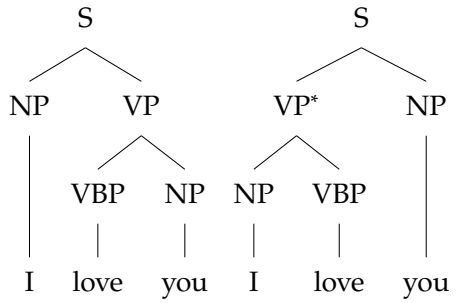


Figure 3.1.: Non-incremental parsing (left) and Incremental parsing (right)

$$\dfrac{\dfrac{}{(\mathrm{NP}\backslash\mathrm{S})/\mathrm{NP}, \mathrm{NP} \vdash \mathrm{NP}\backslash\mathrm{S}}\,\mathsf{A}> \quad \dfrac{}{\mathrm{NP}, \mathrm{NP}\backslash\mathrm{S} \vdash \mathrm{S}}\,\mathsf{A}<}{\mathrm{NP}, (\mathrm{NP}\backslash\mathrm{S})/\mathrm{NP}, \mathrm{NP} \vdash \mathrm{S}}\,\textsc{Cut}$$

According to the proof net's convention, we use a graph. The axiom schemata A>, A< are represented by the graphs in Figure 3.2, and the cut rule is denoted as the connection of the two graphs. Then, the graph of the above example is represented as the graph in Figure 3.3. For the sake of convenience, we employ a new notation of graphs; the graph in Figure 2 is $\mathsf{A}<[\mathrm{NP}, \mathrm{NP}\backslash\mathrm{S}]_{\mathrm{s}}$ and $\mathsf{A}>[(\mathrm{NP}\backslash\mathrm{S})/\mathrm{NP}, \mathrm{NP}]_{\mathrm{NP}\backslash\mathrm{S}}$. We use the binary operator $\odot$ as an adjoining tree by Cut. Then, the graph in Figure 3.3 is the following expression.

$$\mathsf{A}>[(\mathrm{NP}\backslash\mathrm{S})/\mathrm{NP}, \mathrm{NP}]_{\mathrm{NP}\backslash\mathrm{S}} \odot \mathsf{A}<[\mathrm{NP}, \mathrm{NP}\backslash\mathrm{S}]_{\mathrm{s}} = \mathsf{A}>[\mathrm{NP}, \mathsf{A}<[(\mathrm{NP}\backslash\mathrm{S})/\mathrm{NP}, \mathrm{NP}]_{\mathrm{NP}\backslash\mathrm{S}}]_{\mathrm{s}}$$

Generally, only the following eight patterns are possible for each triplet of nodes, where capital Roman letters are expressions with the root category denoted as the subscript. We especially call the patterns (7) and (8) the *backward long reference*. In these patterns, the third category includes a hidden category for the first category, *e.g.*, The category z in the pattern (7) appears in the left-most/ right-most nodes as a part of functional categories. For example, post-posited adverbs 'only', in linguistics, is corresponding to this reference, which are grammatical rules but are not preferable in the conversation.

(1) $\mathsf{A}>[\mathsf{A}>[\mathrm{X}_{(\alpha/\gamma)/\beta}, \mathrm{Y}_{\beta}], \mathrm{Z}_{\gamma}]_{\alpha}$
(2) $\mathsf{A}>[\mathsf{A}<[\mathrm{X}_{\alpha}, \mathrm{Y}_{\alpha\backslash(\beta/\gamma)}], \mathrm{Z}_{\gamma}]_{\beta}$
(3) $\mathsf{A}<[\mathsf{A}>[\mathrm{X}_{\alpha/\beta}, \mathrm{Y}_{\beta}], \mathrm{Z}_{\alpha\backslash\gamma}]_{\gamma}$
(4) $\mathsf{A}<[\mathsf{A}<[\mathrm{X}_{\alpha}, \mathrm{Y}_{\alpha\backslash\beta}], \mathrm{Z}_{\beta\backslash\gamma}]_{\gamma}$
(5) $\mathsf{A}>[\mathrm{X}_{\alpha/\beta}, \mathsf{A}>[\mathrm{Y}_{\beta/\gamma}, \mathrm{Z}_{\gamma}]]_{\alpha}$
(6) $\mathsf{A}<[\mathrm{X}_{\alpha}, \mathsf{A}>[\mathrm{Y}_{(\alpha\backslash\gamma)/\beta}, \mathrm{Z}_{\beta}]]_{\gamma}$
(7) $\mathsf{A}>[\mathrm{X}_{\alpha/\gamma}, \mathsf{A}<[\mathrm{Y}_{\beta}, \mathrm{Z}_{\beta\backslash\gamma}]]_{\alpha}$
(8) $\mathsf{A}<[\mathrm{X}_{\alpha}, \mathsf{A}<[\mathrm{Y}_{\gamma}, \mathrm{Z}_{\gamma\backslash(\alpha\backslash\beta)}]]_{\beta}$

In the system A, (I) is not parsed incrementally because there is a binary node on the right branch in Figure 3. Hence, we employ a new proof system ABTD^{*}, which parses (I) without binary nodes on the right branch. Note that the system allows the

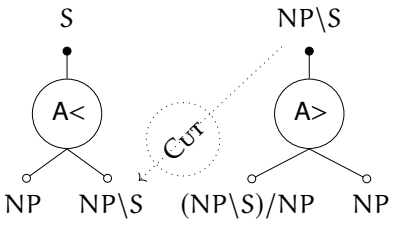


Figure 3.2.: Graph of $A <$ and $A >$

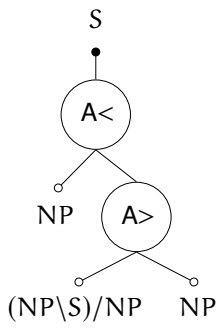


Figure 3.3.: Graph of (I)

cut rule only on the left-most category. Therefore, it is exactly corresponding to the left-branching tree that has no binary nodes on the right branches. Hereafter, we use the binary operator $\circledast$ corresponding to Cut*.

$$\frac{}{\alpha \vdash^* \alpha}\ \mathrm{Id} \qquad \frac{}{\alpha/\beta, \beta \vdash^* \alpha}\ \mathsf{A}> \qquad \frac{}{\alpha, \alpha\backslash\beta \vdash^* \beta}\ \mathsf{A}< \qquad \frac{}{\alpha/\beta, \beta/z \vdash^* \alpha/z}\ \mathsf{B}>$$

$$\frac{}{\alpha/\beta \vdash^* (\alpha/z)/(\beta/z)}\ \mathsf{D}> \qquad \frac{}{\alpha \vdash^* \beta/(\alpha\backslash\beta)}\ \mathsf{T}< \qquad \frac{\Gamma \vdash^* \alpha \qquad \alpha, \Delta \vdash^* \beta}{\Gamma, \Delta \vdash^* \beta}\ \mathrm{Cut}^*$$

In this section, we show the main theorem; For a certain parsing tree generated by CG, we can generate the equivalent parsing tree without binary nodes on the right branch by the combinator rules B, D, T, where the condition is that there are no backward long references (7) and (8) on the above eight patterns.

Theorem 11. *If $\Gamma \vdash a$ is derived by a tree without the backward long reference, then $\Gamma \vdash^* a$, which is derived by a left-branching tree.*

Proof. We use $\mathbb{1}$ and $\mathbb{2}$ to denote applicable single/ binary axiomata. To declare the correspondence, we attach the subscript to these symbols if they are needed. We prove this theorem by the mathematical induction with respect to the number of leaves. If the tree has less than three leaves, it is obviously a left-branching tree. We define the transformation $\triangleright$ of the tree. The following is the inductive definition, that is, we need to process the transformation on the upside to process the ones on the downside. Let the capital Roman letters be parsing trees of CG.

$$\frac{}{x \triangleright x}\ \mathrm{I} \qquad \frac{}{\mathbb{2}[x, y] \triangleright \mathbb{2}[x, y]}\ \mathrm{B} \qquad \frac{\mathbb{2}_1[X, Y] \triangleright A_a \qquad \mathbb{2}_2[a, Z] \triangleright B}{\mathbb{2}_2[\mathbb{2}_1[X, Y], Z] \triangleright A \circledast B}\ \mathrm{E} \qquad \frac{X \triangleright A}{\mathbb{1}[X] \triangleright A \circledast \mathbb{1}[x]}\ \mathrm{U}$$

$$\frac{\mathsf{B}>[X, \mathsf{T}<[Y]] \triangleright A_a \qquad \mathsf{A}>[a, Z] \triangleright B}{\mathsf{A}>[X, \mathsf{A}>[Y, Z]] \triangleright A \circledast B}\ 1 \qquad \frac{\mathbb{2}[X, Y] \triangleright A_a \qquad \mathsf{B}>[a, \mathsf{T}<[Z]] \triangleright B}{\mathsf{B}>[\mathbb{2}[X, Y], \mathsf{T}<[Z]] \triangleright A \circledast B}\ 1\mathrm{L}$$

$$\frac{\mathsf{D}>[X] \triangleright A_a \qquad \mathsf{A}>[a, \mathsf{A}>[Y, Z]] \triangleright B}{\mathsf{B}>[X, \mathsf{T}<[\mathsf{A}>[Y, Z]]] \triangleright A \circledast B}\ 1\mathrm{R} \qquad \frac{\mathsf{T}<[X] \triangleright A_a \qquad \mathsf{A}>[a, \mathsf{A}>[Y, Z]] \triangleright B}{\mathsf{A}<[X, \mathsf{A}>[Y, Z]] \triangleright A \circledast B}\ 5$$

Here, we show the following two properties of the transformation $\triangleright$.

(i) The transformation works for any tree without the backward long reference.

(ii) The transformation terminates and we obtain the left-branching tree.

First, we prove (i) with respect to the outermost form of the tree.

- If the tree is just a category, then it is transformed by I.
- If the tree has only two leaves, then it is transformed by B.
- If the outermost form of the tree is the pattern (1), (2), (3), and (4), then it is transformed by E. Following that, as $\mathbb{2}_1[X, Y]$ and $\mathbb{2}_2[a, Z]$ are the trees of less than n leaves in CG, it can be transformed by $\triangleright$.

- Note that category α in all the transformations appears at the top-most in a tree. This means that the category α remains unchanged by the transformation because there are no more applicable transformation rules to α.
- If the outermost form of the tree is the unary rule, then it is transformed by U. As the tree X is also the tree in CG, it can be transformed to A inductively.
- If the outermost form of the tree is the pattern (5), then it is transformed by 1. There is no backward long reference in the tree by the assumption.
  - For $\mathsf{A}\!>[\alpha, Z]$, there are two cases: Z is a leaf or $\mathsf{A}\!>[\ldots]$. If it is a leaf, then it is transformed by B. Otherwise, as it is a tree of less than n leaves in CG, it is inductively transformed by $\triangleright$.
  - For $\mathsf{B}\!>[X, \mathsf{T}\!<[Y]]$, it is not a tree in CG because of $\mathsf{B}\!>$ and $\mathsf{T}\!<$. Thus, we must define the transformation for each situation. If the outermost form of X is $\mathfrak{D}[\ldots]$, then it is transformed by 1L. If the outermost form of Y is $\mathfrak{D}[\ldots]$, then it is transformed by 1R.
- If the outermost form of the tree is $\mathsf{B}\!>[\mathfrak{D}[X, Y], \mathsf{T}\!<[Z]]$, then it is transformed by 1L. $\mathfrak{D}[X, Y]$ is inductively transformed by $\triangleright$ because it is the tree of less than n leaves in CG. $\mathsf{B}\!>[\alpha, \mathsf{T}\!<[Z]]$ is transformed by 1R.
- If the outermost form of the tree is $\mathsf{B}\!>[X, \mathsf{T}\!<[\mathsf{A}\!>[Y, Z]]]$, then it is transformed by 1R. $\mathsf{D}\!>[X]$ is transformed by U. $\mathsf{A}\!>[\alpha, \mathsf{A}\!>[Y, Z]]$ is transformed inductively because the tree has less than n leaves.
- If the outermost form of the tree is the pattern (6), it is reduced to the pattern (5) by the transformation rule 5. $\mathsf{T}\!<[X]$ is transformed by U. $\mathsf{A}\!>[\alpha, \mathsf{A}\!>[Y, Z]]$ is transformed by (5).
- There is no other outermost form because of the assumption of the theorem.

Next, we prove (ii) for the transformation $\triangleright$. For each transformation, we adjoin the transformed subtree by $\circledast$ to adjoin the left-branching trees without any right-branching. Thus, the transformation generates a left-branching tree only. Moreover, for each transformation step, the depth of the tree and the number of leaves of a subtree decreases. Since the number of leaves and the depth of the tree is finite, the transformation should terminate in finite steps. If we have $\Gamma \vdash \alpha$ holding the assumption, then there is a tree X, and we obtain a certain Y by $X \triangleright Y$. Y is a graph representation of the proof in $\mathcal{ABTD}^*$. Thus, we obtain the proof of $\Gamma \vdash^* \alpha$. □

3.2. Grammar extraction from CCG trees

We explain the extraction of grammar rules from a *constituency tree*, which is a parse tree based on phrase structure grammar as opposed to dependency tree and dependency grammar [39] . The constituency tree and the derivation of the sentenc (5) are in

Figure 2.1 and Figure 2.3, respectively, where we can see the correspondence between the two kinds of parsing trees; a node of the constituency tree corresponds to an application of deduction rule.

Base Case

The minimum constituency tree is a single node tree of a holophrases, *e.g.*, "Thanks" Then, the grammar rule is $thanks \to S : thanks$. The second minimum constituency tree is a binary tree of a two words sentence, *e.g.*, "I see". It has three nodes, S, NP, and VP, for the root, 'I', and 'see', respectively. Then, we consider four deduction rules, $<$, $>$, $< B$, and $> B$ with an atomic category S.

($<$) Let NP be an atomic category. Then, VP is a functional category NP\S.

($>$) Let VP be an atomic category. Then, NP is a functional category S/VP.

($< B$) Let NP and VP are functional categories. Then, the rule deduces a functional category but S is an atomic category.

($> B$) Let NP and VP are functional categories. Then, the rule deduces a functional category but S is an atomic category.

Therefore, we extract two lexicons
$\{I \to NP : I, see \to NP\backslash S : see\}$ and
$\{I \to S/VP : I, see \to VP : see\}$.

Induction Case

We consider the extraction with the larger constituency tree. Let X be a parent node, and Y and Z be child nodes. By induction, we have a procedure to extract the corresponding categories of the child nodes. Then, we consider four deduction rules, $<$, $>$, $< B$, and $> B$ for X.

($<$) Z is a functional category $Y\backslash X$ for any categories X and Y.

($>$) Y is a functional category X/Z for any categories X and Z.

($< B$) Y and Z are functional categories A/B and B/C, respectively, if X is functional category A/C.

($> B$) Y and Z are functional categories $A\backslash B$ and $B\backslash C$, respectively, if X is a functional category $A\backslash C$.

In the above discussion, we can determine the categories of the child nodes from the parent node. By fixing the category of the root node, we can inductively extract a lexicon by applying it from the root node to the leaves.

We deal with the extraction corresponding to the reverse direction of the construction of grammar rules using the four deduction rules. Our direction is based on this correspondence. Once we have a CCG parser (constructor), we can also use it as a grammar extractor. Here, we show this idea in the Prolog program.

Furthermore, this parser is applicable to solve two NLP tasks of *filling a masked sentence* and *generating a sentence* as follows.

- To guess a proper missed word, *e.g.*, "Alice talks ( ) Bob and Cathy."
- To generate an entire sentence based on a set of grammar rules, *e.g.*, "Cathy talks with Alice."

The Definite Clause Grammar (DCG) parser [25] also solves these tasks with a set of grammar rules. Compared to the conventional parsers, our parser is versatile because we show the bidirectional algorithm between a sentence and a tree, and that between a tree and a grammar. Therefore, it also implies the bidirectional algorithm between grammar and a sentence.

Our approach is to employ *bidirectionality* [12] and *non-determinism* [33] in logic programming to extract a set of grammar rules from a treebank.

- Bidirectionality: When we define a rule `twelve(X, Y) :- 12 #= X * Y`, `?- twelve(2, Y).` is `Y = 6`, and `?- twelve(X, 4).` is `X = 3`. In this example, we have originally intended to calculate the multiplication to be 12, given `X` and `Y`, we can change the input/ output to find 3 for `twelve(X, 4)`.
- Non-determinism: In the above example, `?- twelve(X, Y).` returns all factors of 12. In general, the Prolog interpreter finds all solutions by backtracking.

Originally, this approach was introduced to DCG [25] in Prolog, and moreover, it was employed to implement an effective CCG parser [6]. However, the CCG parser is used only in syntactic analysis, which produces a tree from a given grammar, and thus it does not aim at grammar extraction. Here, we consider another direction from grammar to a tree-based in logic programming. Contrary to the common usage of the parser, we consider extracting a set of grammar rules from a tree. Our parser takes two arguments, a set of grammar rules and *a derivation*, which is a proof diagram by CCG that corresponds to a tree structure composed by the grammar, as in Figure 2.3.

First, we show that the parser returns true if the first argument, a set of grammar rules, and the second argument, a derivation are consistent. Thereafter, we reverse the input/ output and utilize this derivation to find all the probable grammar rules to satisfy the truth condition.

As the application of the parser, we implement a software system, *Generator for CCG (GCCG)*, for grammar extraction. The extractor retrieves all the probable grammar rules in Figure 2.2 from a tree in Figure 2.1. Then, the extractor works as a constructor to show all the probable derivations in CCG. An example of derivation is shown in Figure 2.3. In summary, this system solves the following two difficulties regarding CCG.

- The ambiguity of choosing a set of plausible grammar rules.

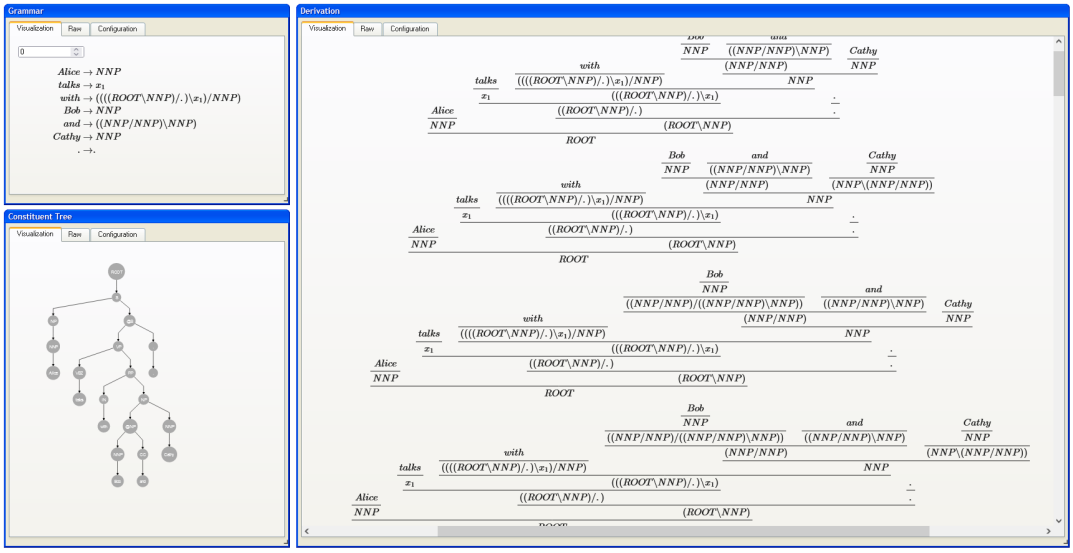


Figure 3.4.: interface of GCCG

- The complexity of all the probable readings by the derivation constructed from the set of grammar rules.

Thus, we expect our system to help annotate the training data with the grammar extraction and to visualize all the probable readings by derivation.

There is no syntax specifying output variables in the usual Prolog because we only know Prolog predicate is true or false. Every argument of predicates is treated equally in the interpreter, which unifies variables according to the sub-goals defined in the rule. For example, a predicate `append/3` takes three arguments, two lists, and a concatenation of them.

Listing 3.1: Definition of append/3

```
1 append([], X, X).
2 append([X|Y], Z, [X|W]) :- append(Y, Z, W).
```

The predicate returns true or false if we pass non-variables to the predicate.

Listing 3.2: Usage of append/3

```
1 ?- append([1, 2], [3, 4], [1, 2, 3, 4]).
2 true
3 ?- append([1, 2], [3, 4, 4], [1, 2, 3, 4]).
4 false
```

However, the interpreter searches a term for a sufficient assignment of a variable if we pass a variable to the predicate.

Listing 3.3: Usage of append/3 with variables

```
?- append([1, 2], [3, 4], Z).
Z = [1, 2, 3, 4]
?- append([1, 2], Y, [1, 2, 3, 4]).
Y = [3, 4]
```

Moreover, the interpreter searches terms for all possible variable assignments of arguments if we pass variable arguments to the predicate.

Listing 3.4: Unification of variables

```
?- append(X, Y, [1, 2, 3, 4]).
X = [] Y = [1, 2, 3, 4];
X = [1] Y = [2, 3, 4];
X = [1, 2] Y = [3, 4];
X = [1, 2, 3] Y = [4];
X = [1, 2, 3, 4] Y = []
```

We commonly use one argument as output and other arguments as input in the Prolog program, *e.g.*, we use the first two arguments as input lists and the third arguments as a result of concatenation. Although we implicitly use the output argument of the predicates, we always change the usage of predicates. For instance, we can use `append/3` to take a sub-list as follows. We call this property the bidirectionality [12] (or bidirectionality, invertibility in some literature) of the Prolog program.

Listing 3.5: Bidirectionality of Prolog

```
?- append(X, _, [1, 2, 3, 4]).
X = [];
X = [1];
X = [1, 2];
X = [1, 2, 3];
X = [1, 2, 3, 4];
```

Next, we discuss the condition of bi-directionality. The I/O operation is obviously prohibited. We cannot get the standard output as the standard input. In the same sense, other destructive operations such as setting environment variables, and the forking processes, are also prohibited. Further, the cut operator sometimes prevents bi-directionality. For example, the following `append/3` is not bidirectional.

Listing 3.6: Non-bidirectional append/3

```
1 append([], X, X).
2 append([X|Y], Z, [X|W]) :- !, append(Y, Z, W).
```

Thus, we take care of introducing a cut operator. Furthermore, some of the arithmetic conditions are not bidirectional. For instance, the inequality `>/2` is not bidirectional. Note that we cannot use the predicates which do not halt for some arguments for this purpose as follows.

Listing 3.7: Bidirectional append/3

```
1 append(X, Y, []) :- append(X, Y, []).
2 append([], X, X).
3 append([X|Y], Z, [X|W]) :- append(Y, Z, W).
```

On these conditions, we implement a CCG parser to make a grammar extractor.

In the Prolog program, we define the infix operators for the functional categories.

Listing 3.8: Infix operators

```
1 :- op(500,xfx,/), op(500,xfx,\), op(550,xfx,@).
```

Since we cannot declare the infix operator `:` to combine a category and a semantic representation, we use `@`. For example, $talk \rightarrow NP\backslash S : talk$ is `lexeme(talk, np\s@talk).` in Prolog program.

Listing 3.9: Lexicon

```
1 lexeme(alice, nnp@alice).
2 lexeme(bob, nnp@bob).
3 lexeme(cathy, nnp@cathy).
4 lexeme(talk, vp/pp@talk).
5 lexeme(with, pp/np@with).
6 lexeme(and, nnp\(np/nnp)@and).
7 lexeme(period, vp\(np\s)@'.').
```

Then, we define deduction rules < and > as follows.

Listing 3.10: Deduction rules

```
1 rule(X@L, X\Y@R, Y@rule('<', X@L, X\Y@R)).
2 rule(X/Y@L, Y@R, X@rule('>', X/Y@L, Y@R)).
```

In the same way, we define the following deduction rules. for $<\mathrm{B}$ and $>\mathrm{B}$, respectively.

Listing 3.11: Composition rules

```
1 rule(X\Y@L, Y\Z@R, X\Z@rule('<B', X\Y@L, Y\Z@R)).
2 rule(X/Y@L, Y/Z@R, X/Z@rule('>B', X/Y@L, Y/Z@R)).
```

For example,
`rule(vp/pp@talk,pp/np@with,s@rule('>B', talk, with)).` is true in this program because of the following derivation.

$$\frac{\mathrm{VP/PP} : talk \qquad \mathrm{PP/NP} : with}{\mathrm{VP/NP} : \lambda x.with\, x\, talk} > \mathrm{B}$$

Finally, we apply these rules to the given term recursively until all categories of lexeme become a leaf of the derivation as follows.

Listing 3.12: Parser in Prolog

```
1 parse0([T], T).
2 parse0(S, T) :-
3     append([X|L], [Y|R], S),
4     parse0([X|L], U),
5     parse0([Y|R], V),
6     rule(U, V, T).
```

At line 1, we define `parse0/2` for a single node tree, which is the base case. At line 3, we split a sentence to two subsentences for the induction case by `append/3`. At line 6 and line 7, we process two subsentences by `parse0/2` inductively. Finally, we apply `rule/3` to the given term at the last line. For instance, we process Sentence (5) as follows.

Listing 3.13: Parsering a sentence

```
1  ?- use_module(library(yall)).
2  ?- {U}/(
3      S = [alice, talk, with, bob, and, cathy, '.'],
4      maplist(lexeme, S, T),
5      parse0(T, U)
6     ).
7  U = s@rule(<, nnp@alice, nnp\s@rule(<, vp@rule(>,
8      vp/pp@talk, pp@rule(>, ... / ... @ with,
9      np@rule(..., ..., ...))), vp\(nnp\s)@('.'))) ;
10 U = s@rule(<, nnp@alice, nnp\s@rule(<, vp@rule(>,
11     vp/pp@talk, pp@rule(>, ... / ... @ rule(...,
```

```
    ..., ...), nnp@cathy)), vp\(nnp\s)@('.'))) ;
U = s@rule(<, nnp@alice, nnp\s@rule(<, vp@rule(>,
    vp/np@rule('>B', ... / ... @ talk, ... / ..
    @ with), np@rule(>, ... / ... @ rule(...,
    ..., ...), nnp@cathy)), vp\(nnp\s)@('.'))) ;
U = s@rule(<, nnp@alice, nnp\s@rule(<, vp@rule(>,
    vp/nnp@rule('>B', ... / ... @ talk, ... / ...
    @ rule(..., ..., ...)), nnp@cathy),
    vp\(nnp\s)@('.'))) ;
U = s@rule(<, nnp@alice, nnp\s@rule(<, vp@rule(>,
    vp/nnp@rule('>B', ... / ... @ rule(..., ...,
    ...), ... / ... @ rule(..., ..., ...)),
    nnp@cathy), vp\(nnp\s)@('.'))) ;
```

This program does two things. One is to search for a grammar rule by `lexeme/2`, and another is to construct a derivation from a set of grammar rules. Further, it shows all the possible readings of the sentence by non-determinism (in the sense of backtracking) of the Prolog program.

In this section, we assume no other deduction rules such as the type-raising. In fact, the program with the type-raising sometimes falls in an infinite loop, and thus, we cannot construct the derivation, so this is a problem in the practical application.

In the reverse direction of the parser, the interpreter returns the set of grammar rules if we use the first argument as an output as follows.

Listing 3.14: Reversal parser

```
?- parse0([ALICE,TALK,WITH,BOB,AND,CATHY,PERIOD],
    s@rule(_, NNP@alice, _S@rule(_, VP@rule(_,
        VBZ@talk, PP@rule(Rule, IN@with,
        NP@rule(_, _NP@rule(_, NNP@bob, CC@and),
        NNP@cathy))), PRD@'.'))).
ALICE = _NP\(IN\(VBZ\VP))@alice,
TALK = VBZ@talk,
WITH = IN@with,
BOB = _NP\(IN\(VBZ\VP))@bob,
AND =  (_NP\(IN\(VBZ\VP)))\_NP@and,
CATHY = _NP\(IN\(VBZ\VP))@cathy,
PERIOD = VP\((_NP\(IN\(VBZ\VP)))\s)@('.'),
NNP = _NP\(IN\(VBZ\VP)),
_S =  (_NP\(IN\(VBZ\VP)))\s,
PP = VBZ\VP,
Rule =  (<),
NP = IN\(VBZ\VP),
CC =  (_NP\(IN\(VBZ\VP)))\_NP,
PRD = VP\((_NP\(IN\(VBZ\VP)))\s) ;
```

Then, the above code returns all the probable grammar rules extracted from a tree by non-determinism (in the sense of backtracking) of the Prolog program. We show the number of sets of grammar rules as follows.

Listing 3.15: Number of the sets of grammar rules

```
?- findall(1,
    (parse0([A, T, W, B, N, C, P],
      s@rule(_, NNP@alice, _S@rule(_, VP@rule(_,
      VBZ@talk, PP@rule(_, IN@with, NP@rule(_,
      _NP@rule(_, NNP@bob, CC@and), NNP@cathy))),
      PRD@'.'))),
      acyclic_term([A, T, W, B, N, C, P])),
    L),
   length(L, N).
N = 265,
L = [1, 1, 1, 1, 1, 1, 1, 1, 1|...].
```

The result shows 265 sets of grammar rules extracted from this tree. It is hard for us to find out all the probable syntactic assignments manually. This is the reason why we employ this parser for the interactive support system.

We mainly used one argument as an output for the grammar constructor/extractor. Let us consider passing two arguments as output to the predicate.

Listing 3.16: Generating a sentence

```
?- between(1, inf, N), length(S, N),
   maplist(lexeme, Sentence, Categories),
   parse0(Categories, s@Tree).
N = 2,
Sentence = [alice, talk],
GrammarRules = [np@alice, np\s@talk],
Tree = rule(lapp, talk, alice) .
```

The above query produces all the probable sentences using lexeme registered in the database. We, further, retrieve a set of grammar rules and a derivation with respect to the sentence. Moreover, we can fill a masked sentence "Alice talks ( ) Bob and Alice." as follows.

Listing 3.17: Masked sentence

```
?- {UNKNOWN}/(
    S = [alice, talk, UNKNOWN, bob, and, cathy, '.'],
    maplist(lexeme, S, T),
    parse0(T, s@_)
```

```
5 ).
6 UNKNOWN = with
```

The Prolog program exhaustively searches for unknown variables that appeared in a masked sentence, and next, it constructs a derivation with categories. Finally, it returns the word if the unification is successful. Those functions are similar to Definite Clause Grammar (DCG) in logic programming. However, our program also produces a set of grammar rules additionally, and this is superior to the DCG parser.

When we develop a lexicon for CCG, a visualization of a constituency tree and a running example would be helpful because it is hard for us to know a behavior of a large category such as a preposition, e.g, $(((\mathrm{ROOT}\backslash\mathrm{NNP})/.)\backslash x_1)/\mathrm{NNP}$. We present an all-in-one system named GCCG, which provides three functions.

1. Visualizing a constituency tree using Stanford CoreNLP
2. Generating all the probable lexicons from the constituency tree
3. Showing all the probable derivations constructed from the lexicon

This system provides three functionalities: to show all the probable lexicons at the left top corner, to visualize a constituency tree at the left-bottom corner, and to show all the probable derivations constructed from the lexicon at the right, as shown in Fig 3.4.

Note that we introduce a new predicate `rule/2` in addition to `rule/3`. Hence, we employ the unary deduction rules, which are the type-raising rules. we make the parser predicate semi-decidable, *i.e.*, the parser accepts every expected string and otherwise rejects or loops in the program. The unary deduction rules would be applied infinitely because of the Prolog unification strategy as follows.

$$\cfrac{\cfrac{\cfrac{\vdots}{X : x}}{Y/(X\backslash Y) : \lambda f.fx}}{Z/((Y/(X\backslash Y))\backslash Z) : \lambda z.z(\lambda f.fx)}$$

$$\vdots$$

Hence, the program may not halt if the given input is impossible to be unified. Further, we cannot guarantee the bi-directionality of the parser among all the arguments. However, according to the linguistic observation, we can restrict the search path of the parser, and thus, the program halt for all arguments. Therefore, we still can guarantee bidirectionality except for the sentence generation.

3.3. Interactive Incremental CCG Parser

The compositionality principle states that the semantics is obtained in parallel to the completion of a syntactic structure. Thus, to obtain a partial semantics we need to

assume that there is a partial tree structure even in the middle of an utterance. To achieve this property, we employ combinatory categorial grammar (CCG) [32], which enables us to acquire a semantic structure and a parsing tree simultaneously. However, those conventional grammar rules in CCG are insufficient for incremental parsing. Therefore, we introduce new rules as a natural extension of existing formalisms, and in addition, we implement an interactive parser that can externalize a parsing tree for each word input.

For example, we consider obtaining a tree of "Alice" and "Alice loves" from the sentence "Alice loves Bob," found in the incremental parsing of the sentence. Then, we construct a partial tree whenever a new succeeding word appears, regarding it as a token consumption. As a result, we find a left-branching tree in Figure 1, without employing extra memory for floating tokens that are un-treed words.

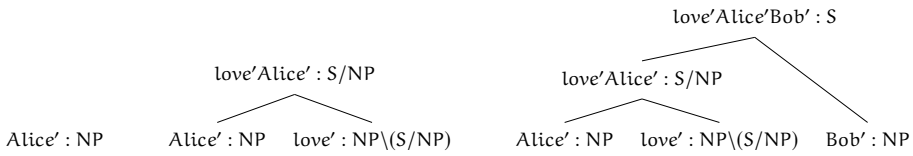


Figure 3.5.: Transition of the left-branching tree

From the semantic point of view, we need to preserve the semantics, that is a sequence of terms obtained from lexical items, between the non-left-branching tree and the left-branching tree if they consist of the same categories. This is because categorial grammar and its derivations are in the simply-typed lambda calculus; that is, the reduction steps do not affect the resultant sequence of semantics terms.

From the syntactic point of view, we can generally assign multiple different categories for each word because these categories always remain equivocal until the end of the sentence. However, we do not argue this issue in this thesis; instead, we deal with all the possible category assignments.

Stanjević and Steedman [29] implemented an incremental parsing algorithm employing the tree rotation which depends on the following composition rules.

$$X/Y \quad W\backslash(Y/Z) \Rightarrow_{<B^2_\times} W\backslash(X/Z) \qquad\qquad W\backslash(Y/Z) \quad Y\backslash X \Rightarrow_{>B^2} W\backslash(X/Z)$$

As the first contribution, we argue that the introduction of B^2 and $B^2_\times$ in [29] are rather haphazard and are not linguistically indispensable. In general, it is not preferable to introduce new rules to solve certain specific language phenomena. If we admit the introduction of B^2, $B^2_\times$ or possible introduction of B^3 mentioned in [29], we may also need to consider B^4 or more B^n. Considering the possibility that we may face such a situation, we should provide a general principle to tolerate any number of arguments for a given word. This is the reason why we would like to reduce these binary, or n-ary operations to unary derivation rules.

We here get back to the principles of CCG and introduce a set of derivation rules of

D, Dx, Q, Qx.

$$X/Y \Rightarrow_{>D} (X/Z)/(Y/Z) \qquad X/Y \Rightarrow_{>Q} (Z/X)\backslash(Z/Y) \qquad X\backslash Y \Rightarrow_{<D} (Z\backslash X)\backslash(Z\backslash Y)$$
$$X\backslash Y \Rightarrow_{<Q} (X\backslash Z)/(Y\backslash Z) \qquad X/Y \Rightarrow_{>Dx} (Z\backslash X)/(Z\backslash Y) \qquad X\backslash Y \Rightarrow_{>Qx} (Z/Y)\backslash(X\backslash Z)$$
$$X\backslash Y \Rightarrow_{<Dx} (X/Z)\backslash(Y/Z) \qquad X/Y \Rightarrow_{<Qx} (Z/Y)/(X\backslash Z)$$

Here are comparisons;

- B^2 and B^2_x correspond to two operations; one is the composition with two arguments and another is to indicate the head of the constituent.
- D, Dx, Q, and Qx correspond to the single operation that is to indicate the head of the constituent as the same as the type-raising rules, *e.g.*, a pair of categories $X/Y, Y$ is to be $X/Y, (X/Y)\backslash X$ in a derivation, by which we switch the argument part to the functional part, and so do the functional part of the argument part.

Note that while we can derive B^2 and D^2_x from D, Dx, Q, and Qx, the reverse does not hold. Therefore, our extension is a generalization of the preceding work [29]. Furthermore, our new rules are unary so that the derivation is more versatile and can derive a certain set of sentences that are not obtained only by binary rules. Instead, the derivation steps may increase when we employ only unary rules.

As the second contribution, we implement our CCG parser in logic programming. The main predicate takes three arguments as in `parse(Grammar, Sentence, Tree)`, namely, a set of grammar rules, a sentence, and a tree. Generally, we build a constituency tree structure from a sentence using the parser with some given grammar rules, *i.e.*, the building process is a sequence of unification between applicable `Grammar` rules and a given `Sentence`. However, we can execute the parser in the reversal way, as in definite clause grammar [25]; when we execute the unification between an input `Sentence` and a parsed `Tree`, we can replicate a set of innate `Grammar` rules.

Based on these two principles, we implement an interactive and incremental CCG parser. This system takes a sentence as input and displays the possible parsing results in CCG. For each word input, the partial sentence is processed as follows;

1. A syntax tree is created for the partial sentence using Stanford CoreNLP. This result is shown in the left-bottom corner of Figure 2.
2. The tree and the sentence are used to extract the grammar rules employed in the parser. This result is shown in the left-top corner of Figure 2.
3. Based on the extracted grammar, the parser outputs all possible parsing results in CCG. This result is shown on the right-hand side of Figure 2.

In the grammar extraction stage, the part of speech (POS) is regarded as a variable for logic programming, and grammar rules are extracted by the process of unification. For this reason, as mentioned above, the type of the partial tree is not uniquely determined, and so all the possible grammar rules are exhaustively consulted. With this system, we illustrate the growing process of a left-branching tree. Moreover, it also shows how the parser builds a semantics structure from a sentence.

Table 3.1.: Extension of CCG Rules

$X/Y \quad Y \Rightarrow_{>} X$		$Y \Rightarrow_{>T} (X/Y)\backslash X$	
$X \quad X\backslash Y \Rightarrow_{<} Y$		$X \Rightarrow_{<T} Y/(X\backslash Y)$	
$X/Y \quad Y/Z \Rightarrow_{>B} X/Z$		$X/Y \Rightarrow_{>D} (X/Z)/(Y/Z)$	$X/Y \Rightarrow_{>Q} (Z/X)\backslash(Z/Y)$
$X\backslash Y \quad Y\backslash Z \Rightarrow_{<B} X\backslash Z$		$X\backslash Y \Rightarrow_{<D} (Z\backslash X)\backslash(Z\backslash Y)$	$X\backslash Y \Rightarrow_{<Q} (X\backslash Z)/(Y\backslash Z)$
$X/Y \quad Z\backslash Y \Rightarrow_{>Bx} Z\backslash X$		$X/Y \Rightarrow_{>Dx} (Z\backslash X)/(Z\backslash Y)$	$X\backslash Y \Rightarrow_{>Qx} (Z/Y)\backslash(X\backslash Z)$
$X/Y \quad X\backslash Z \Rightarrow_{<Bx} Z/Y$		$X\backslash Y \Rightarrow_{<Dx} (X/Z)\backslash(Y/Z)$	$X/Y \Rightarrow_{<Qx} (Z/Y)/(X\backslash Z)$

3.4. Losing a head in Grammar Extraction

A constituency tree reflects an important syntactic structure of a natural language sentence. The tree consists of the constituent nodes labeled by the part of speech (POS) and the leaves labeled by lexical items. For example, we parse Sentence 6 as Figure 3.6.

(6) I really love you.

The corresponding language of the constituency tree is the context-free language (CFL), which only requires the reference information of sibling nodes, *i.e.*, there is no crossing dependency between the nodes.

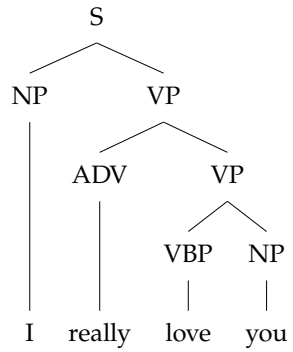


Figure 3.6.: Constituency of Sentence 6

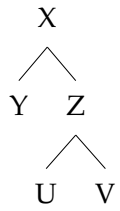


Figure 3.7.: Multiple nodes

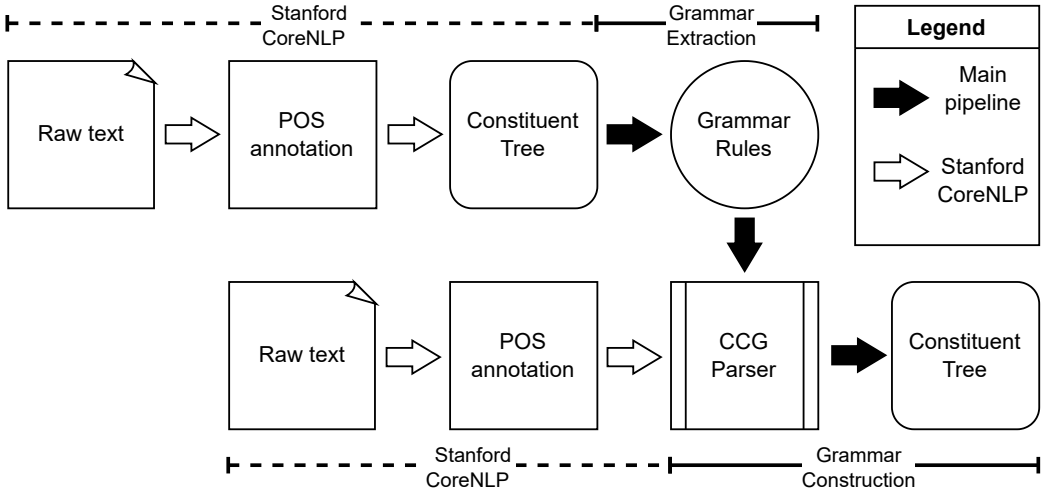


Figure 3.8.: Pipeline of CCG parsing

In the state-of-the-art machine learning study, we build a language model with a collection of constituency trees called a treebank [20]. To train the language model using the constituency trees, we extract grammar rules from this tree with categorial grammar (CG) [32], of which the generative power is the same as CFL.

Our approach is based on pattern matching, regarding the constituency tree. The grammar extraction algorithm [38] is summarized as follows; Transform each node, *e.g.*, the root node in Figure 3.7, to a grammar rule: $Y \vdash X/Z$ and $Z \vdash Y\backslash X$, where $\mapsto$ is the mapping from a word to a category. The following is an example of the extracted grammar rules of Sentence 6.

$$\mathrm{I} \mapsto \mathrm{NP} \qquad \texttt{really} \mapsto \mathrm{ADV} \qquad \texttt{love} \mapsto \mathrm{VBP} \qquad \texttt{you} \mapsto \mathrm{NP}$$

$$\mathrm{NP} \vdash \mathrm{S/VP} \qquad \mathrm{ADV} \vdash \mathrm{VP/VP} \qquad \mathrm{VBP} \vdash \mathrm{VP/NP}$$

In the X-bar theory of Chomsky [8], phrase structure is described as the following pattern as shown in Figure 3.9.

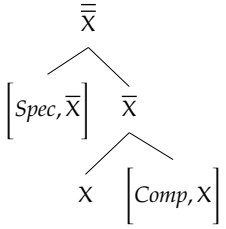


Figure 3.9.: X-bar Schema

$$\left(\left[Spec, \overline{X}\right] \backslash \overline{\overline{X}}\right) / \left[Comp, X\right]$$

Figure 3.10.: Category of the head

Let X be the main constituent. Then, the structure has two extra constituents: $\left[Spec, \overline{X}\right]$ and $\left[Comp, X\right]$. The *Comp* is the auxiliary complement of X; *e.g.*, NP is a complement of VBP in Figure 3.6. The *Spec* is the auxiliary specifier of X; *e.g.*, ADV is the specifier of VP in Figure 3.6. X becomes $\overline{X}$ by the compliment and becomes $\overline{\overline{X}}$ by the specifier additionally. Here, X is the main component of the constituency; *i.e.*, X is the *head* of this phrase structure.

The left side of '$\bullet \vdash \bullet$' is a head of the clause; *e.g.*, VP and VBP are heads of Figure 3.6. Further, from the viewpoint of semantic analysis, we commonly deal with the head as a predicate. For example, the verb 'love' is a head of Sentence 6, and it represents a predicate logic form $\mathtt{love}(\mathtt{I}, \mathtt{you})$.

However, in this thesis, we aim at showing to loss of head information by the grammar extraction regarding the pipeline as shown in Figure 3.8. Here, *'Losing a head'* refers to missing important information. Further, we show how we supplement head information with the generalized version of the type-raising rule [32]. We employ the supplement to the grammar contraction phase.

In this section, we solve two membership problems.

1. What sentences are generated by the language?
   e.g., Sentence 6 is a member of English sentences.

2. How much generative power does the grammar have?
   e.g., the grammar of English sentences is a member of mildly context-sensitive grammar (MCSG).

We introduce two formalisms characterized by MICSG and context-free grammar (CFG), in which the phrase structure relation does not cross over each other in a sentence; that is, the representation is a tree. We parse Sentence 6 as Figure 3.6.

We formalize the tree structure focusing on the *biting* relation, which is a ternary relation that a node bites a sibling node to construct a parent node. In Figure 3.7, Y takes Z and returns X. Hereafter, we denote the relation as $Y \vdash X/Z$. Further, there is a similar relation in Figure 3.7; Z takes Y and returns X, which is denoted as $Z \vdash Y\backslash X$.

The atomic category is a constituent label, which is a part of speech (POS). The functional category is a pair of categories bonded by the constructors / and \. The functional categories X/Y and Y\X is a head for the category Y, where the head is the main factor to construct the higher category X.

We define the system to parse a sentence from the grammar rules. Let the capital Greek letters be sequences of categories. We initially define an axiom schema $X \vdash X$. Further, We construct a category from two construction rules.

$$\frac{\Gamma \vdash X \qquad Y, \Delta \vdash Z}{\Gamma, X\backslash Y, \Delta \vdash Z}\backslash \qquad \frac{\Gamma \vdash X \qquad \Delta, Y \vdash Z}{\Delta, Y/X, \Gamma \vdash Z}/$$

Further, we substitute a category by another construction rule.

$$\frac{\Gamma, X, \Delta \vdash Y \qquad \Sigma \vdash X}{\Gamma, \Sigma, \Delta \vdash Y}\,\text{Cut}$$

Using the three rules, we parse a sentence. For example, we parse Sentence 6 as follows.

$$\frac{\dfrac{\text{VP} \vdash \text{VP} \qquad \text{NP} \vdash \text{NP}}{\text{VP/NP}, \text{NP} \vdash \text{VP}}/ \qquad \text{VBP} \vdash \text{VP/NP}}{\text{VBP}, \text{NP} \vdash \text{VP} \cdots \text{(i)}}\,\text{Cut}$$

$$\frac{\dfrac{\text{VP} \vdash \text{VP} \qquad \text{VP} \vdash \text{VP}}{\text{VP/VP}, \text{VP} \vdash \text{VP}}/ \qquad \text{ADV} \vdash \text{VP/NP}}{\text{ADV}, \text{VP} \vdash \text{VP} \cdots \text{(ii)}}\,\text{Cut}$$

$$\frac{\dfrac{\text{S} \vdash \text{S} \qquad \text{VP} \vdash \text{VP}}{\text{S/VP}, \text{VP} \vdash \text{S}}/ \qquad \text{NP} \vdash \text{S/NP}}{\text{NP}, \text{VP} \vdash \text{S} \cdots \text{(iii)}}\,\text{Cut}$$

$$\frac{\dfrac{\begin{array}{c}\text{(i)}\\ \vdots \\ \text{VBP}, \text{NP} \vdash \text{VP}\end{array} \qquad \begin{array}{c}\text{(ii)}\\ \vdots \\ \text{ADV}, \text{VP} \vdash \text{VP}\end{array}}{\text{ADV}, \text{VBP}, \text{NP} \vdash \text{VP}}/ \qquad \begin{array}{c}\text{(iii)}\\ \vdots \\ \text{NP}, \text{VP} \vdash \text{S}\end{array}}{\text{NP}, \text{ADV}, \text{VBP}, \text{NP} \vdash \text{S}}\,\text{Cut}$$

We introduce combinatory categorial grammar (CCG) [32] to deal with a superset of context-free language (CFL), which is an extension of CG. The additional construction rules are corresponding to combinatory logic [15]. The following list is a common set of combinators for CCG.

- Application: f and a then fa
- Type-raising: a then $\lambda f.fa$
- Composition: f and g then $\lambda x.f(g(x))$

There are variants of CCG with additional combinators. By extra combinators, the generative power becomes greater than CFG. Since it is not enough to parse natural language sentences by CFG, we commonly employ CCG so to do.

The following forms are axiom schemata of the application. Note that we can obtain the same form using CG.

$$X/Y, Y \vdash X \qquad\qquad Y, Y\backslash X \vdash X$$

Next, the following are axiom schemata of the type-raising combinator. This is a currying variant of the application. We move the functional category to the right-hand side.

$$Y \vdash (X/Y)\backslash X \qquad\qquad Y \vdash X/(Y\backslash X)$$

Last, the following forms are axiom schemata of the composition combinator. As a side note, the composition is a syllogism in Logic. Let $X \to Y$ and $Y \to Z$ then we conclude $X \to Z$.

$$X/Y, Y/Z \vdash X/Z \qquad\qquad X\backslash Y, Y\backslash Z \vdash X\backslash Y$$

Additionally, we define the non-standard combinator called **Q**-combinator and **D**-combinator [37, 23]. These are employed in CCG to the incremental natural language parser. The concept of these combinators is the generalization of the type-raising to deal with the composition as the same as the application. The following forms are axiom schemata by the movement of the left-most functional category to the right-hand side.

$$Y/Z \vdash (X/Y)\backslash(X/Z) \qquad\qquad Y\backslash Z \vdash (X\backslash Y)\backslash(X\backslash Y)$$

The following forms are axiom schemata by the movement of the right-most functional category to the right-hand side.

$$X/Y \vdash (X/Z)/(Y/Z) \qquad\qquad X\backslash Y \vdash (X\backslash Z)/(Y\backslash Z)$$

For example, we parse Sentence 6 as follows.

$$\frac{\mathrm{VP/VP}, \mathrm{VP/NP} \vdash \mathrm{VP/NP} \qquad \mathrm{ADV} \vdash \mathrm{VP/VP}}{\mathrm{ADV}, \mathrm{VP/NP} \vdash \mathrm{VP/NP}}\,\mathrm{Cut}$$

$$\frac{\mathrm{ADV}, \mathrm{VP/NP} \vdash \mathrm{VP/NP} \qquad \mathrm{VBP} \vdash \mathrm{VP/NP}}{\mathrm{ADV}, \mathrm{VBP} \vdash \mathrm{VP/NP}}\,\mathrm{Cut}$$

$$\frac{\mathrm{ADV}, \mathrm{VBP} \vdash \mathrm{VP/NP} \qquad \mathrm{VP/NP}, \mathrm{NP} \vdash \mathrm{VP}}{\mathrm{ADV}, \mathrm{VBP}, \mathrm{NP} \vdash \mathrm{VP}}\,\mathrm{Cut}$$

$$\frac{\mathrm{S/VP}, \mathrm{VP} \vdash \mathrm{S} \qquad \mathrm{NP} \vdash \mathrm{S/NP}}{\mathrm{NP}, \mathrm{VP} \vdash \mathrm{S}}\,\mathrm{Cut}$$

$$\frac{\text{NP}, \text{VP} \vdash \text{S} \qquad \text{ADV}, \text{VBP}, \text{NP} \vdash \text{VP}}{\text{NP}, \text{ADV}, \text{VBP}, \text{NP} \vdash \text{S}}\ \text{Cut}$$

Grammatical information is a significant feature of a language model. The grammar information is not only for the syntactic analysis but also for the semantic analysis to construct a first-order predicate logic representation. Then, We obtain the semantic representation $\mathtt{really}(\mathtt{love}(\mathtt{you}))(\mathtt{I})$ from Sentence 6 by Montague Semantics approach [22] as follows.

$$\begin{aligned}
(1)\cdots &\quad \text{vp/np}, \text{np} \vdash \text{vp} \\
(2)\cdots &\quad \text{vp/vp}, \text{vp} \vdash \text{vp} \\
(3)\cdots &\quad \text{vp} \vdash \text{np}\backslash\text{s} \\
(4)\cdots &\quad \text{np}, \text{np}\backslash\text{s} \vdash \text{s}
\end{aligned}$$

$$\begin{aligned}
(1)\cdots &\quad \lambda x.\mathtt{love}(x)\ \ \mathtt{you} \rightarrow \mathtt{love}(\mathtt{you}) \\
(2)\cdots &\quad \lambda x.\mathtt{really}(x)\ \ \mathtt{love}(\mathtt{you}) \rightarrow \mathtt{really}(\mathtt{love}(\mathtt{you})) \\
(3)\cdots &\quad \mathtt{really}(\mathtt{love}(\mathtt{you})) \rightarrow \lambda x.\mathtt{really}(\mathtt{love}(\mathtt{you}))(x) \\
(4)\cdots &\quad \mathtt{i}\ \lambda x.\mathtt{really}(\mathtt{love}(\mathtt{you}))(x) \rightarrow \mathtt{really}(\mathtt{love}(\mathtt{you}))(\mathtt{i})
\end{aligned}$$

Since the semantic representation relies on the header information, we need grammar extraction as a functional category in CG from the constituency tree. The functional category is corresponding to the biting relation between child nodes and a parent node of the constituency tree. In this section, we show the main pipeline of the grammar extraction system and detail the pattern matching of the constituency tree.

We employ Stanford CoreNLP [19] to annotate the raw text by POS tags and obtain a constituency tree. The main pipeline of our system is the following three steps:

1. Extract grammar from constituency trees.
2. Employ the grammar to the CCG parser.
3. Construct another constituency tree.

As was shown in Figure 3.8 in Section I, we show the detail of Step 1. For Step 2 and Step 3, we follow the CCG parsing algorithm described in Section II. This CCG parser is written in the logic programming language Prolog [38].

We show the grammar extraction first. We retrieve a constituency tree such as Figure 3.6. The tree is a binary tree in which a node has two child nodes as a branch. We decompose the tree into the ternary relation. The relation of CG is the application of functional category as follows.

$$\text{X/Y}, \text{Y} \vdash \text{X} \qquad\qquad \text{Y}, \text{Y}\backslash\text{X} \vdash \text{X}$$

Additionally, we also consider the relation of CCG. Note that the relation is not only the application but also the other combinator, which takes two arguments. The

Algorithm 1 Grammar extraction

```
1:  function EXTRACT_GRAMMAR(node)
2:      if node has child nodes then
3:          L_0, child_1, child_2 ← node
4:          L_1, _, _ ← child_1
5:          L_2, _, _ ← child_2
6:          X, Y, Z are fresh categories
7:          grammar_0 ∈ {
              {L_1 ⊢ X/Y, L_2 ⊢ Y, X ⊢ L_0},
              {L_1 ⊢ Y, L_2 ⊢ Y\X, X ⊢ L_0},
              {L_1 ⊢ X/Y, L_2 ⊢ Y/Z, X/Z ⊢ L_0},
              {L_1 ⊢ X\Y, L_2 ⊢ Y\Z, X\Z ⊢ L_0}}
8:          grammar_1 ← EXTRACT_GRAMMAR(child_1)
9:          grammar_2 ← EXTRACT_GRAMMAR(child_2)
10:         return grammar_0 ∪ grammar_1 ∪ grammar_2
11:     else
12:         return ∅
13:     end if
14: end function
```

following is the composition **B**-combinator [32].

$$X/Y, Y/Z \vdash X/Z \qquad\qquad X\backslash Y, Y\backslash Z \vdash X\backslash Y$$

Algorithm 1 is a pseudocode converting each branch of the constituency to the above four grammar rules. We recursively run the function at Line 8 and Line 9. Initially, we call the function with the root node of the given tree. At Line 3, we decompose the node into the POS tag L_0 and the child nodes $child_1, child_2$, which also have POS tag L_1, L_2 respectively. Next, let X, Y, Z be fresh categories. Then, we choose one of the grammar rules at Line 7. At Line 10, we merge the chosen grammar with other grammar regarding the child nodes, $child_1$ and $child_2$. If the node has no child nodes, *i.e.*, the node is a leaf, then the set of grammar rules is the empty $\emptyset$. Note that '_' is the wildcard, which means that we never refer them.

We first annotate the raw text by standard CoreNLP. Next, we parse the annotated text. Finally, we apply Algorithm 1 to the constituency tree. Then, we obtain the set of grammar rules. The number of possible grammar patterns exponentially increases concerning the number of branches. Each branch has four choices. Therefore, there are 4^N sets of grammar rules, where N is the number of nodes. For simplicity, we choose one grammar for each node.

In Algorithm 1, we only consider the binary tree that has no node with a single child node. However, we can deal with such a tree by the addition of new grammar corresponding to the type-raising and **Q**-combinator [37] and **D**-combinator [23] because these combinators take a single argument.

We show the result of the grammar extraction of Sentence 6 with the constituency tree of Fig 3.6 by Algorithm 1. A, B, C, D, E and F are fresh categories generated by the extraction algorithm.

$$\begin{array}{ccc} VP \vdash A\backslash B & NP \vdash A & B \vdash S \\ VP \vdash C\backslash D & ADV \vdash C & D \vdash VP \\ VBP \vdash E/F & NP \vdash F & E \vdash NP \end{array}$$

We obtain the above grammar rules by choosing the grammar rule corresponding to the application. Here, VP and VBP are functional categories; *i.e.*, they are the head of this constituency tree. Note that the head of the constituency tree might be changed by what grammar rules were chosen for each node.

In grammar construction as shown in Figure 3.8 in Section I, we finally obtain the following form:

$$NP, ADV, VBP, NP \vdash S$$

We employ the construction algorithm as shown in Section II. The fresh variables generated by the grammar extraction algorithm will be reduced by the following construction rule.

$$\frac{\Gamma, X, \Delta \vdash Y \qquad \Sigma \vdash X}{\Gamma, \Sigma, \Delta \vdash Y}$$

Moreover, there are no such categories in the conclusion as shown in Section II.

We obtain the grammar from the given text using the grammar extraction of our pipeline. In this section, we inspect that the algorithm extracts grammatical features of the constituency tree with some information loss. We show that one of the lost information is the header information of the constituency tree by grammar extraction.

We find this pattern on the constituency tree and extract the head information. Note that the specifier and the complement of X-bar theory [8] are rearranged manually for each target language. In Figure 3.6, the example of the head is VBP, which takes NP as the complement and does ADV as the specifier.

In CG and CCG, X is the functional category taking the compliment and the specifier. Then, Figure 3.10 is the functional category representation corresponding to X.

The grammar extraction characterizes the constituent as the functional category and its argument category. In this sense, the functional category is the head of the clause. At Line 7 in Algorithm 1, there are four choices for the clause into the grammar rules.

- Choice 1: The clause is the application and the left child node is the head because of the functional category.
- Choice 2: The clause is the application and the right child node is the head because of the functional category.
- Choice 3: The clause is the composition and there are no heads because both categories are functional.

- Choice 4: The clause is the composition and there are no heads because both categories are functional.

In Choice 1, we lose the possibility that the right child node is the head. In Choice 2, we lose the possibility that the left child node is the head. Further, in Choice 3 and Choice 4, we lose the head information regarding the current node. Losing a head is problematic according to the X-bar theory.

Algorithm 1 misses some head information from the constituency tree. Thus, in this section, we show to supplement the lost information with the type-raising and its variants. The head is corresponding to the functional category. Hence, we consider how to make the category functional in CCG.

First, we show handling this problem regarding Choice 1 and Choice 2 in Section IV. The following grammar rule is available in both grammar formalism: CG and CCG.

$$X/Y, Y \vdash X \qquad\qquad Y, Y\backslash X \vdash Y$$

To make the category functional, we employ the following type-raising rule of CCG.

$$X \vdash Y/(X\backslash Y) \qquad\qquad X \vdash (Y/X)\backslash Y$$

We can transpose the biting relation of application; *i.e.*, we make Y functional by the type-raising rule.

$$X/Y, (X/Y)\backslash Y \vdash Y \qquad\qquad Y/(X\backslash Y), X\backslash Y \vdash Y$$

This is the way to supplement head information using the type-raising rule. Note that type-raising is not a rule of CG. Thus, we can supplement heads only in CCG.

We, next, inspect the same way to supplement the head for Choice 3 and Choice 4. In the following composition rules, both categories on the left-hand side are functional.

$$X/Y, Y/Z \vdash X/Z \qquad\qquad X\backslash Y, Y\backslash Z \vdash X\backslash Z$$

With only the standard CCG rules, we cannot supplement the head information because we need to run the conversion from the category to the complicated category recursively; *e.g.*, X/Y becomes $(X/Z)/(Y/Z)$ to be the head of the node. The problem with the conversion is that the category Z pops up from inside the deeper part of the complicated functional category. Hence, we cannot employ the type-raising rule as the same as the application.

Here, we employ the four unary rules relating **Q**-combinator and **D**-combinator, which are defined as the same as the type-raising rule. The type-raising rule is a variant of the application rules by the movement of category from the left-hand side to the right-hand side. Similarly, the following rules are variants of the composition rule.

$$\begin{aligned} Y/Z &\vdash (X/Y)\backslash(X/Z) & \qquad Y\backslash Z &\vdash (X\backslash Y)\backslash(X\backslash Y) \\ X/Y &\vdash (X/Z)/(Y/Z) & \qquad X\backslash Y &\vdash (X\backslash Z)/(Y\backslash Z) \end{aligned}$$

These rules work as the same as supplementing the head of the application rule. Each head part becomes a complicated functional category.

$$(X/Y)/(Y/Z), Y/Z \vdash X/Z \qquad X/Y, (X/Y)\backslash(Y/Z) \vdash X/Z$$
$$(X\backslash Y)/(Y\backslash Z), Y\backslash Z \vdash X\backslash Z \qquad X\backslash Y, (X\backslash Y)\backslash(Y\backslash Z) \vdash X\backslash Z$$

Note that these extensions widen the target language; *e.g.*, the grammar rule $(Y/X)\backslash Y, ((Y/X)\backslash Y)\backslash Z \vdash Z$ might appear in the parsing phrase in CCG but it so does not in CG because CG has no type-raising rule. This is the same as other combinators.

3.5. Formalization of CG and CCG

We show the formal proof of the generative power of CG and CCG. Further, we also inspect the inclusion relation between the variants of CG. The formal proof is written in Isabelle/HOL [24], which is the proof assistant system.

We inductively define CG as follows. This definition contains the three grammar rules of CG: left application `LA`, right application `RA`, and `Cut`. The capital Greek letters are sequences of categories and `@` is the concatenation operator of the sequences.

Listing 3.18: Inductive Definition of CG

```
1  inductive CG ::
2    "'a category list ⇒ 'a category ⇒ bool"
3    (infix "⊢CG" 60)
4  where
5    LA  : "[B←A, A] ⊢CG B"
6  | RA  : "[A, A→B] ⊢CG B"
7  | Cut : "⟦Γ ⊢CG A; Δ@[A]@Σ⊢CG B⟧
8           ⟹ Γ@Δ@Σ ⊢CG B"
```

In the same manner, we define CCG as follows. Since CCG is the extension of CG, it contains `LA`, `RA`, and `Cut`. Additionally, we also define the left composition `LB`, right composition `RB`, and type-raising `LT` and `RT`.

Listing 3.19: Inductive Definition of CCG

```
1  inductive CCG ::
2      "'a category list ⇒ 'a category ⇒ bool"
3      (infix "⊢CCG" 60)
4  where
5    LA : "[B←A, A] ⊢CCG B"
6  | LT : "[A] ⊢CCG (B←A)→B"
7  | LB : "[A←B, B←C] ⊢CCG A←C"
8  | RA : "[A, A→B] ⊢CCG B"
9  | RT : "[A] ⊢CCG B←(A→B)"
```

```
  | RB : "[A→B, B→C] ⊢CCG A→C"
  | Cut : "⟦Γ ⊢CCG A; Δ@[A]@Σ⊢CCG B ⟧
           ⟹ Δ@Γ@Σ⊢CCG B"
```

The following is the formal proof of inclusion relation; *i.e.,* CCG includes CG. We initially show that $\Gamma \vdash A$ in CCG if $\Gamma \vdash A$ in CG. Next, we show the theorem of inclusion.

Listing 3.20: Inclusion relation between CG and CCG (1)

```
lemma CG_is_CCG :
  fixes Γ :: "'a category list"
    and A :: "'a category"
  assumes "Γ ⊢CG A"
  shows "Γ⊢CCG A"
  apply (rule CG.induct)
  by(simp_all add:assms CCG.LA CCG.RA CCG.Cut)
```

Listing 3.21: Inclusion relation between CG and CCG (2)

```
theorem CCG_includes_CG :
  fixes f :: "'a ⇒ 'b category"
    and S :: "'b category"
  assumes "LCG = {W . map f W ⊢CG S}"
      and "LCCG = {W . map f W ⊢CCG S}"
    shows "LCG ⊆ LCCG"
apply (simp add: assms)
using CG_is_CCG by auto
```

Finally, we formalize the variants of CCG with combinators **Q** and **D**, which is called QCCG [37]. Compared to CCG, the additional rules are `LD`, `LQ`, `RD`, and `RQ`.

Listing 3.22: Inductive Definition of QCCG

```
inductive QCCG ::
    "'a category list ⇒ 'a category ⇒ bool"
    (infix "⊢QCCG" 60)
  where
    LA : "[B←A,A] ⊢QCCG B"
  | LT : "[A] ⊢QCCG (B←A)→B"
  | LB : "[A←B,B←C] ⊢QCCG A←C"
  | LD : "[A←B]⊢QCCG (A←C)←(B←C)"
  | LQ : "[A←B]⊢QCCG (C←A)→(C←B)"
  | RA : "[A,A→B] ⊢QCCG B"
  | RT : "[A] ⊢QCCG B←(A→B)"
```

```
  | RB : "[A→B,B→C] ⊢QCCG A→C"
  | RD : "[A→B]⊢QCCG (C→A)→(C→B)"
  | RQ : "[A→B]⊢QCCG (A→C)←(B→C)"
  | Cut : "⟦Γ ⊢QCCG A; Δ@[A]@Σ⊢QCCG B⟧
           ⟹ Δ@Γ@Σ⊢QCCG B"
```

The following is the formal proof of inclusion relation regarding our grammar formalism; *i.e.*, QCCG includes CCG. We initially show the lemma: if $\Gamma \vdash A$ in CCG then $\Gamma \vdash A$ in QCCG. Next, we show the theorem of inclusion.

Listing 3.23: Inclusion relation between CCG and QCCG (1)

```
lemma CCG_is_QCCG :
  fixes Γ :: "'a category list"
    and A :: "'a category"
  assumes "Γ ⊢CCG A"
  shows "Γ⊢QCCG A"
  apply (rule CCG.induct)
  apply (simp add: assms)
  by (simp_all add: QCCG.LA QCCG.LT QCCG.LB
           QCCG.RA QCCG.RT QCCG.RB QCCG.Cut)
```

Listing 3.24: Inclusion relation between CCG and QCCG (2)

```
theorem QCCG_includes_CCG :
  fixes f :: "'a ⇒ 'b category"
    and S :: "'b category"
  assumes "LCCG = {W . map f W ⊢CCG S}"
      and "LQCCG = {W . map f W ⊢QCCG S}"
    shows "LCCG ⊆ LQCCG"
apply (simp add: assms) done
```

The above formal proof shows that grammar formalism is the legitimate extension of the standard CG and CCG. Hence, the generative power of this extension is superior to the original grammar. Moreover, grammar solves losing the head information.

In this section, we showed the grammar extraction algorithm to construct the set of grammar rules from the given treebank according to the pipeline as shown in Figure 3.8 based on Algorithm 1. In order to obtain categorial grammar, we only extracted application rules for each branch. According to the algorithm's strategy, we non-deterministically chose only one grammar rule from the candidates. To obtain combinatory categorial grammar, we extracted composition rules and categorial grammar. In addition to the extraction, we showed the grammar construction in the sequent calculus; we chained each grammar rule by a cut rule.

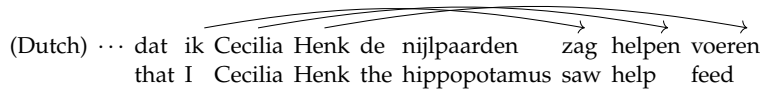
Although the head is a significant feature in syntactic theory, as well as in semantic parsing of Montague Semantics. Since our algorithm could not retrieve the head information, we considered compensating for the lost information. To do this, we employed the type-raising rule in categorial grammar. Further, in combinatory categorial grammar, we utilized the extra combinators, **Q** and **D**. By these combinators, we recursively converted categories to complicated functional ones to clarify the biting relation.

4. Proof System of Categorial Grammar

4.1. Cross-serial Dependency

Cross-serial dependency is a phenomenon that word references over other references, which is investigated with CCG by [32]. The following Dutch sentence has a cross-serial dependency for example.

(Egnlish) $\cdots$ that I saw Cecilia help Henk feed the hippopotamus ((4.1))



The verbs *zag*, *helpen* and *voeren* depend on the noun phrases *ik*, *Cecilia*, and *Henk* respectively. Since the dependency of each pair is crossing on the sentence. This dependency does not appear in Context-Free Grammar (CFG) as $\{a^n b^m a^n b^m\}$ is not CFG. [32] provided a new rule in CCG as *Dutch forward crossed composition*.

[32] provided a new rule in CCG as *Dutch forward crossed composition*.

Definition 12 (Dutch forward crossed composition)**.** Dutch forward crossed composition is defined as;

$$\frac{X/Y : f \qquad Z\$\backslash Y : \lambda z^Z\$.gz\$}{Z\$\backslash X : \lambda z^Z\$.f(gz\$)}$$

where Y is a verb phrase of subordinate sentence VP_{-SUB}, $Y\backslash Z\$$ is reputation of typing rule.

However, we propose the new rule in CCG instead of Dutch forward crossing composition because this rule depends on the syntactic type $Y = V_{-SUB}$.

Definition 13 (CPS Transformation in CCG [36])**.** CPS-transformation of combinators in CCG is defined as;

$$\frac{U/V\$: f}{(X/V\$)/(U\backslash X) : \lambda k^{U\backslash X}.\lambda v^V\$.k(fv\$)}\ /\text{CPS}$$

$$\frac{V\$\backslash U : f}{(X/U)\backslash(V\$\backslash X) : \lambda k^{X/U}.\lambda v^V\$.k(fv\$)}\ \backslash\text{CPS}$$

This rule is corresponding to the type-raising rule, which is mentioned in its relationship to the CPS-transformation of the noun phrase by [10]. The subject type-raising rule and this rule are the CPS transformation. In the fact, we get the subject type-raising rule when $U/V\$ = V$ has no reputation in $\$$.

$$\frac{U : f}{X/(U\backslash X) : \lambda k^{U\backslash X}.kf}\ /\mathrm{CPS}^0$$

$$\frac{U : f}{(X/U)\backslash X : \lambda k^{X/U}.kf}\ \backslash\mathrm{CPS}^0$$

Thus we can also remove subject type-raising from rules if we add the above two rules.

We show the result of parsing sentence (4.1). Since the two rules $/\mathrm{CPS}$ and $\backslash\mathrm{CPS}$ are type-raising rules in contrast to that Dutch forward crossed a composition rule, the step was increased but we use only the basic composition rules in the result.

4.2. Decidable Algorithm for CG with Type-raising

CG is a formalism for representing natural language syntax. We assign a category to each word and a rule to each phrase. The category consists of two-directional arrows. For example, the verb phrase is represented by the category $\mathsf{NP}\backslash\mathsf{S}$ as the verb phrase takes a subject from the left-hand side, *e.g.*, 'He **walks**'. For another example, the adverb is represented by the category $\mathsf{ADJ}/\mathsf{ADJ}$ as the adverb takes an adjective from the right-hand side, *e.g.*, '**very** fast'. As these arrows correspond to the implication in Logic, we can use the theorem-proving approach to parse natural language by CG.

CCG is an extension of CG with combinatory rules to analyze linguistic phenomena. One of the combinatory rules is the T combinator called *type-raising rule* to correspond with swapping a head in X-bar theory, where the head takes another component. For instance, X is raised to $Y/(X\backslash Y)$ by the T combinator. The verb phrase takes noun phrase, *i.e.*, we regard the verb phrase as a head $\mathsf{NP}\backslash\mathsf{S}$. By the T combinator, the noun phrase is a head $\mathsf{S}/(\mathsf{NP}\backslash\mathsf{S})$.

In CG and its variants, we parse a sentence by proving the theorem $\Gamma \vdash \mathsf{S}$ where Γ is a sequence of categories, *e.g.*, "He walks" is given by $\mathsf{NP}, \mathsf{NP}\backslash\mathsf{S} \vdash \mathsf{S}$. Since the natural language sentence is not commutative, we could not exchange the categories in Γ. Moreover, the sequent calculus of CG with the type-raising rule is defined as follows, where Roman letters are categories, and Greek letters are sequences of categories.

$$\frac{}{X \vdash X}\ \mathrm{Id} \qquad \frac{}{X/Y,\ Y \vdash X}\ \mathrm{A}> \qquad \frac{}{X,\ X\backslash Y \vdash Y}\ \mathrm{A}<$$

$$\frac{}{Y \vdash (X/Y)\backslash X}\ \mathrm{T}> \qquad \frac{}{X \vdash Y/(X\backslash Y)}\ \mathrm{T}< \qquad \frac{\Gamma \vdash X \qquad \Sigma, X, \Delta \vdash Y}{\Sigma, \Gamma, \Delta \vdash Y}\ \mathrm{Cut}$$

As the non-axiom rule is only the cut rule, the proof is not cut-free. The non-cut-free proof is a problem for the decidability of the sequent calculus. Thus, it is also a problem for the parsing algorithm. Especially T combinator is the hard rule for decidability.

Hence, there is a limitation in the usage of the rule in most CCG parsing algorithms. For example, a parser allows the type raising only for the noun phrase. In this thesis, we eliminate the limitation of the type-raising rule by the proof-theoretic analysis.

First, we show the decidability of the parsing algorithm in CG without the type-raising rule [34]. We represent the parsing tree of CG shown in Figure 1 as a sequent calculus proof shown in Figure 2. In each branch, the length of the antecedent increases for each cut rule if the antecedent is non-empty. The number of candidates of parsing tree with $n+1$ words is $\frac{1}{n+1} \cdot {}_{2n}C_n$, which is a Catalan Therefore, we could decide whether the given sentence is grammatical in finite steps.

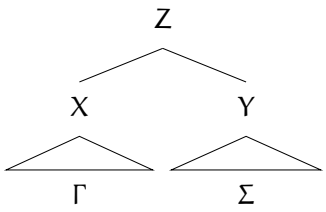


Figure 4.1.: Tree of $\Gamma, \Sigma \vdash Z$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash X} \quad \dfrac{}{X, Y \vdash Z}\, A<\text{ or }A> \quad \vdots}{\dfrac{\Gamma, Y \vdash Z}{\Gamma, \Sigma \vdash Z}\,\text{Cut}\qquad \Sigma \vdash Y}\,\text{Cut}$$

Figure 4.2.: Proof of $\Gamma, \Sigma \vdash Z$

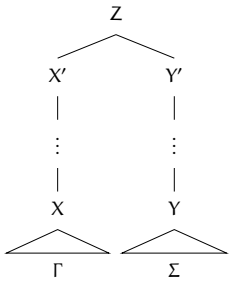


Figure 4.3.: Tree of $\Gamma, \Sigma \vdash Z$ with T

Next, we show the decidability of the parsing algorithm in CG with the type-raising rule. The parsing tree is formed by the cut rule and the type-raising rule, as shown in Figure 3. By the type-raising rule, we produce many candidates of trees. Thus, the above algorithm never halts if the sentence is ungrammatical. We here show the lemma: If a tree has unary rules in both branches, there is another parsing tree without the unary

rules in one branch. The type-raising rule is swapping a head. Thus, to swap a head in both branches is 'swapping and swapping again.' At least one swapping is redundant. By the lemma and the analysis of the complexity of the category (the number of / and \), we show the theorem: The number of type-rasing in Figure 3 is less than the maximum number of / and \ in the categories X and Y. Therefore, we prove each binary branch in finite steps. This is the decidability of the parsing algorithm in CG with the type-raising rule.

This section showed the decidable algorithm to deduce s from a given sentence. In formal grammar, we remove the limitation of the type-raising rule in categorial grammar.

4.3. Provability of CPS transformation

The *ad-hoc* grammar rules were employed to deal with linguistic phenomena in categorial grammar (CG). Such a haphazard introduction potentially exceeds the original grammar class in Chomsky's hierarchy; the grammar rules may over-generate ungrammatical sentences. For example, the subordinate clause "that cat walks" is scrambled into "cat that walks" by the cross-composition rule (Bx) [32].

$$\dfrac{\mathrm{that} : \mathrm{SBAR/S} \qquad \dfrac{\mathrm{cat} : \mathrm{NP} \qquad \mathit{walks} : \mathrm{NP\backslash S}}{\mathrm{cat}\,\mathit{walks} : \mathrm{S}}\,<}{\mathrm{that}\,\mathrm{cat}\,\mathit{walks} : \mathrm{SBAR}}\,>$$

$$\dfrac{\mathrm{cat} : \mathrm{NP} \qquad \dfrac{\mathrm{that} : \mathrm{SBAR/S} \qquad \mathit{walks} : \mathrm{NP\backslash S}}{\mathrm{that}\,\mathit{walks} : \mathrm{NP\backslash SBAR}}\,>\mathrm{B}x}{\mathrm{cat}\,\mathrm{that}\,\mathit{walks} : \mathrm{SBAR}}\,<$$

Hence, we verify the generative power of added grammar rules using the formal method. Our targets are CPS transformation rules investigated by Plotkin [26] and Barker [2]. Note that there are differences in generative power between them. We show that Plotkin's CPS transformation rule is unprovable in Lambek calculus [18], which is the mathematical formalization of categorial grammar. Further, we show that these grammar rules might generate inappropriate sentences from the unprovability.

There are two motivations for the CPS transformation rule. One is to modify the scope of quantifiers, and another is to extend grammar rules for a specific linguistic phenomenon. Generally, the first motivation is not problematic because it does not violate the original syntax theory. However, the second motivation is disputable as to whether it solves only the targeted phenomenon. For example, both the type-raising rule and the cross-composition rule are well-known grammar rules in CG, however, actually, the former is provable in Lambek calculus while the latter is not. The following proof is a usage of the CPS transformation rule. In this thesis, we regard that only those sentences provable by calculus are grammatical.

$$\dfrac{\dfrac{\mathit{cat} : \mathrm{NP}}{\mathit{cat} : \mathrm{S}/(\mathrm{NP}\backslash\mathrm{S})}\,\mathrm{CPS} \qquad \mathit{walks} : \mathrm{NP}\backslash\mathrm{S}}{\mathit{cat}\,\mathit{walks} : \mathrm{S}}\,>$$

In Lemma 5, we showed the provability of the type-raising rules by showing the derivation in Lambek Calculus. Here, On the contrary, we show the *unprovability* of a CCG rule in LC.

Lemma 14 (Unprovability of cross-composition in LC)**.** *The sequent* $\alpha/\beta, \alpha\backslash\gamma \vdash \gamma/\beta$ *is unprovable in LC.*

We must show that there is no way to derive the goal sequent from the initial sequent. For instance, three forms of sequents are unprovable in LC as follows, where the atomic term φ is different from the atomic term ψ. The third form below, for example, is derived from $\varphi \vdash \varphi$ and $\vdash \psi$, which are reduced to another unprovable sequent.

$$\vdash \varphi \qquad\qquad \varphi \vdash \psi \qquad\qquad \varphi, \psi \vdash \varphi \tag{4.2}$$

We search all the possible rule applications in LC and show that each derivation path closes with an unprovable sequent. Assume that α, β, γ are atomic. In Figure 4.4, we show a graph of derivation paths from the goal sequent p2. Note that the sequent is unprovable if at least one sequent in presumptions is unprovable. The red edge is I\. The blue edge is I/. The green edge is \I. The purple edge is /I. All leaves close with the unprovable sequent.

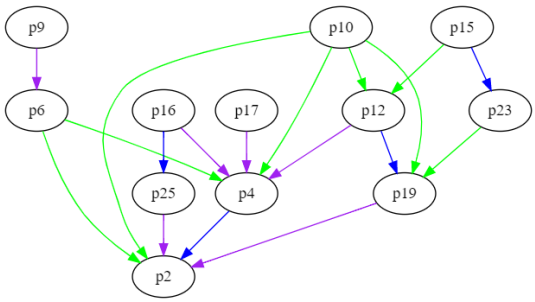


Figure 4.4.: Unprovability of cross-composition

p9 : $\vdash \beta$	p6 : $\alpha/\beta \vdash \alpha$	p10 : $\vdash \alpha$	p15 : $\gamma, \beta \vdash \gamma$
p12 : $\alpha, \alpha\backslash\gamma, \beta \vdash \gamma$	p16 : $\alpha, \beta \vdash \gamma$	p17 : $\alpha \vdash \gamma$	p4 : $\alpha/\beta, \alpha\backslash\gamma, \beta \vdash \gamma$
p23 : $\gamma \vdash \gamma/\beta$	p19 : $\alpha, \alpha\backslash\gamma \vdash \gamma/\beta$	p25 : $\alpha \vdash \gamma/\beta$	p2 : $\alpha/\beta, \alpha\backslash\gamma \vdash \gamma/b$

The type of lambda calculus is not directed, while LC is two-directional. In other words, if we translate the type properties of the lambda calculus into LC, there are

multiple possible choices of translations. Thus, the CPS transformation is not unique in LC. For instance, Barker's CPS transformation becomes as follows.

$$\alpha \vdash \beta/(\alpha\backslash\beta) \qquad\qquad \alpha \vdash (\beta/\alpha)\backslash\beta \tag{4.3}$$

The only two transformations above are introduced as the CPS transformations [2] because both sequence are provable by Lemma 5. Thus, Barker's CPS transformation is harmless as to provability in Lambek calculus if we add them as a basic rule. However, other variations $\alpha \vdash \beta\backslash(\beta\backslash\alpha)$, $\alpha \vdash (\alpha/\beta)/\beta$, and Plotkin's CPS transformation are not.

Definition 15 (Plotkin's CPS transformation in LC)**.** Let γ be an answer type and A be a set of all atomic terms of LC. Two relations $\cdot \xrightarrow{\gamma} \cdot$ and $\cdot \overset{\gamma}{\rightarrow} \cdot$ are inductively defined as $\langle\langle\cdot\rangle\rangle$ and $\langle\cdot\rangle$, respectively.

$$\frac{\alpha \overset{\gamma}{\rightarrow} \tau}{\alpha \xrightarrow{\gamma} \gamma/(\tau\backslash\gamma)} \quad \frac{\alpha \overset{\gamma}{\rightarrow} \tau}{\alpha \xrightarrow{\gamma} (\gamma/\tau)\backslash\gamma} \quad \frac{\alpha \in A}{\alpha \overset{\gamma}{\rightarrow} \alpha} \cdot \frac{\alpha \overset{\gamma}{\rightarrow} \tau \;\; \beta \xrightarrow{\gamma} \upsilon}{\alpha\backslash\beta \overset{\gamma}{\rightarrow} \tau\backslash\upsilon} \cdot \frac{\alpha \overset{\gamma}{\rightarrow} \tau \;\; \beta \xrightarrow{\gamma} \upsilon}{\beta/\alpha \overset{\gamma}{\rightarrow} \upsilon/\tau}$$

Compared to Barker's CPS transformation, we execute the transformation recursively. Thus, the size of the resulting term increases exponentially. Moreover, we must try all possible rules to find a derivation path from the goal sequent to leaves. Since the generated proof is lengthy, we only show the graph of the unprovable derivation paths (see Figure 4.5).

Theorem 16 (Unprovability of Plotkin's CPS transformation in LC)**.** *There exists an unprovable sequent* $\varphi \vdash \psi$ *even if* $\varphi \xrightarrow{\delta} \psi$*, where* δ *is an answer type.*

We consider the sequent $\gamma/(\beta/\alpha) \vdash \psi$ where α, β, γ are atomic and $\gamma/(\beta/\alpha) \xrightarrow{\delta} \psi$. Here, $\psi \equiv \delta/(((\delta/(\gamma\backslash\delta))/((\delta/(\beta\backslash\delta))/\alpha))\backslash\delta)$, which is a CPS-transformed term. We search all the possible rule applications in LC and show that each derivation path closes with an unprovable sequent.

p17 :$\gamma, \delta \vdash \gamma$
p14 :$\gamma, (\delta/(\beta\backslash\delta))/\alpha \vdash \gamma$
p27 :$\delta \vdash \beta$
p29 :$\delta/(\beta\backslash\delta) \vdash \beta$
p33 :$\delta \vdash \beta/\alpha$
p21 :$(\delta/(\beta\backslash\delta))/\alpha \vdash \beta/\alpha$
p35 :$\gamma/(\beta/\alpha), \delta/(\beta\backslash\delta) \vdash \gamma$
p46 :$\delta \vdash \gamma$
p43 :$(\delta/(\beta\backslash\delta))/\alpha \vdash \gamma$
p53 :$\gamma, \delta, \gamma\backslash\delta \vdash \delta$
p51 :$\gamma, \delta/(\beta\backslash\delta), \gamma\backslash\delta \vdash \delta$
p49 :$\gamma, (\delta/(\beta\backslash\delta))/\alpha, \gamma\backslash\delta \vdash \delta$
p73 :$\gamma/(\beta/\alpha), \delta, \gamma\backslash\delta \vdash \delta$
p65 :$\gamma/(\beta/\alpha), \delta/(\beta\backslash\delta), \gamma\backslash\delta \vdash \delta$

p16 :$\gamma, \delta/(\beta\backslash\delta) \vdash \gamma$
p26 :$\delta, \alpha \vdash \beta$
p25 :$\delta/(\beta\backslash\delta), \alpha \vdash \beta$
p23 :$(\delta/(\beta\backslash\delta))/\alpha, \alpha \vdash \beta$
p31 :$\delta/(\beta\backslash\delta) \vdash \beta/\alpha$
p39 :$\gamma/(\beta/\alpha), \delta \vdash \gamma$
p12 :$\gamma/(\beta/\alpha), (\delta/(\beta\backslash\delta))/\alpha \vdash \gamma$
p45 :$\delta/(\beta\backslash\delta) \vdash \gamma$
p47 :$\vdash \gamma$
p54 :$\gamma, \delta \vdash \delta$
p56 :$\gamma, \delta/(\beta\backslash\delta) \vdash \delta$
p63 :$\gamma \vdash \delta$
p81 :$\gamma/(\beta/\alpha), \delta \vdash \delta$
p83 :$\gamma/(\beta/\alpha), \delta/(\beta\backslash\delta) \vdash \delta$

p10 : $\gamma/(\beta/\alpha), (\delta/(\beta\backslash\delta))/\alpha, \gamma\backslash\delta \vdash \delta$
p87 : $\gamma, \delta/(\beta\backslash\delta) \vdash \delta/(\gamma\backslash\delta)$
p109 : $\gamma/(\beta/\alpha), \delta \vdash \delta/(\gamma\backslash\delta)$
p8 : $\gamma/(\beta/\alpha), (\delta/(\beta\backslash\delta))/\alpha \vdash \delta/(\gamma\backslash\delta)$
p6 : $\gamma/(\beta/\alpha) \vdash (\delta/(\gamma\backslash\delta))/((\delta/(\beta\backslash\delta))/\alpha)$
p136 : $\beta \vdash \gamma$
p133 : $\gamma\backslash\delta \vdash \beta\backslash\delta$
p143 : $\beta \vdash \delta$
p138 : $\delta/(\beta\backslash\delta) \vdash \delta$
p147 : $\delta \vdash \delta/(\gamma\backslash\delta)$
p123 : $(\delta/(\beta\backslash\delta))/\alpha \vdash \delta/(\gamma\backslash\delta)$
p149 : $\gamma, ((\delta/(\gamma\backslash\delta))/((\delta/(\beta\backslash\delta))/\alpha))\backslash\delta \vdash \delta$
p151 : $\gamma \vdash \delta/(((\delta/(\gamma\backslash\delta))/((\delta/(\beta\backslash\delta))/\alpha))\backslash\delta)$

p89 : $\gamma, \delta \vdash \delta/(\gamma\backslash\delta)$
p85 : $\gamma, (\delta/(\beta\backslash\delta))/\alpha \vdash \delta/(\gamma\backslash\delta)$
p99 : $\gamma/(\beta/\alpha), \delta/(\beta\backslash\delta) \vdash \delta/(\gamma\backslash\delta)$
p119 : $\gamma \vdash (\delta/(\gamma\backslash\delta))/((\delta/(\beta\backslash\delta))/\alpha)$
p129 : $\delta, \gamma\backslash\delta \vdash \delta$
p135 : $\beta, \gamma\backslash\delta \vdash \delta$
p127 : $\delta/(\beta\backslash\delta), \gamma\backslash\delta \vdash \delta$
p142 : $\vdash \beta\backslash\delta$
p125 : $(\delta/(\beta\backslash\delta))/\alpha, \gamma\backslash\delta \vdash \delta$
p145 : $\delta/(\beta\backslash\delta) \vdash \delta/(\gamma\backslash\delta)$
p121 : $\vdash (\delta/(\gamma\backslash\delta))/((\delta/(\beta\backslash\delta))/\alpha)$
p4 : $\gamma/(\beta/\alpha), ((\delta/(\gamma\backslash\delta))/((\delta/(\beta\backslash\delta))/\alpha))\backslash\delta \vdash \delta$
p2 : $\gamma/(\beta/\alpha) \vdash \delta/(((\delta/(\gamma\backslash\delta))/((\delta/(\beta\backslash\delta))/\alpha))\backslash\delta)$

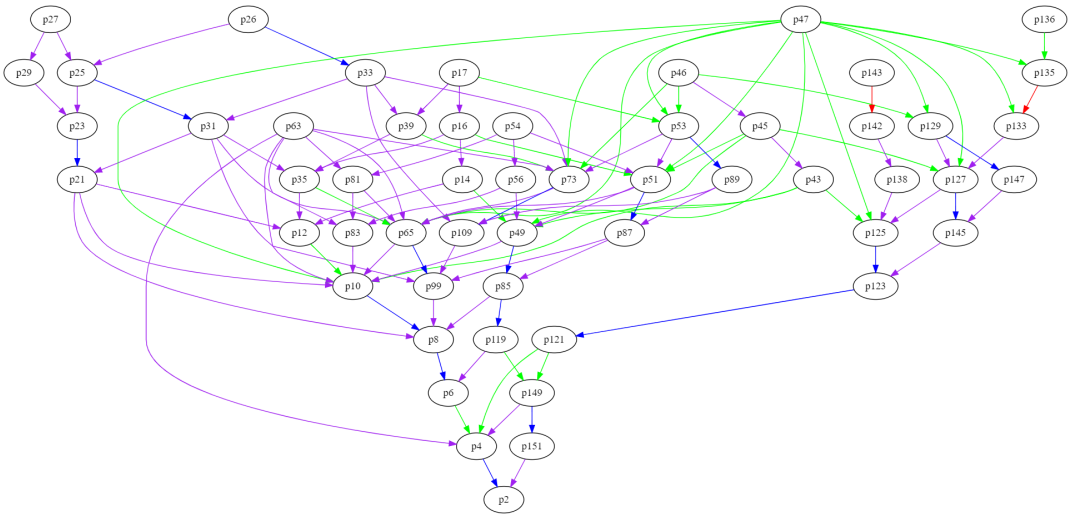


Figure 4.5.: Proof of unprovability

Thus far, we showed Baker's CPS transformation is provable in LC, and Plotkin's CPS transformation is not. Hereafter, we propose a moderate CPS translation which is the same translation of Plotkin's CPS transformation, but we restrict the types of lambda terms. Our CPS translation is closer to Plotkin's CPS transformation than Barker's but is still provable in LC.

Here, we inductively define two relations $\cdot \overset{\gamma}{\twoheadrightarrow} \cdot$ and $\cdot \overset{\gamma}{\rightarrowtail} \cdot$ corresponding to $\langle\langle\cdot\rangle\rangle$ and $\langle\cdot\rangle$, respectively.

Definition 17 (Type-restricted CPS transformation in LC)**.** Let γ be an answer type and A be a set of all atomic terms of LC. We inductively define two relations $\cdot \overset{\gamma}{\twoheadrightarrow} \cdot$ and $\cdot \overset{\gamma}{\rightarrowtail} \cdot$ corresponding to $\langle\langle\cdot\rangle\rangle$ and $\langle\cdot\rangle$, respectively.

$$\frac{\alpha \overset{\gamma}{\rightharpoonup} \tau}{\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \gamma/(\tau\backslash\gamma)} \quad \frac{\alpha \overset{\gamma}{\rightharpoonup} \tau}{\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} (\gamma/\tau)\backslash\gamma} \quad \frac{\alpha \in A}{\alpha \overset{\gamma}{\rightharpoonup} \alpha} \cdot \frac{\alpha \in A \;\; \beta \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \upsilon}{\alpha\backslash\beta \overset{\gamma}{\rightharpoonup} \alpha\backslash\upsilon} \cdot \frac{\alpha \in A \;\; \beta \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \upsilon}{\beta/\alpha \overset{\gamma}{\rightharpoonup} \upsilon/\alpha}$$

Note that there is no transformation from a higher-order functional term, which recursively takes other functional terms, *e.g.*, $\gamma/(\beta/\alpha)$, that caused the failure of proof in Plotkin's CPS transformation. Theorem 18 shows that the transformation in Definition 17 is provable in LC.

Theorem 18 (Provability of type-restricted CPS transformation in LC)**.** *Let* γ, φ *and* ψ *be terms of LC. Then, (i)* $\alpha \vdash \beta$ *if* $\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \beta$*, and moreover, (ii)* $\alpha \vdash \beta$ *if* $\alpha \overset{\gamma}{\rightharpoonup} \beta$.

We mutually prove both statements (i) and (ii) by mathematical induction for the length of α.

1. Assume that α is atomic.
   a) Assume $\alpha \overset{\gamma}{\rightharpoonup} \alpha$. $\alpha \vdash \alpha$ holds.
   b) Assume $\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} (\gamma/\alpha)\backslash\gamma$. $\alpha \vdash (\gamma/\alpha)\backslash\gamma$ holds by Lemma 5.
   c) Assume $\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \gamma/(\alpha\backslash\gamma)$. $\alpha \vdash \gamma/(\alpha\backslash\gamma)$ holds by Lemma 5.
2. Assume $\alpha = \beta\backslash\delta$ and $\delta \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \eta$.
   a) Assume $\alpha \overset{\gamma}{\rightharpoonup} \beta\backslash\eta$. As β is atomic by Definition 17, $\beta \vdash \beta$ holds. Moreover, $\delta \vdash \eta$ holds by the induction hypothesis. Thus, $\beta\backslash\delta \vdash \beta\backslash\eta$.
   b) Assume $\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \gamma/((\beta\backslash\eta)\backslash\gamma)$. In the same way as Proof 2a, $\beta\backslash\delta \vdash \beta\backslash\eta$. Thus, $\alpha \vdash \gamma/((\beta\backslash\eta)\backslash\gamma)$ holds by Lemma 5.
   c) Assume $\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} (\gamma/(\beta\backslash\eta))\backslash\gamma$. In the same way as Proof 2a, $\beta\backslash\delta \vdash \beta\backslash\eta$. Thus, $\alpha \vdash (\gamma/(\beta\backslash\eta))\backslash\gamma$ holds by Lemma 5.
3. Assume $\alpha = \delta/\beta$ and $\delta \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \eta$
   a) Assume $\alpha \overset{\gamma}{\rightharpoonup} \eta/\beta$. As β is atomic by Definition 17, $\beta \vdash \beta$ holds. Moreover, $\delta \vdash \eta$ holds by the induction hypothesis. Thus $\delta/\beta \vdash \eta/\beta$.
   b) Assume $\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} \gamma/((\eta/\beta)\backslash\gamma)$. In the same way as Proof 3a, $\delta/\beta \vdash \eta/\beta$. Thus, $\alpha \vdash \gamma/((\eta/\beta)\backslash\gamma)$ holds by Lemma 5.
   c) Assume $\alpha \overset{\gamma}{\rightarrow\!\!\!\!\rightarrow} (\gamma/(\eta/\beta))\backslash\gamma$. In the same way as Proof 3a, $\delta/\beta \vdash \eta/\beta$. Thus, $\alpha \vdash (\gamma/(\eta/\beta))\backslash\gamma$ holds by Lemma 5.

Therefore, (i) and (ii) hold.

We showed the sketches of proof for Lemmata. Next, we verify them by Isabelle/HOL. Further, we also provide proof of the restricted CPS transformation. First, we translate Lambek calculus to Isabelle/HOL as follows. `category` is a terminology of categorial grammar. It corresponds with a type of simply-typed lambda calculus. In this section,

we use ← and → instead of / and \ because it is hard to use / and \ in Isabelle/HOL. Further, ^ is a mark to denote a symbol as an atomic symbol.

Listing 4.1: Category of Lambeck Calculus

```
1 datatype 'a category =
2   Atomic 'a ("^")
3   | RightFunctional "'a category" "'a category" (infix "→" 60)
4   | LeftFunctional "'a category" "'a category" (infix "←" 60)
```

We next define the system of Lambek calculus as follows. `@` is a concatenation of a list of data structures. `[x]⊢x` is a sequent $x \vdash x$. Further, `X@[x]⊢y` is a sequent $X, x \vdash x$. `⟦...;...⟧` is a set of assumptions. ⟹ is an implication of Isabelle/HOL. The following is a translation of Definition 3.

Listing 4.2: Lambek Calculus

```
inductive LC::
  "'a category list ⇒ 'a category ⇒ bool" (infix "⊢" 55)
  where
    r1: "([x]⊢x)"
  | r2: "(X@[x]⊢y) ⟹ (X⊢y←x)"
  | r3: "([x]@X⊢y) ⟹ (X⊢x→y)"
  | r4: "⟦(Y⊢y); (X@[x]@Z⊢z)⟧ ⟹ (X@Y@[y→x]@Z⊢z)"
  | r5: "⟦(X@[x]@Z⊢z); (Y⊢y)⟧ ⟹ (X@[x←y]@Y@Z⊢z)"
```

By the above source code, we translate the proof sketch of Lemma 14. We show the counter-example of the statement that the cross composition is provable. Let `a,b,c` be atomic and be different from each other. Then the following code is the verification of Figure 4.4 where `subst`,`simp`, and `simp_all` are substitution commands to modify the original statement to the simplified statement. `auto` and `fastforce` are automatic deduction commands to test the provability.

Listing 4.3 is the proof of Lemma 14. The label for each line, such as `p23`, is a label of each deduction step shown in Figure 4.4. The following code verifies the exhaustiveness of the proof by `auto` and `fastforce`.

Listing 4.3: Unprovability of Cross composition

```
theorem
  assumes "distinct [a,b,c,d]"
  shows "¬([^a←^b,^a→^c]⊢^c←^b)"
proof -
have p9: "¬([]⊢^b)"
  apply auto apply (subst (asm) LC.simps) by auto
have p6: "¬([^a← ^b]⊢^a)"
  apply auto apply (subst (asm) LC.simps) apply auto
  by  (simp add: p9 Cons_eq_append_conv)+
have p10: "¬([]⊢^a)"
  apply auto apply (subst (asm) LC.simps) by auto
have p15: "¬([^c,^b]⊢^c)"
  apply auto apply (subst (asm) LC.simps) apply auto
  by (simp add: p10 Cons_eq_append_conv)+
have p12: "¬([^a,^a→ ^c,^b]⊢^c)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p10 p15 by fastforce
have p16: "¬([^a,^b]⊢^c)"
  apply auto apply (subst (asm) LC.simps) apply auto
  by (simp_all add: Cons_eq_append_conv)+
have p17: "¬([^a]⊢^c)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
```

```
  using assms by auto
have p4: "¬([^a← ^b,^a→ ^c,^b]⊢^c)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p10 p15 p12 p16 p17 by fastforce+
have p23: "¬([^c]⊢^c← ^b)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  by (simp add: p15)
have p19: "¬([^a,^a→ ^c]⊢^c← ^b)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p10 p23 p12 by auto+
have p25: "¬([^a]⊢^c← ^b)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  by (simp add: p16)
show p2: "¬([^a← ^b,^a→ ^c]⊢^c← ^b)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p10 p23 p4 p25 p9 by fastforce+
qed
```

In addition, Listing C.1 in Appendix B is the formal proof of Theorem 16. Furthermore, Listing C.2 in Appendix B is the formal proof of Theorem 18.

5. Conclusion: Verified Formal Linguistics

5.1. Summary

We investigated Barker's and Plotkin's CPS transformations on Lambek calculus. Since the negative proofs by Lambek calculus were hard to verify manually due to the huge search space, we analyzed graphs of the paths of the proofs using Isabelle/HOL. We showed that Barker's CPS transformation was provable in Lambek calculus, whereas Plotkin's CPS transformation was not. Finally, we proposed a syntactic restriction on CPS transformations among the CPS transformations, which ensured provability in Lambek calculus.

We showed the grammar extraction algorithm to construct the set of grammar rules from the given treebank according to the pipeline shown in Figure 3.8, based on Algorithm 1. To obtain categorial grammar, we only extracted application rules for each branch. According to the algorithm's strategy, we non-deterministically chose only one grammar rule from the candidates. To obtain combinatory categorial grammar, we extracted composition rules in addition to categorial grammar. In addition to the extraction, we showed the grammar construction in the sequent calculus: we chained each grammar rule by a cut rule.

The head is a significant feature in syntactic theory and in the semantic parsing of Montague Semantics [22]. Since our algorithm could not retrieve the head information, we considered compensating for the lost information by employing the type-raising rule in categorial grammar. Furthermore, in combinatory categorial grammar, we utilized the extra combinators **Q** and **D**. By these combinators, we recursively converted categories to complicated functional ones to clarify the biting relation.

Then, we verified that our grammar was a legitimate extension of CG and CCG. This process was verified by the formal proof written in Isabelle/HOL. We showed these proofs for two lemmata and two theorems. Initially, we proved that a language generated by CG was in what by CCG. Next, we proved that a language generated by CCG was in what by the combinatory categorial grammar with **Q**-combinator and **D**-combinator.

We pointed out that we extended the set of sentences to supplement the lost head information, which was not problematic for natural language processing tasks because the order of words was still fixed, and we never employed cross-composition. Thus, our grammar remained in the same class as context-free grammar in the Chomsky hierarchy. However, the expansion of language would be problematic regarding the over-generation of sentences by our grammar. Therefore, we will add constraints to the parser and the generator in the future to prevent over-generation. The constraints are considered the polymorphic category or exclude the extraneous sentences.

Furthermore, we have shown the efficient transformation from a given syntactic tree to a left-branching one regarding patterns (1) – (6) in Section 3.1, although we excluded sentences with backward long references. The na"ive implementation [37] of the incremental parser produces backtracking many times. As a consequence, the complexity of the incremental parser to obtain the left-branching tree is exponential, whereas a non-incremental parser, in general, is $O(n^k)$. However, we could obtain a transforming algorithm with $O(n)$, and thus, the parsing complexity to obtain the left-branching tree is the same as $O(n^k)$ because $O(n^k) + O(n) = O(n^k)$.

We have not dealt with the issue of the long backward reference. Empirically, we observe the tree for such a reference through the Q combinator rules. However, the left-branching tree undergoes drastic changes, resulting in several variations from the original trees. Hence, we leave the transformation for these patterns as future work.

Furthermore, we pointed out that in non-incremental parsing, we do not consider the input order of words. We demonstrated that progressive parsing does not alter the meaning, and that the category of the partial sentence is not uniquely determined by progressive parsing. The uniqueness of the meaning was explained by lambda calculus. Standard CCG rules cannot achieve incremental parsing. Therefore, we introduced the previous research and its alternative, the unary rules D, Dx, Q, Qx. We showed that using these unary rules allows for the derivation of existing rules and is, therefore, a more powerful grammar. Next, we implemented a parser using logic programming. We not only generated a parsing tree but were also able to extract grammatical rules with the parser. Finally, we used these grammar rules to implement software for interactive parsing.

Parsing each word displays the transition of the parsing tree with incremental parsing. Additionally, based on the compositionality of CCG, we demonstrated the transition of the semantic term as well as the parsing tree. As the parser was based on the unification of logic programming, it performs an exhaustive search internally. This is significantly more inefficient than existing CCG parsers, so it will be necessary to measure efficiency by separating grammar extraction from parsing and eliminating non-deterministic programming.

Our contribution is three-fold: (a) We introduced the new grammar rule named type-restricted CPS transformation. (b) We developed the new grammar extraction system that maintains the head-dependency relation. (c) We demonstrated a constructive (partial) proof of the left-branching derivation in categorial grammar and Lambek calculus.

5.2. Conclusion

In Chapter 1, we presented the problems related to three topics: incremental reading, heads and dependencies, and continuations in natural language. This study was initially motivated by the human reading process observed in psycho-linguistics. The reading process is incremental, and the reader can understand the sentence by reading each word. Additionally, the reader can understand the sentence by starting from the head

of the sentence. Next, we explained the formal linguistics theory of the reading process in the syntactic theory, which is based on the head-dependency relation. The head is a significant feature in both syntactic theory and semantic parsing, and we provided a solution for the losing-a-head problem in grammar extraction. Finally, we uncovered the theoretical underpinnings of formal linguistics theory through the continuation of passing style transformation in the categorial grammar and Lambek calculus.

In Chapter 3, we presented linguistic problems. Firstly, we demonstrated the potential problem of incremental reading in the categorial grammar by proving the existence of left-branching derivation in Sectio 3.1. In other words, it is theoretically possible to parse a sentence from its head, resulting in a left-branching derivation. However, the algorithm for incremental parsing has not yet been developed.

In Sections 3.2 and 3.3, we proposed a solution to the incremental parsing problem. We developed new grammar rules for incremental parsing using the grammar extraction system and the CCG parser. Furthermore, we highlighted the ambiguity of incremental parsing due to the head-dependency relation and the losing-a-head problem in Sectio 3.4.

In Chapter 4, we presented the theoretical underpinnings of formal linguistics theory. We linked the grammaticality of a sentence to the provability of the implicational logic system. In Chapter 3, we augmented the generative power with new grammar rules for incremental parsing. However, this potentially leads to the over-generation of sentences. In Chapter 4, we restricted the grammar system by reducing the parsing algorithm to the proving system, while ensuring soundness and completeness.

In Section 4.1, we introduced the CPS transformation to the categorial grammar. The CPS transformation is a generalization of the type-raising rule to parse sentences incrementally, as shown in Chapter 3. In the later sections, we demonstrated the provability and decidability of the CPS transformation in the Lambek calculus.

Throughout this thesis, we also focused on the formal proof of linguistic theory and logic systems. Each theory and formalization is computationally expensive to prove manually, as shown in Section 4.2. Therefore, we implemented the proof system in Isabelle/HOL and generated each proof mechanically.

The generative power is a trade-off between the efficiency of the parsing algorithm and the verification cost. In the linguistics community, new grammar rules are proposed daily to precisely describe natural language. Similarly, in the natural language processing community, new grammar rules are proposed to skip redundant computation. However, new grammar rules are not always efficient, and the verification cost is not always cheap. Moreover, new grammar rules are not always sound and complete from a proof-theoretic viewpoint, meaning that they may cause over-generation of sentences.

In this thesis, we introduced a new discipline for grammar rules: the provability of the implicational logic system. Furthermore, we provided new tools to check the discipline of grammar rules using Isabelle/HOL. In other words, we proposed a perspective to restrict/reduce the generative power of grammar rules in our thesis.

Our primary focus in this thesis was English, where sentences are typically in a fixed order, similar to the Lambek Calculus. However, to apply our theory to other languages, we must consider the sentence order of that language. In Japanese, for

example, the sentence order is not always fixed and can differ from that of the Lambek Calculus. This often necessitates the introduction of powerful grammar rules, which can result in over-generation of sentences. Therefore, we require a robust proof system to check the grammar rules' discipline regarding the provability of the implicational logic system. We will continue to work on this problem using proof assistant systems to verify provability mechanically.

Bibliography

[1] Chris Barker. "Continuations in natural language". In: *CW* 4 (2004), pp. 1–11.

[2] Chris Barker and Chung-chieh Shan. *Continuations and natural language*. Vol. 53. Oxford Studies in Theoretical, 2014.

[3] Guy Barry and Martin Pickering. "Dependency and constituency in categorial grammar". In: *Studies in Categorial Grammar. Edinburgh Working Papers in Cognitive Science* 5 (1990).

[4] Daisuke Bekki and Kenichi Asai. "Representing covert movements by delimited continuations". In: *JSAI International Symposium on Artificial Intelligence*. Springer. 2009, pp. 161–180.

[5] Ann Bies et al. "Bracketing guidelines for Treebank II style Penn Treebank project". In: *University of Pennsylvania* 97 (1995), p. 100.

[6] Johan Bos. "Towards wide-coverage semantic interpretation". In: *Proceedings of Sixth International Workshop on Computational Semantics IWCS-6*. Vol. 4253. 2005.

[7] Noam Chomsky. *Aspects of the theory of syntax*. MIT press, 1965.

[8] Noam Chomsky et al. *Remarks on nominalization*. Linguistics Club, Indiana University, 1968.

[9] Olivier Danvy. "Back to direct style". In: *Science of Computer Programming* 22.3 (1994), pp. 183–195.

[10] Philippe De Groote. "Type raising, continuations, and classical logic". In: *Proceedings of the thirteenth Amsterdam Colloquium*. ILLC Amsterdam. 2001, pp. 97–101.

[11] Vera Demberg. "Incremental derivations in CCG". In: *Proceedings of the 11th International Workshop on Tree Adjoining Grammars and Related Formalisms*. 2012.

[12] Marc Dymetman and Pierre Isabelle. "Reversible logic grammars for machine translation". In: *Proceedings of the Second Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*. 1988.

[13] Lyn Frazier and Keith Rayner. "Making and correcting errors during sentence comprehension: Eye movements in the analysis of structurally ambiguous sentences". In: *Cognitive psychology* 14.2 (1982), pp. 178–210.

[14] Ahmed Hefny, Hany Hassan, and Mohamed Bahgat. "Incremental combinatory categorial grammar and its derivations". In: *International Conference on Intelligent Text Processing and Computational Linguistics*. 2011.

[15] J Roger Hindley and Jonathan P Seldin. *Lambda-calculus and Combinators, an Introduction*. Vol. 2. Cambridge University Press Cambridge, 2008.

[16] Joe Hurd. "First-order proof tactics in higher-order logic theorem provers". In: *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports* (2003), pp. 56–68.

[17] Jayant Krishnamurthy and Tom Mitchell. "Joint syntactic and semantic parsing with combinatory categorial grammar". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2014.

[18] Joachim Lambek. "The mathematics of sentence structure". In: *The American Mathematical Monthly* 65.3 (1958), pp. 154–170.

[19] Christopher D Manning et al. "The Stanford CoreNLP natural language processing toolkit". In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 2014, pp. 55–60.

[20] Mary Ann Marcinkiewicz. "Building a large annotated corpus of English: The Penn Treebank". In: *Using Large Corpora* 273 (1994).

[21] Pascual Martínez-Gómez et al. "ccg2lambda: A compositional semantics system". In: *Proceedings of ACL-2016 System Demonstrations*. 2016.

[22] Richard Montague. "The proper treatment of quantification in ordinary English". In: *Approaches to natural language*. Springer, 1973, pp. 221–242.

[23] Michael Moortgat. "Mixed composition and discontinuous dependencies". In: *Categorial Grammars and Natural Language Structures*. Springer, 1988, pp. 319–348.

[24] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[25] Fernando CN Pereira and David HD Warren. "Definite clause grammars for language analysis-a survey of the formalism and a comparison with augmented transition networks". In: *Artificial intelligence* 13.3 (1980), pp. 231–278.

[26] Gordon D. Plotkin. "Call-by-name, call-by-value and the λ-calculus". In: *Theoretical computer science* 1.2 (1975), pp. 125–159.

[27] John C Reynolds. "The discoveries of continuations". In: *Lisp and symbolic computation* 6.3 (1993), pp. 233–247.

[28] Beatrice Santorini. "Part-of-speech tagging guidelines for the Penn Treebank project (3rd revision, 2nd printing)". In: *Ms., Department of Linguistics, UPenn. Philadelphia, PA* (1990).

[29] Miloš Stanojević and Mark Steedman. "CCG parsing algorithm with incremental tree rotation". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 228–239.

[30] Miloš Stanojević and Mark Steedman. "Max-margin incremental CCG parsing". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2020.

[31] Mark Steedman. "Categorial grammar". In: *Lingua* 90.3 (1993), pp. 221–258.

[32] Mark Steedman. *The syntactic process*. MIT press, 2001.

[33] Leon Sterling and Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[34] Masaya Taniguchi. "Decidable Algorithm for Categorial Grammar with Type-raising". In: *4th The Proof Society Autumn School and Workshop*. 2022.

[35] Masaya Taniguchi. "Unprovabiliy of Continuation-Passing Style Transformation in Lambek Calculus". In: *European Summer School in Logic, Language and Information, Student Session* (2022).

[36] Masaya Taniguchi and Satoshi Tojo. "Generic Framework to Uncross Dependency". In: *25th International Symposium on Artificial Life and Robotics*. 2020.

[37] Masaya Taniguchi and Satoshi Tojo. "Incremental derivations with Q combinator in CCG". In: *Proceedings of Logic and Engineering of Natural Language Semantics 18*. This is not in the postproceedings. 2021.

[38] Masaya Taniguchi and Satoshi Tojo. "Interactive Grammar Extraction from a Treebank". In: *Proceedings of The 16th International Conference on Knowledge, Information and Creativity Support Systems*. 2021.

[39] Masaya Taniguchi and Satoshi Tojo. "Interactive Grammar Extraction from a Treebank". In: *Journal of Intelligent Informatics and Smart Technology* 8 (2022).

A. Simply-typed Lambda Calculus

We follow the definitions and proofs, which are introduced by Hindley [15].

Definition 19 (Types $\tau_{\rightarrow}$)**.** Let α be atomic types such as υ, ν, then the types $\tau_{\rightarrow}$ of the calculus $\lambda_{\rightarrow}$ are defined as;

$$\tau_{\rightarrow} ::= \tau_{\rightarrow} \rightarrow \tau_{\rightarrow} \mid \alpha \tag{A.1}$$

$$\alpha ::= \upsilon \mid \nu \mid \cdots \tag{A.2}$$

The parentheses are reserved to denote a priority of the constructor $\rightarrow$.

Definition 20 (Typed variables and constants)**.** Assume that there exists infinite sequences of untyped variables, then the typed variable is defined as;

- No untyped variable receives more than one type.
- Every type is attached to an infinite sequence of variables.

Let x be a variable and τ be a type of $\tau_{\rightarrow}$, then the typed variable is written as x^{τ}. Assume that there exists a finite or infinite sequence of atomic constants typed $\tau_{\rightarrow}$ besides the typed variables, then it is written as c^{τ} where c is an atomic constant and τ is a type of $\tau_{\rightarrow}$.

Definition 21 (Typed λ-terms)**.** Typed λ-terms are defined by the following conditions;

- All typed variables and atomic constants are typed λ-terms.
- Let τ and σ be types of $\tau_{\rightarrow}$ and x^{τ} be a typed variable and M^{σ} be a typed λ-term, then $(\lambda x^{\tau}.M^{\sigma})^{\tau\rightarrow\sigma}$ is a typed λ-term.
- Let τ and σ be types given by Definition 19 and $M^{\tau\rightarrow\sigma}$ and N^{τ} be typed λ-terms, then $(M^{\tau\rightarrow\sigma}N^{\tau})^{\sigma}$ is typed λ-term.

Note that we write $\upsilon\tau$ as the type σ of $(M^{\tau\rightarrow\sigma}N^{\tau})^{\sigma}$, then υ is $\tau \rightarrow \sigma$.

Definition 22 (Length of a typed λ-term)**.** The length of a typed λ-term M^{τ} (written as $L(M^{\tau})$) is the total number of occurrences of atoms in M^{τ}. Let x^{τ} be a typed variable and c^{τ} be a typed atomic constants, then the length $L(M^{\tau})$ is defined as;

$$L(x^{\tau}) = 1; \tag{A.3}$$

$$L(c^{\tau}) = 1; \tag{A.4}$$

$$L\left((S^{\sigma}N^{\upsilon})^{\sigma\upsilon}\right) = L(S^{\sigma}) + L(N^{\upsilon}) \qquad \text{if } \tau \equiv \sigma\upsilon; \tag{A.5}$$

$$L\left((\lambda x^{\sigma}.N^{\upsilon})^{\sigma\rightarrow\upsilon}\right) = 1 + L(N^{\upsilon}) \qquad \text{if } \tau \equiv \sigma \rightarrow \upsilon. \tag{A.6}$$

Definition 23 (Free and bound variables)**.** For typed λ-terms P^σ and Q^τ, the relation P^σ occurs in Q^τ (or P^σ is a subterm of Q^τ, or Q^τ contains P^σ) is defined by induction on Q^τ.

- P^σ occurs in P;
- if P^σ occurs in M^υ or in N^ϕ, then P^σ occurs in $\left(M^\upsilon N^\phi\right)^{\upsilon\phi}$;
- if P^σ occurs in M^υ, then P^σ occurs in $\left(\lambda x^\phi.M^\upsilon\right)^{\phi\to\upsilon}$
- if $P^\sigma \equiv x^\sigma$, then P^σ occurs in $(\lambda x^\sigma.M^\upsilon)^{\sigma\to\upsilon}$

For a particular occurrence of $\left(\lambda x^\phi.M^\upsilon\right)^{\phi\to\upsilon}$ in a term P^σ, the occurrence of M is called the *scope* of the occurrence of λx on the left. An occurrence of a variable x^ϕ in a term P^σ is called

- *bound* if it is in the scope of λx^ϕ in P^σ,
- *bound and binding* if and only if it is the x^ϕ in λx^ϕ,
- *free* otherwise, and we write it as $x^\phi \in \mathrm{FV}(P^\sigma)$.

Theorem 24 (Substitution)**.** *For any typed λ-term* M^τ, N^σ *and typed variable* x^σ*, defines* $[N^\sigma/x^\sigma]M^\tau$ *to be the result of substituting* N^σ *for every free occurrence of* x^σ *in* M^τ*, and changing bound variable to avoid clashes. The precise definition is by induction on* M*, as follows.*

$$[N^\sigma/x^\sigma]x^\sigma \equiv N^\sigma; \tag{A.7}$$
$$[N^\sigma/x^\sigma]a^\tau \equiv a^\tau; \tag{A.8}$$
$$[N^\sigma/x^\sigma](P^\tau Q^\upsilon)^{\tau\upsilon} \equiv ([N^\sigma/x^\sigma]P^\tau\,[N^\sigma/x^\sigma]Q^\upsilon)^{\tau\upsilon}; \tag{A.9}$$
$$[N^\sigma/x^\sigma](\lambda x^\sigma.P^\tau)^{\sigma\to\tau} \equiv (\lambda x^\sigma.P^\tau)^{\sigma\to\tau}; \tag{A.10}$$
$$[N^\sigma/x^\sigma](\lambda y^\tau.P^\upsilon)^{\tau\to\upsilon} \equiv (\lambda y^\tau.P^\upsilon)^{\tau\to\upsilon} \tag{A.11}$$
$$\text{if } x^\sigma \notin \mathrm{FV}(P^\upsilon); \tag{A.12}$$
$$[N^\sigma/x^\sigma](\lambda y^\tau.P^\upsilon)^{\tau\to\upsilon} \equiv (\lambda y^\tau.[N^\sigma/x^\sigma]P^\upsilon)^{\tau\to\upsilon} \tag{A.13}$$
$$\text{if } x^\sigma \in \mathrm{FV}(P^\upsilon) \text{ and } y^\tau \notin \mathrm{FV}(N^\sigma); \tag{A.14}$$
$$[N^\sigma/x^\sigma](\lambda y^\tau.P^\upsilon)^{\tau\to\upsilon} \equiv (\lambda z^\tau.[N^\sigma/x^\sigma][z^\tau/y^\tau]P^\upsilon)^{\tau\to\upsilon} \tag{A.15}$$
$$\text{if } x^\sigma \in \mathrm{FV}(P^\upsilon) \text{ and } y^\tau \in \mathrm{FV}(N^\sigma). \tag{A.16}$$

where $x \not\equiv y$ *and* z *is chosen to be the first variable* $z \in \mathrm{FV}\left((N^\upsilon P^\tau)^{\upsilon\tau}\right)$*. Note that we do not define* $[N^\sigma/x^\tau]M^\upsilon$ *when* $\sigma \not\equiv \tau$*.*

We prove types of the typed λ-terms are the same before the substitutions. We use induction on the length of the typed λ-terms.

(A.7),(A.8) Let the length of the typed λ-term be 1, then the term is a typed variable or atomic constant. If the typed λ-term is the same variable, it is substituted by other typed λ-term that have the same type. Otherwise, it is not substituted by the other typed λ-term. Thus the typed λ-terms are the same before the substitutions.

(A.9) Let $L(P^\tau Q^\upsilon) = m + n$, and every type of typed λ-term that length is less than $n + m$ is the same as one before the substitution, then the types of substituted typed λ-terms $[N^\sigma/x^\sigma]P^\tau$ and $[N^\sigma/x^\sigma]Q^\upsilon$ have the same type of P^τ and Q^τ, where $L(P^\tau) = m < m+n$ and $L(Q^\upsilon) = n < m+n$. Thus, the type of $[N^\sigma/x^\sigma](P^\tau Q^\upsilon)^{\tau\upsilon} \equiv ([N^\sigma/x^\sigma]P^\tau\,[N^\sigma/x^\sigma]Q^\upsilon)^{\tau\upsilon}$ is $\tau\upsilon$, which is the same as one before the substitutions.

(A.10),(A.12) Since there is no substitution, the type of the typed λ-term is the same.

(A.14) Let $L\left((\lambda y^\tau.P^\upsilon)^{\tau\to\upsilon}\right) = n + 1$, and every type of typed λ-term that length is less than $n+1$ is the same as one before the substitution, then the types of substituted typed λ-term $[N^\sigma/x^\sigma]P^\upsilon$ have the same type of P^υ, where $L(P^\upsilon) = n < n+1$. Thus, the type of $[N^\sigma/x^\sigma](\lambda y^\tau.P^\upsilon)^{\tau\to\upsilon} \equiv (\lambda y^\tau.[N^\sigma/x^\sigma]P^\upsilon)^{\tau\to\upsilon}$ is $\tau \to \upsilon$, which is the same as one before the substitutions.

(A.16) Let $L\left((\lambda y^\tau.P^\upsilon)^{\tau\to\upsilon}\right) = n + 1$, and every type of typed λ-term that length is less than $n+1$ is the same as one before the substitution, then the types of substituted typed λ-term $[z^\tau/y^\tau]P^\upsilon$ have the same type of P^υ, where $L(P^\upsilon) = n < n+1$. Since $L([z^\tau/y^\tau]P^\upsilon) = n < n+1$, the type of the typed λ-term $[N^\sigma/x^\sigma][z^\tau/y^\tau]P^\upsilon$ is also the same as one before substitutions. Thus, the type of $[N^\sigma/x^\sigma](\lambda y^\tau.P^\upsilon)^{\tau\to\upsilon} \equiv (\lambda z^\tau.[N^\sigma/x^\sigma][z^\tau/y^\tau]P^\upsilon)^{\tau\to\upsilon}$ is $\tau \to \upsilon$, which is the same as one before the substitutions.

By the induction hypothesis, every substituted typed λ-term is the same as before the substitutions.

Theorem 25 (α-conversion)**.** *Let a term P^τ contain an occurence of $\lambda x^\sigma.M^\upsilon$, $(y^\sigma \in M^\upsilon)^{\sigma\to\upsilon}$, and let $y^\upsilon \notin FV(M^\upsilon)$. The act of replacing this $(\lambda x^\sigma.M^\upsilon)^{\sigma\to\upsilon}$ by $(\lambda y^\sigma.[y^\sigma/x^\sigma]M^\upsilon)^{\sigma\to\upsilon}$ is called a change of bound variable or an α-conversion in P^τ. If and only if P^τ can be changed to Q^τ by a finite (perhaps empty) series of changes of bound variables, we shall say $P^\tau \equiv_\alpha Q^\tau$.*

We have already proved that the type of the typed λ-term is the same as one before α-conversion in Theorem 24.

Theorem 26 (β-reduction)**.** *The formal theory of simply typed β-equality will be called $\lambda\beta^\to$. It has equations $M^\sigma = N^\sigma$ as its formulas, and the following as its axiom schemes and rules:*

$$\frac{}{\Gamma \vdash \left((\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau}N^{\sigma}\right)^{\tau} = [N^{\sigma}/x^{\sigma}]\,M^{\tau}} \text{ (A.0.0.a)}$$

$$\frac{}{\Gamma \vdash M^{\sigma} = M^{\sigma}} \text{ (A.0.0.b)}$$

$$\frac{y^{\sigma} \notin \mathrm{FV}(M^{\tau})}{\Gamma \vdash (\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau} = (\lambda y^{\sigma}.[y^{\sigma}/x^{\sigma}]\,M^{\tau})^{\sigma\to\tau}} \text{ (A.0.0.c)}$$

$$\frac{\Gamma \vdash M^{\sigma} = N^{\sigma}}{\Gamma \vdash (P^{\sigma\to\tau}M^{\sigma})^{\tau} = (P^{\sigma\to\tau}N^{\sigma})^{\tau}} \text{ (A.0.0.d)}$$

$$\frac{\Gamma \vdash M^{\sigma\to\tau} = N^{\sigma\to\tau}}{\Gamma \vdash (M^{\sigma\to\tau}P^{\sigma})^{\tau} = (N^{\sigma\to\tau}P^{\sigma})^{\tau}} \text{ (A.0.0.e)}$$

$$\frac{\Gamma \vdash M^{\tau} = N^{\tau}}{\Gamma \vdash (\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau} = (\lambda x^{\sigma}.N^{\tau})^{\sigma\to\tau}} \text{ (A.0.0.f)}$$

$$\frac{\Gamma \vdash M^{\sigma} = N^{\sigma} \qquad \Gamma \vdash N^{\sigma} = P^{\sigma}}{\Gamma \vdash M^{\sigma} = P^{\sigma}} \text{ (A.0.0.g)}$$

$$\frac{\Gamma \vdash M^{\sigma} = N^{\sigma}}{\Gamma \vdash N^{\sigma} = M^{\sigma}} \text{ (A.0.0.h)}$$

The formal theory of simply typed β-reduciton will be called $\lambda\beta^{\to}$ like that of equality.

$$\frac{}{\Delta \vdash \left((\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau}N^{\sigma}\right)^{\tau} \triangleright [N^{\sigma}/x^{\sigma}]\,M^{\tau}} \text{ (A.0.0.i)}$$

$$\frac{}{\Delta \vdash M^{\sigma} \triangleright M^{\sigma}} \text{ (A.0.0.j)}$$

$$\frac{y^{\sigma} \notin \mathrm{FV}(M^{\tau})}{\Delta \vdash (\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau} \triangleright (\lambda y^{\sigma}.[y^{\sigma}/x^{\sigma}]\,M^{\tau})^{\sigma\to\tau}} \text{ (A.0.0.k)}$$

$$\frac{\Delta \vdash M^{\sigma} \triangleright N^{\sigma}}{\Delta \vdash (P^{\sigma\to\tau}M^{\sigma})^{\tau} \triangleright (P^{\sigma\to\tau}N^{\sigma})^{\tau}} \text{ (A.0.0.l)}$$

$$\frac{\Delta \vdash M^{\sigma\to\tau} \triangleright N^{\sigma\to\tau}}{\Delta \vdash (M^{\sigma\to\tau}P^{\sigma})^{\tau} \triangleright (N^{\sigma\to\tau}P^{\sigma})^{\tau}} \text{ (A.0.0.m)}$$

$$\frac{\Delta \vdash M^{\tau} \triangleright N^{\tau}}{\Delta \vdash (\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau} \triangleright (\lambda x^{\sigma}.N^{\tau})^{\sigma\to\tau}} \text{ (A.0.0.n)}$$

$$\frac{\Delta \vdash M^{\sigma} \triangleright N^{\sigma} \qquad \Delta \vdash N^{\sigma} \triangleright P^{\sigma}}{\Delta \vdash M^{\sigma} \triangleright P^{\sigma}} \text{ (A.0.0.o)}$$

For provability in these theories, we shall write $\lambda\beta^{\to}$ instead of Γ and Δ.

$$\lambda\beta^{\to} \vdash M^{\sigma} = N^{\sigma}, \qquad \lambda\beta^{\to} \vdash M^{\sigma} \triangleright N^{\sigma} \tag{A.17}$$

We prove that the type is the same on simply typed β-equality and simply typed β-reduction.

(A.0.0.c), (A.0.0.k) For any typed λ-term M^τ, the type is the same on the α-conversion because we have already proved in Theorem 25.

(A.0.0.d), (A.0.0.l) For any typed λ-terms M^σ, N^σ, and $P^{\sigma\to\tau}$, the types of $(P^{\sigma\to\tau}M^\sigma)^\tau$ and $(P^{\sigma\to\tau}N^\sigma)^\tau$ are τ by Definition 21.

(A.0.0.e), (A.0.0.m) For any typed λ-terms $M^{\sigma\to\tau}$, $N^{\sigma\to\tau}$, and P^σ, the types of $(M^{\sigma\to\tau}P^\sigma)^\tau$ and $(N^{\sigma\to\tau}P^\sigma)^\tau$ are τ by Definition 21.

(A.0.0.f), (A.0.0.n) For any typed λ-terms M^τ and N^τ, the types of $(\lambda x^\sigma.M^\tau)^{\sigma\to\tau}$ and $(\lambda x^\sigma.N^\tau)^{\sigma\to\tau}$ are $\sigma\to\tau$ by Definition 21.

(A.0.0.g), (A.0.0.o) If the types of M^σ and N^σ are the same and the types of N^σ and P^σ are same, then the types of M^σ and N^σ are also same.

(A.0.0.h) If the type of M^σ is the same as N^σ, then the type of N^σ is the same as M^σ.

Therefore, the type is the same on all rules defined above.

Theorem 27 (η-reduction)**.** *An η-redex is any typed λ-term $\left(\lambda x^\sigma.(M^{\sigma\to\tau}x^\sigma)^\tau\right)^{\sigma\to\tau}$ with $x^\sigma\notin \mathrm{FV}(M^{\sigma\to\tau})$. Its contractum is $M^{\sigma\to\tau}$. The phrases 'P η-contracts to Q' and 'P η-reduces to Q' are defiend by replacing η-redexes by their contracta, like β-reduces in Definition 26.*

$$P \rhd_{1\eta} Q, \qquad P \rhd_\eta Q \tag{A.18}$$

We prove that the type is the same on $\rhd_{1\eta}$ and $\rhd_\eta$. The type of $M^{\sigma\to\tau}$ is $\sigma\to\tau$ that is the same as $\left(\lambda x^\sigma.(M^{\sigma\to\tau}x^\sigma)^\tau\right)^{\sigma\to\tau}$ by Definition 21.

Definition 28 (Simply typed $\beta\eta$)**.** The equality-theory $\lambda\beta\eta^\to$ is defiend by adding to Theorem 26 $\lambda\beta^\to$ the following axiom-scheme:

$$\left(\lambda x^\sigma.(M^{\sigma\to\tau}x^\sigma)^\tau\right)^{\sigma\to\tau} = M^{\sigma\to\tau} \text{ if } x^\sigma\notin \mathrm{FV}(M^{\sigma\to\tau})$$

The reduction-theory $\lambda\beta\eta^\to$ is defiend by adding Theorem 27 to Theorem 26 $\lambda\beta^\to$.

B. Formal Proof of Classical Logic in Hilbert System

We show the propositional logic in Isabelle/HOL and prove the soundness of the Hilbert system. In addition, we show the completeness theorem of this system. We initially define the syntax of the propositional logic in Isabelle/HOL. The system consists of variables, constants, and connectives. Its syntax is recursively defined as follows:

$$\varphi ::= x_i \mid \bot \mid \varphi \supset \varphi \qquad (i \in \mathbb{N}) \tag{B.1}$$

where x_i is a variable, $\bot$ is a constant, and $\supset$ is a connective. Then, the syntax of the propositional logic in Isabelle/ HOL is defined as follows:

Listing B.1: Formula

```
1 datatype formula = Var nat | Imp formula formula | Bot
```

Next, we define the negation of a formula with the implication and the bottom as follows. It means that the negation of a formula is the implication from the formula to the bottom.

Listing B.2: Negation

```
1 definition Not :: "formula ⇒ formula" where
2   "Not x = Imp x Bot"
```

In the same way, we define the disjunction of two formulas as follows: It means that the disjunction of two formulas x, y is the implication from the negation of formula $\neg x$ to another formula y.

Listing B.3: Disjunction

```
1 definition Or :: "formula ⇒ formula ⇒ formula" where
2   "Or x y = Imp (Not x) y"
```

Here, we define the conjunction of two formulas as follows: It means that the conjunction of two formulas x, y is defined by the de-Morgan's law $x \wedge y \iff \neg(\neg x \vee \neg y)$.

Listing B.4: Conjunction

```
definition And :: "formula ⇒ formula ⇒ formula" where
  "And x y = Not (Or (Not x) (Not y))"
```

In addition, we define a new constant $\top$ as the negation of the bottom.

Listing B.5: Top

```
definition Top :: "formula" where
  "Top = Not Bot"
```

In the above, we define the basic syntax of the propositional logic in Isabelle/HOL. Next, we define the semantics of the propositional logic. We assume that the valuation function v is given as the function from natural numbers to boolean values because the variables are indexed by natural numbers. Then, the valuation of the formula is defined as follows. In mathematical notation, it is defined as follows:

$$V(\varphi) = \begin{cases} v(i) & \text{if } \varphi = x_i \\ \mathit{false} & \text{if } \varphi = \bot \\ V(\varphi_1) \longrightarrow V(\varphi_2) & \text{if } \varphi = \varphi_1 \supset \varphi_2 \end{cases} \tag{B.2}$$

Note that the $\longrightarrow$ is a meta-logical implication defined by the truth table. In Isabelle/HOL, it is defined as follows:

Listing B.6: Valuation function

```
fun val :: "(nat ⇒ bool) ⇒ formula ⇒ bool" where
  "val _ Bot = False"
| "val v (Var x) = v x"
| "val v (Imp x y) = ((val v x) ⟶ (val v y))"
```

In this sense, we define the validity of the formula. The validity of the formula φ on the valuation v that satisfies all the formulas in the context Γ is defined as follows:

Listing B.7: Validity of the formula

```
definition valid :: "formula set ⇒ formula ⇒ bool" where
  "valid Γ φ = (∀v . (∀ψ . ψ ∈ Γ ⟶ val v ψ) ⟶ val v φ)"
```

From the above definition, we can prove the semantic deduction theorem: $\Gamma, \psi \vDash \varphi \implies \Gamma \vDash \psi \supset \varphi$. This theorem is proved by the definition of validity in Isabelle/HOL as follows:

Listing B.8: Semantic deduction theorem

```
theorem valid_deduction :
  assumes "valid (Γ ∪ {ψ}) φ"
  shows "valid Γ (Imp ψ φ)"
  using assms valid_def by auto
```

We define the axioms of the Hilbert system $\mathsf{E}, \mathsf{C}, \mathsf{D}$ and modus ponens MP.

$$\begin{aligned} \mathsf{E}: &\quad \alpha \supset (\beta \supset \alpha) \\ \mathsf{C}: &\quad (\neg\alpha \supset \neg\beta) \supset (\beta \supset \alpha) \\ \mathsf{D}: &\quad (\alpha \supset (\beta \supset \gamma)) \supset ((\alpha \supset \beta) \supset (\alpha \supset \gamma)) \\ \mathsf{MP}: &\quad \alpha \text{ and } \alpha \supset \beta \text{ then } \beta \end{aligned}$$

The validity of the Hilbert system is proved simply by the calculation of valuation function as follows:

$$\begin{aligned} &V(\alpha \supset (\beta \supset \alpha)) \\ &\quad = V(\alpha) \longrightarrow (V(\beta) \longrightarrow V(\alpha)) = \mathit{true} \\ &V((\neg\alpha \supset \neg\beta) \supset (\beta \supset \alpha)) \\ &\quad = ((\alpha \longrightarrow \bot) \longrightarrow (\beta \longrightarrow \bot)) \longrightarrow (\beta \longrightarrow \alpha) = \mathit{true} \\ &V((\alpha \supset (\beta \supset \gamma)) \supset ((\alpha \supset \beta) \supset (\alpha \supset \gamma))) \\ &\quad = ((\alpha \longrightarrow (\beta \longrightarrow \gamma)) \longrightarrow ((\alpha \longrightarrow \beta) \longrightarrow (\alpha \longrightarrow \gamma))) = \mathit{true} \\ &V(\alpha) \text{ and } V(\alpha \supset \beta) = V(\alpha) \longrightarrow V(\beta) \text{ then } V(\beta) \text{ is } \mathit{true} \end{aligned}$$

Listing B.9: Validity of the Hilbert system

```
lemma validity_of_E: "valid {} (Imp x (Imp y x))"
  using valid_def by auto

lemma validity_of_C: "valid {} (Imp (Imp (Not x) (Not y)) (Imp y x))"
  using Not_def valid_def by auto

lemma validity_of_D:
  "valid {} (Imp (Imp x (Imp y z)) (Imp (Imp x y) (Imp x z)))"
  using valid_def  by auto

lemma validity_of_MP:
  assumes "valid {} x"
      and "valid {} (Imp x y)"
    shows "valid {} y"
  using assms valid_def by auto
```

In the above, we prove the validity of the Hilbert system $\mathsf{E}, \mathsf{C}, \mathsf{D}$ and modus ponens MP by the definition of validity in Isabelle/HOL. Here, we define the proof system of the Hilbert system $\mathsf{E}, \mathsf{C}, \mathsf{D}$ and modus ponens MP as follows:

$$\vdash \alpha \supset (\beta \supset \beta) \tag{B.3}$$
$$\vdash (\neg\alpha \supset \neg\beta) \supset (\beta \supset \alpha) \tag{B.4}$$
$$\vdash (\alpha \supset (\beta \supset \gamma)) \supset ((\alpha \supset \beta) \supset (\alpha \supset \gamma)) \tag{B.5}$$
$$\Gamma \vdash \alpha \text{ and } \Delta \vdash \alpha \supset \beta \text{ then } \Gamma, \Delta \vdash \beta \tag{B.6}$$

The above inductive definition is defined in Isabelle/HOL as follows:

Listing B.10: Proof system of the Hilbert system

```
inductive provable :: "nat ⇒ formula set ⇒ formula ⇒ bool" where
  E: "provable 0 {} (Imp x (Imp y x))"
| D: "provable 0 {} (Imp (Imp x (Imp y z)) (Imp (Imp x y) (Imp x z)))"
| C: "provable 0 {} (Imp (Imp (Not x) (Not y)) (Imp y x))"
| A: "provable 0 {x} x"
| MP: "⟦provable n Γ x; provable m Δ (Imp x y)⟧
      ⟹ provable (n + m + 1) (Γ ∪ Δ) y"
```

Note that we define the provability of the formula with the number of the proof steps; the axiom is proved by 0 steps, the modus ponens is proved by $n + m + 1$ steps, where n and m are the number of the proof steps of the premises.

The soundness theorem $\vdash \varphi \implies \models \varphi$ of the Hilbert system is mechanically proved by the above lemmas with respect to the validity of all axioms:

Listing B.11: Soundness theorem

```
theorem soundness: "⋀x. provable n {} x ⟹ valid {} x"
  apply (induct n rule: nat_less_induct)
  using provable.simps validity_of_C validity_of_D
        validity_of_E validity_of_MP
  by (smt (verit, best) Suc_eq_plus1 Un_empty
      insert_not_empty less_add_Suc1 less_add_Suc2)
```

Here, we define the compactness of this system; if the formula φ is provable in the given context Γ, then Γ is a fnite set. This theorem is proved by the induction of the proof steps as follows:

Listing B.12: Compactness theorem

```
theorem compactness:
  assumes "provable n Γ φ"
```

```
  shows "finite Γ"
proof -
  have "⋀Γ φ. provable n Γ φ ⟹ finite Γ"
    apply (induct n rule: nat_less_induct)
    apply (erule provable.cases)
    apply auto
    apply (meson less_add_Suc1)
    using less_add_Suc2 by blast
  thus ?thesis
    using assms by auto
qed
```

Here, `erule provable.cases` is a tactic to apply the rule of the inductive definition of the `provable`. Further `meson less_add_Suc1` is a tactic to rewrite a subgloal by the hint `less_add_Suc1`. Moreover, `blast` is a tactic to search a proof machanically. We prove the weakening theorem in a finite context: if $\Gamma \vdash \varphi$ and $\Gamma \subseteq \Delta$, then $\Delta \vdash \varphi$. This theorem is defined in Isabelle/HOL as follows:

Listing B.13: Weakening theorem

```
lemma weakening_prems:
  assumes "finite Δ"
      and "Γ ⊆ Δ"
      and "∃n. provable n Γ φ"
    shows "∃n. provable n Δ φ"
```

This theorem is proved by the induction of the number of formulas in the context Δ as follows:

Listing B.14: Proof of weakening theorem

```
proof -
  have "∀Γ ⊆ Δ. ∀φ. (∃n. provable n Γ φ) ⟶ (∃n. provable n Δ φ)"
    apply (subgoal_tac "finite Δ")
    apply (induct rule: finite_induct)
    using assms apply simp_all
    by (metis (no_types, opaque_lifting)
        A E MP empty_subsetI insert_is_Un subset_Un_eq)
  thus ?thesis using assms by blast
qed
```

As a corollary, we prove the weakening theorem with respect to the implication: if $\Gamma \vdash \varphi$ then $\Gamma \vdash \psi \supset \varphi$. This is proved by the above theorem and the semantic deduction theorem.

Listing B.15: Weakening theorem with respect to the implication

```
lemma weakening_imp:
  assumes "∃n. provable n Γ φ"
  shows "∃n. provable n Γ (Imp ψ φ)"
  by (metis E MP assms sup_bot_right)
```

Here, we prove the deduction theorem in the Hilbert system: if $\Gamma, \varphi \vdash \psi$ then $\Gamma \vdash \varphi \supset \psi$. In Isabelle/ HOL, the deduction theorem is defined as follows:

Listing B.16: Deduction theorem

```
theorem provable_deduction :
   "⋀ Γ ψ φ. provable m (Γ ∪ {φ}) ψ ⟹ ∃n. provable n Γ (Imp φ ψ)"
```

The proof of the deduction theorem is reduced into a simple problem by the induction of the number of the proof steps of the premises as follows:

Listing B.17: Proof of deduction theorem

```
  apply (induct m rule: nat_less_induct)
  apply (erule provable.cases)
  apply auto
  apply (metis A D E MP subset_singleton_iff sup_bot_right)
```

We prove the induction step. First, we assume the part of proof for ψ and $\psi \supset \delta$ with given contexts Δ and Σ.

Listing B.18: Proof of deduction theorem (cont.)

```
proof -
  fix φ ψ δ :: formula and Γ Δ Σ :: "formula set" and a b :: nat
  assume a1: "provable a Δ ψ" and a2: "provable b Σ (Imp ψ δ)"
```

Further, we assume that the $\Gamma \cup \{\varphi\}$ is divided into Δ and Σ. By the compactness, we can assume that Γ is finite. Moreover, $\Delta \setminus \{\varphi\}$ and $\Sigma \setminus \{\varphi\}$ are also finite.

Listing B.19: Proof of deduction theorem (cont.)

```
  assume a3: "insert φ Γ = Δ ∪ Σ"
  hence a4: "finite Γ" and a31: "Δ - {φ} ⊆ Γ" and a32: "Σ - {φ} ⊆ Γ"
    apply (metis a1 a2 compactness finite_UnI finite_insert)
    using a3 by fastforce+
```

Then, the induction hypothesis with respect to the number of deduction steps of the premises is defined as follows.

Listing B.20: Proof of deduction theorem (cont.)

```
1  assume hyp: "∀n<Suc (a + b). ∀Γ φ ψ.
2               provable n (insert ψ Γ) φ ⟶ (∃n. provable n Γ (Imp ψ φ))"
```

We split the hypothesis into the following cases to prove the induction step.

Listing B.21: Proof of deduction theorem (cont.)

```
1  consider (a) "φ ∈ Δ" and "φ ∈ Σ"
2         | (b) "φ ∉ Δ" and "φ ∈ Σ"
3         | (c) "φ ∈ Δ" and "φ ∉ Σ"
4    using a3 by auto
5  thus "∃n. provable n Γ (Imp φ δ)"
```

In case (a), we prove the induction step $\varphi \in \Delta \Longrightarrow \varphi \in \Sigma \Longrightarrow$ `∃n. provable n Γ (Imp φ δ)` by the following steps:
We have an axiom `∃n. provable n Γ (Imp φ ψ)` .

Listing B.22: Proof of deduction theorem, case (a)

```
1  proof cases
2    case a
3    have "∃n. provable n Γ (Imp φ ψ)"
4      using a(1) a1 a31 a4 weakening_prems hyp
5      by (metis insert_Diff_single insert_absorb less_iff_Suc_add)
```

Further, we also have an axiom `∃n. provable n Γ (Imp φ (Imp ψ δ))`

Listing B.23: Proof of deduction theorem, case (a) (cont.)

```
1    moreover have "∃n. provable n Γ (Imp φ (Imp ψ δ))"
2      using a(2) a2 a32 a4 weakening_prems hyp
3      by (metis insert_Diff less_add_Suc2)
```

Finally, we show the `?thesis = ∃n. provable n Γ (Imp φ δ)`.

Listing B.24: Proof of deduction theorem, case (a) (cont.)

```
    ultimately show ?thesis using D MP
      by (metis Un_absorb sup_bot_right)
```

In cases (b) and (c), it is proved in the same way as in case (a).

Listing B.25: Proof of deduction theorem, case (b)

```
  next
    case b
    have "∃n. provable n Γ (Imp φ ψ)"
      using b(1) a1 a32 a4 weakening_imp weakening_prems
      by (metis Diff_empty Diff_insert0 a31)
    moreover have "∃n. provable n Γ (Imp φ (Imp ψ δ))"
      using b(2) a2 a32 a4 weakening_prems hyp
      by (metis insert_Diff_single insert_absorb less_add_Suc2)
    ultimately show ?thesis using D MP
      by (metis Un_absorb sup_bot_right)
```

Listing B.26: Proof of deduction theorem, case (c)

```
  next
    case c
    have "∃n. provable n Γ (Imp φ ψ)"
      using c(1) a1 a31 a4 weakening_prems hyp
      by (metis insert_Diff_single insert_absorb less_iff_Suc_add)
    moreover have "∃n. provable n Γ (Imp φ (Imp ψ δ))"
      using c(2) a2 a32 a4 weakening_imp weakening_prems hyp
      by (metis Diff_empty Diff_insert0)
    ultimately show ?thesis using D MP
      by (metis Un_absorb sup_bot_right)
  qed
qed
```

Finally, we show the completeness of the system using the above deduction theorem. We consider the constructive proof with respect to the variables in the formula. The following function lists the variables in the formula.

Listing B.27: Variables in the formula

```
fun var :: "formula ⇒ formula set" where
  "var (Var n) = {Var n}"
| "var (Imp x y) = ((var x) ∪ (var y))"
| "var Bot = {}"
```

Then, by the definition of the formula, the number of variables in the formula is finite. The finiteness is proved by the following lemma in Isabelle/ HOL.

Listing B.28: Finiteness of variables in the formula

```
lemma compactness_var : "finite (var p)"
  apply (induct p)
  by simp_all
```

We next define the sign function that returns the negated formula if the formula is falsely in the given valuation function.

Listing B.29: Sign function of the formula

```
definition σ :: "(nat ⇒ bool) ⇒ formula ⇒ formula" where
  "σ v p = (if val v p then p else (Not p))"
```

Here, we show the provability of the sign function:

$$\forall \nu. \forall p. \quad \sigma(\nu, x_1), \sigma(\nu, x_2), \cdots, \sigma(\nu, x_i) \vdash \sigma(\nu, p)$$

where $x_1, x_2, \cdots, x_i$ are all the variables in the formula p.

This is defined as follows in Isabelle/HOL.

Listing B.30: Provability of the sign function

```
lemma "⋀v. ∃n. provable n ((σ v) ‘ (var p)) (σ v p)"
```

The proof is done by the induction of a structure of the formula. The simplest case is the case of the variable. Then, it is proved as follows.

Listing B.31: Proof of the provability of the sign function, variable

```
proof (induct p)
  case (Var x)
  thus ?case using A by auto
```

The next case is the constant $\bot$. Note $\sigma(\nu, \bot) = \top$. Then, $\vdash \top$, that is, $\vdash \bot \supset \bot$ is proved by the deduction theorem.

Listing B.32: Proof of the provability of the sign function, bottom

```
next
  case Bot
  thus ?case by
   (metis A Not_def Un_insert_right σ_def
          image_empty provable_deduction
          sup_bot_right val.simps(1) var.simps(3))
```

The last case is the implication formed $p \supset q$. Then, the cases are considered as follows.

Listing B.33: Proof of the provability of the sign function, implication

```
next
  case (Imp p q)
  consider
    (a) "val v p = True" and "val v q = True" |
    (b) "val v p = True" and "val v q = False" |
    (c) "val v p = False" and "val v q = True" |
    (d) "val v p = False" and "val v q = False"
    by auto
  thus ?case
```

We here show case (a) as the typical case. Because of the induction hypothesis, we have the theorem for q.

$$\forall v. \quad \sigma(v, y_1), \sigma(v, y_2), \cdots, \sigma(v, y_j) \vdash \sigma(v, q)$$

where $y_1, y_2, \cdots, y_i$ are all the variables in the formula q.

Listing B.34: Proof of the provability of the sign function, case (a)

```
  proof cases
    case a
    have "∃n. provable n (σ v ‘ var q) q"
    proof -
      have "∃n. provable n (σ v ‘ var q) (σ v q)"
        by (simp add: Imp.hyps(2))
      thus ?thesis by (simp add: Imp.hyps σ_def a(2))
    qed
```

Line 1 in Listing B.35 is proved by the weakening theorem. Line 3 is proved because $p \supset q$ is always true by the assumptions of this case. Line 5 is proved by the definition of `var` function. Finally, we show the theorem for $p \subset q$ in case (a) as follows.

Listing B.35: Proof of the provability of the sign function, case (a) (cont.)

```
    hence "∃n. provable n
           ((σ v ‘ var p)∪(σ v ‘ var q)) (Imp p q)"
      by (metis E MP sup_bot_right)
    moreover have "σ v (Imp p q) = (Imp p q)"
      by (simp add: σ_def a(2))
    moreover have "σ v ‘ var (Imp p q)
                  = (σ v ‘ var p)∪(σ v ‘ var q)"
      by (simp add: image_Un)
    ultimately show ?thesis
      by (metis Imp.hyps(1) MP sup.idem weakening_imp)
```

The other cases (b), (c), and (d) are proved similarly as follows.

Listing B.36: Proof of the provability of the sign function, case (b)

```
  next
    case b
    have "∃n. provable n (σ v ‘ var p) p"
      by (metis Imp.hyps(1) σ_def b(1))
    moreover have "∃n. provable n (σ v ‘ var q) (Not q)"
      by (metis Imp.hyps(2) σ_def b(2))
    ultimately have
      "∃n. provable n
       ((σ v ‘ var p)∪(σ v ‘ var q)) (Not (Imp p q))"
      using negative_imp by blast
    moreover have "σ v (Imp p q) = (Not (Imp p q))"
      by (simp add: Not_def σ_def b(1) b(2))
    moreover have "σ v ‘ var (Imp p q)
                  = (σ v ‘ var p)∪(σ v ‘ var q)"
      by (simp add: image_Un)
    ultimately show ?thesis by auto
```

Listing B.37: Proof of the provability of the sign function, case (c)

```
  next
    case c
    have "∃n. provable n (σ v ‘ var p) (Not p)"
      by (metis Imp.hyps(1) σ_def c(1))
    hence "∃n. provable n (σ v ‘ var p) (Imp p q)"
      using positive_imp by blast
    moreover have "finite (σ v ‘ var q)"
      using compactness_var by blast
    ultimately have
      "∃n. provable n
```

```
      ((σ v ‘ var p)∪(σ v ‘ var q)) (Imp p q)"
    by (metis weakening_prems compactness finite_UnI
              sup.cobounded2 sup_commute)
  moreover have "σ v (Imp p q) = (Imp p q)"
    by (simp add: σ_def c(1))
  moreover have "σ v ‘ var (Imp p q)
              = (σ v ‘ var p)∪(σ v ‘ var q)"
    by (simp add: image_Un)
  ultimately show ?thesis by auto
```

Listing B.38: Proof of the provability of the sign function, case (d)

```
next
  case d
  have "∃n. provable n (σ v ‘ var p) (Not p)"
    by (metis Imp.hyps(1) σ_def d(1))
  hence "∃n. provable n (σ v ‘ var p) (Imp p q)"
    using positive_imp by blast
  moreover have "finite (σ v ‘ var q)"
    using compactness_var by blast
  ultimately have
    "∃n. provable n ((σ v ‘ var p)∪(σ v ‘ var q)) (Imp p q)"
    by (metis weakening_prems compactness finite_UnI
              sup.cobounded2 sup_commute)
  moreover have "σ v (Imp p q) = (Imp p q)"
    by (simp add: σ_def d(1))
  moreover have "σ v ‘ var (Imp p q) = (σ v ‘ var p)∪(σ v ‘ var q)"
    by (simp add: image_Un)
  ultimately show ?thesis by auto
qed
qed
```

Here, we proof two lemmata $(x \supset y) \supset (\neg y \supset \neg x)$ and $(\neg x \supset y) \supset ((x \supset y) \supset y)$. They are proved in Listing B.39 and B.40.

Listing B.39: Proof of dual of C

```
lemma C_dual:
  "∃n. provable n {} (Imp (Imp x y) (Imp (Not y) (Not x)))"
proof -
  have
  "∃n. provable n {}
    (Imp (Imp (Not (Not x)) (Not (Not y))) (Imp (Not y) (Not x)))"
    using C by auto
  moreover have
```

```
    "∃n. provable n {}
     (Imp (Imp x y) (Imp (Not (Not x)) (Not (Not y))))"
  proof -
    have "∃n. provable n {Imp x y} (Imp x y)"
      using A by blast
    then have "∃n. provable n {x, Imp x y} y"
      by (metis A MP Un_empty_left Un_insert_left)
    hence "∃n. provable n {x, Imp x y} (Not (Not y))"
      by (metis Un_empty_right double_neg_intro_imp provable.simps)
    hence "∃n. provable n {Imp x y} (Imp x (Not (Not y)))"
      using provable_deduction by auto
    hence "∃n. provable n {Imp x y}
           (Imp (Not (Not x)) (Not (Not y)))"
      by (metis (full_types) MP Un_empty_right
          Un_insert_right double_neg_elim_prem
          insert_commute provable_deduction)
    thus ?thesis
      using provable_deduction by auto
  qed
  thus ?thesis
    by (metis A MP Un_empty_right Un_insert_right
              calculation provable_deduction)
qed
```

Listing B.40: Proof of the formula from dilemma

```
lemma "∃n. provable n {(Imp (Not y) x), (Imp y x)} x"
proof -
  have "∀ x y. ∃n.
        provable n {} (Imp (Imp x (Not y)) (Imp (Imp x y) (Not x)))"
    using D Not_def by auto
  hence "∀ x y. ∃n.
    provable n {(Imp x (Not y))} (Imp (Imp x y) (Not x))"
    by (metis Not_def Un_empty_right provable.simps)
  moreover have
    "∀ x y. ∃n. provable n {(Imp (Not y) (Not x))} (Imp x y)"
    by (metis Un_empty_right provable.simps)
  ultimately have
    "∀ x y. ∃n.
      provable n {(Imp x (Not y))}
      (Imp (Imp (Not y) (Not x)) (Not x))"
    by (metis MP Un_commute provable_deduction)
  hence
    "∀ x y. ∃n. provable n {}
      (Imp (Imp x (Not y)) (Imp (Imp (Not y) (Not x)) (Not x)))"
    using provable_deduction by fastforce
  moreover have
```

```
    "∀ x y. ∃n.
      provable n {}
       (Imp (Imp (Not (Not y)) (Not x)) (Imp x (Not y)))"
    using C by auto
  ultimately have
    "∀ x y. ∃n. provable n {}
     (Imp (Imp (Not (Not y)) (Not x))
          (Imp (Imp (Not y) (Not x)) (Not x)))"
    by (metis A MP Un_empty_right Un_insert_right provable_deduction)
  hence
    "∃n. provable n {}
      (Imp (Imp (Not (Not y)) (Not (Not x)))
           (Imp (Imp (Not y) (Not (Not x))) (Not (Not x))))"
    by blast
  moreover have "∃n.
    provable n {} (Imp (Imp y x) (Imp (Not (Not y)) (Not (Not x))))"
  proof -
    have "∃n. provable n {}
          (Imp (Imp y x) (Imp (Not x) (Not y)))" using C_dual by auto
    moreover have "∃n. provable n {}
      (Imp (Imp (Not x) (Not y)) (Imp (Not (Not y)) (Not (Not x))))"
      using C_dual by auto
    ultimately show ?thesis
      by (metis A MP Un_empty_right
                Un_insert_right provable_deduction)
  qed
  ultimately have
    "∃n. provable n {}
      (Imp (Imp y x)
           (Imp (Imp (Not y) (Not (Not x))) (Not (Not x))))"
    by (metis A MP Un_empty_right
              Un_insert_right provable_deduction)
  hence
    "∃n. provable n {(Imp y x)}
      (Imp (Imp (Not y) (Not (Not x))) (Not (Not x)))"
    by (metis A MP Un_empty_right)
  moreover have
    "∃n. provable n {Imp (Not y) (Not (Not x))}
      (Imp (Not y) (Not (Not x)))"
    using A by auto
  ultimately have
    "∃n. provable n {(Imp y x), (Imp (Not y) (Not (Not x)))}
      (Not (Not x))"
    using MP by fastforce
  hence "∃n. provable n {(Imp y x), (Imp (Not y) (Not (Not x)))} x"
    by (metis Un_empty_right double_neg_elim_imp provable.simps)
  hence
    "∃n. provable n {(Imp y x)} (Imp (Imp (Not y) (Not (Not x))) x)"
    using provable_deduction
```

```
71      by (metis Un_empty_right Un_insert_right insert_commute)
72    moreover have
73      "∃n. provable n {(Imp (Not y) x)}
74        (Imp (Not y) (Not (Not x)))"
75      proof -
76        have "∃n. provable n {(Imp (Not y) x), (Not y)} x"
77          by (metis A MP Un_empty_left Un_insert_left insert_commute)
78        hence "∃n. provable n {(Imp (Not y) x), (Not y)} (Not (Not x))"
79          by (metis Un_empty_right double_neg_elim_imp provable.simps)
80        thus ?thesis
81          by (metis insert_is_Un provable_deduction)
82      qed
83    ultimately show ?thesis
84      by (metis MP insert_is_Un)
85  qed
```

Here, we show the proof of the completeness theorem: $\vDash p \implies \vdash p$. We assume the valuation function ν is given and consequently, the function V is defined. By the assumption, $V(p) = true$, that is, $\sigma(\nu, p) = p$. Furthermore, by the above lemma, we have:

$$\sigma(\nu, x_1), \sigma(\nu, x_2), \cdots, \sigma(\nu, x_i) \vdash p$$
where $x_1, x_2, \cdots, x_i$ are all the variables in the formula p.

Then, we could take arbitrary any valuation function. Hence, we take two valuation functions ν_1 and ν_2 such that $\nu_1(x_j) = \nu_2(x_j) = \mathit{true}$ for all $j \neq i$. By the lemma, we have:

$$x_1, x_2, \cdots, \neg x_i \vdash p$$
$$x_1, x_2, \cdots, x_i \vdash p$$

By deduction theorem,

$$x_1, x_2, \cdots, x_{i-1} \vdash \neg x_i \supset p$$
$$x_1, x_2, \cdots, x_{i-1} \vdash x_i \supset p$$

By Listings B.40, we have:

$$x_1, x_2, \cdots, x_{i-1} \vdash p$$

In the same way, we have:

$$x_1, x_2, \cdots, \neg x_{i-1} \vdash p$$

By deduction theorem,

$$x_1, x_2, \cdots, x_{i-2} \vdash \neg x_{i-1} \supset p$$
$$x_1, x_2, \cdots, x_{i-2} \vdash x_{i-1} \supset p$$

By Listings B.40, we have:

$$x_1, x_2, \cdots, x_{i-2} \vdash p$$

By applying the same procedure, we have:

$$\vdash p$$

□

C. Formal Proof of Continuations in Lambek Calculus

Listing C.1: Unprovability of Plotkin's CPS transformation

```
lemma
  assumes "distinct [a,b,c,d]"
  shows "¬(
    [^a←(^b←^c)] ⊢
     ^d←(((^d←(^a→^d))←((^d←(^b→^d))←^c))→^d)
  )"
proof -
have p17: "¬([^a,^d]⊢^a)"
  apply auto apply(subst (asm) LC.simps) apply auto
  by (simp add: Cons_eq_append_conv)+
have p16: "¬([^a,^d←(^b→^d)]⊢^a)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p17 by fastforce+
have p14: "¬([^a,(^d←(^b→^d))←^c]⊢^a)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p16 by fastforce+
have p26: "¬([^d,^c]⊢^b)"
  apply auto apply(subst (asm) LC.simps) apply auto
  by (simp add: Cons_eq_append_conv)+
have p27: "¬([^d]⊢^b)"
  apply auto apply(subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using assms by auto
have p25: "¬([^d←(^b→^d),^c]⊢^b)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p26 p27 by fastforce+
have p29: "¬([^d←(^b→^d)]⊢^b)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p27 by fastforce+
have p23: "¬([(^d←(^b→^d))←^c,^c]⊢^b)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
```

```
  using p25 p29 by fastforce+
have p33: "¬([^d]⊢^b←^c)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p26 by fastforce+
have p31: "¬([^d←(^b→^d)]⊢^b←^c)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p25 p33 by fastforce+
have p21: "¬([(^d←(^b→^d))←^c]⊢^b←^c)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p23 p31 by fastforce+
have p39: "¬([^a←(^b←^c),^d]⊢^a)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p17 p33 by fastforce+
have p35: "¬([^a←(^b←^c),^d←(^b→^d)]⊢^a)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p16 p31 p39 by fastforce+
have p12: "¬([^a←(^b←^c),(^d←(^b→^d))←^c]⊢^a)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p14 p21 p35 by fastforce+
have p46: "¬([^d]⊢^a)"
  apply auto apply(subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using assms by auto
have p45: "¬([^d←(^b→^d)]⊢^a)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p46 by fastforce+
have p43: "¬([(^d←(^b→^d))←^c]⊢^a)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p45 by fastforce+
have p47: "¬([]⊢^a)"
  apply auto apply (subst (asm) LC.simps) by auto
have p53: "¬([^a,^d,^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p17 p46 p47 by fastforce+
have p54: "¬([^a,^d]⊢^d)"
  apply auto apply(subst (asm) LC.simps) apply auto
  by (simp add: Cons_eq_append_conv)+
have p51: "¬([^a,^d←(^b→^d),^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
```

```
  using p16 p45 p47 p53 p54 by fastforce+
have p56: "¬([^a,^d←(^b→^d)]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p54 by fastforce+
have p49: "¬([^a,(^d←(^b→^d))←^c,^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p14 p43 p47 p51 p56 by fastforce+
have p63: "¬([^a]⊢^d)"
  apply auto apply(subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using assms by auto
have p73: "¬([^a←(^b←^c),^d,^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p39 p46 p47 p53 p33 p63 by fastforce+
have p81: "¬([^a←(^b←^c),^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p54 p63 by fastforce+
have p65: "¬([^a←(^b←^c),^d←(^b→^d),^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p35 p45 p47 p51 p31 p63 p73 p81 by fastforce+
have p83: "¬([^a←(^b←^c),^d←(^b→^d)]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p56 p63 p81 by fastforce+
have p10: "¬([^a←(^b←^c),(^d←(^b→^d))←^c,^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p12 p43 p47 p49 p21 p63 p65 p83 by fastforce+
have p89: "¬([^a,^d]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p53 by fastforce+
have p87: "¬([^a,^d←(^b→^d)]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p51 p89 by fastforce+
have p85: "¬([^a,(^d←(^b→^d))←^c]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p49 p87 by fastforce+
have p109: "¬([^a←(^b←^c),^d]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p73 p89 p33 by fastforce+
```

```
have p99: "¬([^a←(^b←^c),^d←(^b→^d)]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p65 p87 p31 p109 by fastforce+
have p8: "¬([^a←(^b←^c),(^d←(^b→^d))←^c]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p10 p85 p21 p99 by fastforce+
have p119: "¬([^a]⊢(^d←(^a→^d))←((^d←(^b→^d))←^c))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p85 by fastforce+
have p6:
  "¬([^a←(^b←^c)]⊢(^d←(^a→^d))←((^d←(^b→^d))←^c))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p8 p119 by fastforce+
have p129: "¬([^d,^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p46 p47 by fastforce+
have p136: "¬([^b]⊢^a)"
  apply auto apply(subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using assms by auto
have p135: "¬([^b,^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p136 p47 by fastforce+
have p133: "¬([^a→^d]⊢^b→^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using r1 p136 p135 p47 by fastforce+
have p127: "¬([^d←(^b→^d),^a→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p45 p47 p129 p133 by fastforce+
have p143: "¬([^b]⊢^d)"
  apply auto apply(subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using assms by auto
have p142: "¬([]⊢^b→^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  using p143 by fastforce+
have p138: "¬([^d←(^b→^d)]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p142 by fastforce+
have p125: "¬([(^d←(^b→^d))←^c,^a→^d]⊢^d)"
```

```
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p43 p47 p127 p138 by fastforce+
have p147: "¬([^d]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p129 by fastforce+
have p145: "¬([^d←(^b→^d)]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using r1 p136 p127 p147 by fastforce+
have p123: "¬([(^d←(^b→^d))←^c]⊢^d←(^a→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p125 p145 by fastforce+
have p121: "¬([]⊢(^d←(^a→^d))←((^d←(^b→^d))←^c))"
  apply auto apply (subst (asm) LC.simps) apply auto
  using p123 by fastforce+
have p149:
  "¬([^a,((^d←(^a→^d))←((^d←(^b→^d))←^c))→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p119 p121 by fastforce+
have p4:
  "¬([^a←(^b←^c),
   ((^d←(^a→^d))←((^d←(^b→^d))←^c))→^d]⊢^d)"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p6 p121 p149 p63 by fastforce+
have p151:
  "¬([^a]⊢^d←(((^d←(^a→^d))←((^d←(^b→^d))←^c))→^d))"
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p149 by fastforce+
show p2: ?thesis
  apply auto apply (subst (asm) LC.simps) apply auto
  apply (simp_all add: Cons_eq_append_conv)+
  using p4 p151 by fastforce+
qed
```

Listing C.2: Type-restricted CPS transformation

```
lemma rev_right_intro_1:
  assumes "A⊢y←x"
    shows "A@[x]⊢y"
proof -
  have "[y←x]@[x]⊢y"
```

```
    by (metis append_Cons append_Nil
              identity left_intro_2)
  thus ?thesis
    by (metis append_Nil assms cut)
qed

lemma rev_right_intro_2:
  assumes "A⊢x→y"
    shows "[x]@A⊢y"
proof -
  have "[x]@[x→y]⊢y"
    by (metis append_Cons append_Nil
              identity left_intro_1)
  thus ?thesis
    by (metis append_Nil2 assms cut)
qed

theorem type_raising_1: "[x]⊢(y←x)→y"
  using
    rev_right_intro_1
    rev_right_intro_2
    right_intro_1
    identity by blast

theorem type_raising_2: "[x]⊢y←(x→y)"
  using
    rev_right_intro_1
    rev_right_intro_2
    right_intro_2
    identity
  by blast

inductive
    CPS::"'a category⇒'a category⇒'a category⇒bool"
and CPS'::"'a category⇒'a category⇒'a category⇒bool"
  where
  "CPS' a x y ⟹ CPS a x ((a←y)→a)"
  | "CPS' a x y ⟹ CPS a x (a←(y→a))"
  | "⟦CPS' a x1 y1;CPS a x2 y2⟧⟹CPS'a(x1→x2) (y1→y2)"
  | "⟦CPS' a x1 y1;CPS a x2 y2⟧⟹CPS'a(x2←x1) (y2←y1)"
  | "CPS' a (^x) (^x)"

lemma CPS_and_CPS':
  "{y. CPS a x y} = ((λy. (a←y)→a) ‘ {y. CPS' a x y})
                  ∪ ((λy. a←(y→a)) ‘ {y. CPS' a x y})"
  apply auto
  apply (smt(verit,ccfv_SIG)CollectI CPS.simps image_iff)
  apply (simp add: CPS_CPS'.intros(1))
  by (simp add: CPS_CPS'.intros(2))
```

```
55
56 theorem CPS_is_finite:
57   fixes a x :: "'a category"
58   shows "finite {y. CPS' a x y}"
59     and "finite {y. CPS a x y}"
60 proof (induct x)
61   case (Atomic x)
62   case 1
63   have "{y. CPS' a (^x) y} = {(^x)}"
64     proof
65       have "⋀y. CPS' a (^x) y ⟹ y=(^x)"
66         by (metis category.distinct(1)
67                   category.distinct(3) CPS'.simps)
68       thus "{y. CPS' a (^x) y} ⊆ {(^x)}" by auto
69       have "CPS' a (^x) (^x)"
70         by (simp add: CPS_CPS'.intros(5))
71       thus "{(^x)} ⊆ {y. CPS' a (^x) y}" by auto
72     qed
73   thus ?case by auto
74   case 2
75     thus ?case
76     by (simp add:"finite{b.CPS'a(^x) b}"CPS_and_CPS')
77 next
78   case (LeftFunctional x1 x2)
79   case 1
80   have "finite ((λ(u, v). (u←v))
81                 `({y. CPS a x1 y}×{y. CPS' a x2 y}))"
82     using LeftFunctional.hyps(2) LeftFunctional.hyps(3)
83     by blast
84   moreover have "{y. CPS' a (x1←x2) y}
85     =(λ(u,v).(u←v))`({y.CPS a x1 y}×{y.CPS' a x2 y})"
86     apply auto
87     apply (smt (verit, best) CollectI SigmaI
88            category.distinct(1) category.distinct(5)
89            category.inject(2) CPS'.cases pair_imageI)
90     by (simp add: CPS_CPS'.intros(4))
91   ultimately show ?case
92     by presburger
93   case 2
94   have "finite (((λy. (a←y)→a)
95      ` {y. CPS' a (x1←x2) y}) ∪ ((λy. a←(y→a))
96      ` {y. CPS' a (x1←x2) y}))"
97     using "finite {b. CPS' a (x1 ← x2) b}"
98     by blast
99   thus ?case by (simp add: CPS_and_CPS')
100 next
101   case (RightFunctional x1 x2)
102   case 1
103   have "finite ((λ(u, v).
```

```
        (u→v)) ‘ ({y. CPS’ a x1 y}×{y. CPS a x2 y}))"
    using RightFunctional.hyps(1) RightFunctional.hyps(4)
    by blast
  moreover have "{y. CPS’ a (x1→x2) y}
    = (λ(u, v). (u→v))
    ‘ ({y. CPS’ a x1 y}×{y. CPS a x2 y})"
    apply auto
    apply (smt (verit, ccfv_SIG) CollectI SigmaI
           category.distinct(3) category.distinct(5)
           category.inject(3) CPS’.cases pair_imageI)
    by (simp add: CPS_CPS’.intros(3))
  ultimately show ?case by presburger
  case 2 show ?case
  by (simp add: "finite {b. CPS’ a (x1 → x2) b}"
              CPS_and_CPS’)
qed

inductive
rCPS::"’a category⇒’a category⇒’a category⇒bool"
and rCPS’::"’a category⇒’a category⇒’a category⇒bool"
  where
  "rCPS’ a x y ⟹ rCPS a x ((a←y)→a)"
  | "rCPS’ a x y ⟹ rCPS a x (a←(y→a))"
  | "rCPS a x2 y2 ⟹ rCPS’ a ((^x1)→x2) ((^x1)→y2)"
  | "rCPS a x2 y2 ⟹ rCPS’ a (x2←(^x1)) (y2←(^x1))"
  | "rCPS’ a (^x) (^x)"

lemma CPS_contains_rCPS:
  fixes a x :: "’a category"
  shows "{y. rCPS’ a x y} ⊆ {y. CPS’ a x y}"
    and "{y. rCPS a x y} ⊆ {y. CPS a x y}"
proof (induct x)
  case (Atomic x)
  case 1 show ?case
  apply auto
  using CPS’.simps rCPS’.cases by fastforce
  case 2 show ?case
  apply auto
  by (smt (verit, ccfv_threshold)
     "{b. rCPS’ a (^ x) b} ⊆ {b. CPS’ a (^ x) b}"
     rCPS.cases CPS_CPS’.intros(1) CPS_CPS’.intros(2)
     mem_Collect_eq subsetD)
next
  case (LeftFunctional x1 x2)
  case 1 show ?case
  apply auto
  by (smt (verit, ccfv_SIG) Collect_mono_iff
      LeftFunctional.hyps(2) category.distinct(5)
      category.inject(2) CPS’.simps rCPS’.cases)
```

```
  case 2 show ?case
  apply auto
  by (smt (verit, ccfv_SIG) Collect_mono_iff
     "{b. rCPS'a(x1←x2)b}⊆{b.CPS'a(x1←x2)b}"
     CPS.simps rCPS.cases)
next
  case (RightFunctional x1 x2)
  case 1 show ?case
  apply auto
  by (smt (verit, ccfv_SIG) Collect_mono_iff
     RightFunctional.hyps(4) category.inject(3)
     category.simps(9) CPS'.simps rCPS'.cases)
  case 2 show ?case
  apply auto
  by (smt (verit, ccfv_SIG) Collect_mono_iff
      "{b.rCPS'a(x1→x2)b}⊆{b.CPS'a(x1→x2)b}"
      CPS.simps rCPS.cases)
qed

theorem rCPS_is_finite:
  shows "finite {y. rCPS' a x y}"
    and "finite {y. rCPS a x y}"
apply (meson CPS_contains_rCPS(1)
       CPS_is_finite(1) infinite_super)
by (meson CPS_contains_rCPS(2)
    CPS_is_finite(2) infinite_super)

theorem rCPS_transformation:
  fixes a x :: "'a category"
  shows "⋀y. rCPS' a x y ⟹ [x]⊢y"
    and "⋀y. rCPS a x y ⟹ [x]⊢y"
proof (induct x)
  case (Atomic x)
  {case 1
    show ?case
    using "1.prems" rCPS'.cases identity
    by blast}
  {case 2
    have "⋀y. rCPS' a (^x) y ⟹ [^x]⊢y"
      using "2.prems" rCPS'.cases identity by blast
    moreover hence "⋀y. rCPS' a (^x) y ⟹ [^x]⊢(a←y)→a"
      by (metis append_Cons append_Nil cut type_raising_1)
    moreover hence "⋀y. rCPS' a (^x) y ⟹ [^x]⊢a←(y→a)"
      by (metis category.distinct(1) category.distinct(3)
                rCPS'.cases type_raising_2)
    ultimately show "⋀y. rCPS a (^x) y ⟹ [^x]⊢y"
      by (metis rCPS.cases)}
next
  case (LeftFunctional x1 x2)
```

```
  hence "⋀y1. rCPS a x1 y1 ⟹ [x1←x2]@[x2]⊢y1"
    by (metis append.left_neutral append_Cons
              cut identity rev_right_intro_1)
  hence "⋀y1. rCPS a x1 y1 ⟹ [x1←x2]⊢y1←x2"
    using rev_right_intro_1 right_intro_2 by blast
  hence "⋀y1. rCPS' a (x1←x2) (y1←x2)
         ⟹ [x1←x2]⊢y1←x2"
    using rCPS'.cases by force
  thus "⋀y. rCPS' a (x1←x2) y ⟹ [x1←x2]⊢y"
    using rCPS'.cases by force
  hence "⋀y. rCPS' a (x1←x2) y
         ⟹ [x1←x2]⊢(a←y)→a ∧ [x1←x2]⊢a←(y→a)"
    by (metis append.left_neutral append_Cons
              cut type_raising_1 type_raising_2)
  thus "⋀y. rCPS a (x1←x2) y
        ⟹ [x1←x2]⊢y"
    by (metis rCPS.cases)
next
  case (RightFunctional x1 x2)
  hence "⋀y2. rCPS a x2 y2
         ⟹ [x1]@[x1→x2]⊢y2"
    by (metis (no_types, hide_lams)
              append.left_neutral append_Nil2
              cut rev_right_intro_1 type_raising_2)
  hence "⋀y2. rCPS a x2 y2
         ⟹ [x1→x2]⊢x1→y2"
    using rev_right_intro_2 right_intro_1 by blast
  hence "⋀y2. rCPS' a (x1→x2) (x1→y2)
         ⟹ [x1→x2]⊢x1→y2"
    using rCPS'.cases by force
  thus "⋀y. rCPS' a (x1→x2) y
        ⟹ [x1→x2]⊢y"
    using rCPS'.cases by force
  hence "⋀y. rCPS' a (x1→x2) y
         ⟹ [x1→x2]⊢(a←y)→a ∧ [x1→x2]⊢a←(y→a)"
    by (metis append.left_neutral append_Cons
              cut type_raising_1 type_raising_2)
  thus "⋀y. rCPS a (x1→x2) y ⟹ [x1→x2]⊢y"
    by (metis rCPS.cases)
qed
```

D. Publications

Journal

- Masaya Taniguchi and Satoshi Tojo, “Interactive Grammar Extraction from a TreeBank,” Journal of Intelligent Informatics and Smart Technology, vol. 8, 2022

International Conference

- Masaya Taniguchi and Satoshi Tojo, “Left-branching tree in CCG with D combinator,” Logic and Engineering of Natural Language Semantics 19, 2022
- Masaya Taniguchi, “Decidable Parsing Algorithm for Categorial Grammar with Type-raising,” 4th The Proof Society Autumn School and Workshop, 2022
- Masaya Taniguchi and Satoshi Tojo, “Losing a Head in Grammar Extraction,” 14th IEEE International Conference on Knowledge and Systems Engineering, 2022
- Masaya Taniguchi and Satoshi Tojo, and Koji Mineshima, “Interactive CCG Parsing with Incremental Trees,” Bridges and Gaps between Formal and Computational Linguistics, 33rd European Summer School in Logic, Language and Information, 2022
- Masaya Taniguchi, “Unprovability of Continuation-Passing Style Transformation in Lambek Calculus,” Student Session, 33rd European Summer School in Logic, Language and Information, 2022
- Masaya Taniguchi and Satoshi Tojo, “Incremental derivations with Q combinator in CCG,” Logic and Engineering of Natural Language Semantics 18, 2021
- Masaya Taniguchi and Satoshi Tojo, “Interactive Grammar Extraction from a Treebank,” The Sixteenth International Conference on Knowledge, Information and Creativity Support Systems, 2021
- Masaya Taniguchi and Satoshi Tojo, “Generic Framework to Uncross Dependency,” 25th International Symposium on Artificial Life and Robotics, 2020
- Hiroki Sudo, Masaya Taniguchi, and Satoshi Tojo, “Finding Grammar in Music by Evolutionary Linguistics,” Thirteenth International Symposium on Artificial Life and Robotics KICSS 2018, 2018

- Song Yang, Masaya Taniguchi, Satoshi Tojo, “4-valued Logic for Agent Communication with Private / Public Information Passing”, 11th The International Conference on Agents and Artificial Intelligence, 2018

Domestic Conference

- Masaya Taniguchi, “CG for Ungrammatical Sentences: Proving the Unprovability,” みちのく情報伝達学セミナー, 2022, (invited)
- Masaya Taniguchi, “Formalization of Categorial Grammar,” The 17th Theorem Proving and Provers meeting, 2021
- Masaya Taniguchi, “Continuations and Polymorphic Lambek Calculus”, 論理・言語・代数系と計算機科学の周辺領域, 2021
- Masaya Taniguchi, “CPS変換と多相範疇文法”, The 17th Theorem Proving and Provers meeting, 2020
- Masaya Taniguchi, “Introduction to Montague Grammar,” 数学基礎論若手の会, 2018
- Masaya Taniguchi, “Subjunctive Markers and Delimited Continuations,” 第36回記号論理と情報科学研究集会, 2019