Formalized Lambek Calculus in Higher Order Logic (HOL4)

Chun Tian

Scuola di Scienze, Università di Bologna
`chun.tian@studio.unibo.it`
Numero di matricola: 0000735539

Abstract. In this project, a rather complete proof-theoretical formalization of Lambek Calculus (non-associative with arbitrary extensions) has been ported from Coq proof assistent to HOL4 theorem prover, with some improvements and new theorems.
Three deduction systems (Syntactic Calculus, Natural Deduction and Sequent Calculus) of Lambek Calculus are defined with many related theorems proved. The equivalance between these systems are formally proved. Finally, a formalization of Sequent Calculus proofs (where Coq has built-in supports) has been designed and implemented in HOL4. Some basic results including the sub-formula properties of the so-called "cut-free" proofs are formally proved.
This work can be considered as the preliminary work towards a language parser based on category grammars which is not multimodal but still has ability to support context-sensitive languages through customized extensions.

1 Introduction

1.1 Phrase structure grammars and Chomsky Hierarchy

Roughly speaking, sentences in formal and natural languages are constructed by sequences of smaller units, i.e. phrases and words. And determining if any given sequence of phrases or words (coming from a finite lexicon of certain language) forms a grammatically correct sentence in that language, and when it's correct, finding out the "structure" (and even the "meaning") of that sentence, are the central goals in both linguistics and Natural Language Processing in computer science.

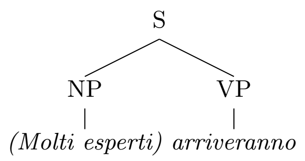
Since the year when Noam Chomsky published his famous "Hierarchy" [1] and his work on phrase structures of English and Spanish [2] in 1950s, the concepts of *context-free* and *context-sensitive* languages (and the intemediate areas between them) with the uses of *rewriting rules* to represent the phrase structure grammar of any given language, has dominated the parsing theory until today.

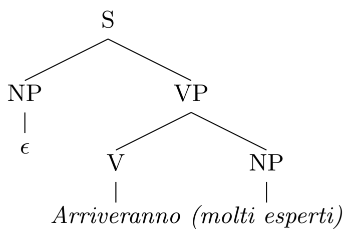
The rule-based grammar for context-free languages and certain more restricted languages, together with the related parsing theory and algorithms, worked so well in handing artificial languages (formal languages and programming languages). But parsing natural languages has always been a complex and difficult area.

The first clue is that, natural languages are NOT context-free, although the precise statement only says: *there exist some natural languages which are not context-free*, and the proof is totally based on some rare phenomenons in certain languages, e.g. the so-called *cross-serial dependencies* in Swiss German and Dutch. For most of other natural languages, e.g. English and Italian, a proof that these languages are not context-free, doesn't exist yet. Therefore, it's quite fair to say, after ignoring some rarely used part, all natural languages can indeed be defined by context-free grammars.

While it's quite common for any programming langauge compiler to use a complete, hand-written or program-generated grammar to describe their targeting language, the task of writing down all the grammar rules for a natural language, e.g. English and Italian, is incredibly hard. (Nevertherless they do exist for English). Efficiency is not a big problem (given the fact that each sentence as parsing unit is relative small, e. g.. almost always less than 100 words), ambiguities are indeed problems, but it's now generally accepted for any grammar parsing tool to output more than one possible interpretations for any given sentences (then there're higher level linguistic theories to help disambiguiting them).

The real problem is, the grammar rules for each languages are so different, none of rewriting rules can be considered as *universal* to all human languages. For example, one may think that all sentences have a Noun Phrase (NP) followed by a Verb Phrase (VP) at top level, e.g. in the Italian sentence "molti esperti arriveranno", NP is "molti esperti" and VP is "arriverranno". But in Italian one can also put the subject after the verb and say "arriveranno esperti" instead. If we have to maintain the order between NP and VP, then this new sentence must be interpreted in a different way (see p. 33 of [3]) with an





empty NP at the beginning (see Fig 1.1). [1] . Here the main argument is, phrase structures of the two sentences shouldn't be such different when Italian speakers simply wanted to emphasize the action (Verb) by speaking that word first. But within the framework of Chomksy, such complexity is inevitable.

Chomsky further developped his phrase structure theory into a more complex theory called "Government and Binding" (GB) in 1990s, and by distinguishing the so-called *surface structure* and *deep structure*, different speaking ways of the same sentence now can be finally turned into the same structure. This new GB theory is dedicated for parsing only natural languages, but the attepmts to find evidences for the existence of universal grammars failed again.

Several years later, Chomsky almost completely abandoned GB theory and turned into the "Minimalist Program", which this new grammar theory is much simplier than GB theory while is still capable to analyze the phrase structure of context-sensitive languages.

1.2 Categorial Grammars and Lambek Calculus

On the other side, different grammar theories have been introduced in parallel. One of them is the so-called *Categorial Grammar.*

Categorial Grammar was firstly introduced by Polish philosopher and logician Kazimierz Ajdukiewicz in 1935 [4], which is based on ideas from precedent Polish logicians. In this whole new grammar system, Ajdukiewicz assigned each word (or lexical entry) a "category" which is defined inductively:

1. *Basic categories.* The two primitive types n (for entities or individuals or first order terms) and s (for propositions or truth values) are categories.
2. *Functor categorites.* Whenever N is a category and $D_1, \dots, D_n$ is a sequence or multiset of categories, then $\dfrac{N}{D_1 \cdots D_n}$ is itself a category.

To decide the category of syntactically connected expressions, the following rules are used:

1. a word or lexical entry is syntactically connected, and its category is the basic their assigned category,
2. given n syntactically connected expressions $d1, \dots, d_n$ of respective categories $D_1, \dots, D_n$, and an expression f of category $\dfrac{N}{D_1 \cdots D_n}$, the expression $fd_1 \cdots d_n$ (or *any permutation of it!*) is syntactically connected and has category N.

There're two notable observations about Ajdukiewicz's categorial grammar:

1. The calculus of categorites works just like elementary arithmetics: functor categorites are divisions of categories, and syntactic connections are multiplications of categories. Thus if one word has category $\dfrac{A}{B}$, the other has category B, their connecion has category $\dfrac{A}{B} \cdot B = A$.
2. Although the original purpose of the grammar is to check logical formulae in Polish notation, in which the word order is not really a problem, but we can imagine that, for highly inflected languages like Latin and Sanskrit, which has rather flexible word orders, this category grammar may result into quite compat while still correct grammars (for at least a small portion). Nevertheless it's impossible to handle English.

In 1953, just two years before Chomsky published his phrase structure grammar theory and the famous hierarchy, Israeli philosopher, mathematician, and linguist Yehoshua Bar-Hillel made an important enhancement [5] to Ajdukiewicz's categorial grammar. The resulting new categorial grammar is now called *Ajdukiewicz-Bar grammar* or *AB grammar.*

[1] Relaxing the order of NP and VP in the grammar is not an option, because it will be illegal in English

In AB grammar, word orders are now respected, and the single "division" operator in categories are not divided into two different directions, i.e. $\dfrac{A}{B}$ now becomes A/B (read as "A over B") and $A \setminus B$ (read as "B under A", and the syntactic connections cannot be permutated with changing the overall category, i.e. in general, $AB \neq BA$. In formal languages, the so-called *category* is defined as follows:

$$C ::= P \quad | \quad (C/C) \quad | \quad (C \setminus C)$$

where P is a basic category usually containing S (for sentences) and np (for noun phrases) and n (for nouns). And the only rules in this grammar systems are, $\forall u, v \in C$,

$$\begin{aligned} u(u \setminus v) &\longrightarrow v \qquad (\setminus_e) \\ (v/u)u &\longrightarrow v \qquad (/_e) \end{aligned}$$

Although AB grammar system looks quite simple, it's proved that AB grammar is weakly equivalent to a corresponding context-free grammar, and there're algorithms to learn AB grammars from parsing results. [6]. However, having only elimination rules, it's impossible to derive fomulae like $(x/y)(y/z) \longrightarrow (x/z)$, because the only purposes of the rules is to reduce two categories into a small one, which is part of the two categories. AB grammar is just a rule-based system but formal system.

Finally, in 1958, Joachim Lambek [7] succesfully defined a formal system for syntactic calculus in which all category formulae can be derived from a basic set of rules (as axioms) and basic logic formulae. It's later called *(Associative) Lambek Calculus*, or **L**. In Lambek Calculus, now the syntactic connections are representated explicitly by a new connective "dot" $(\cdot)$ and the associativity $(A \cdot B) \cdot C = A \cdot (B \cdot C)$ is assumed as axiom (or rule). Thus, the category in Lambek calculus is defined as follows:

$$L ::= P \quad | \quad (L/L) \quad | \quad (L \setminus L) \quad | \quad L \cdot L$$

where P is a basic category. Lambek introduced a relation $\rightarrow$ and use $x \rightarrow y$ to indicate that any expression (string of words) of type (or category) x also has type y, and the following rules were used as axioms of the formal system:

(a) $x \rightarrow x$;
(b) $(x \cdot y) \cdot z \rightarrow x \cdot (y \cdot z)$;
(b') $x \cdot (y \cdot z) \rightarrow (x \cdot y) \cdot z$;
(c) if $x \cdot y \rightarrow z$, then $x \rightarrow z/y$;
(c') if $x \cdot y \rightarrow z$, then $y \rightarrow x \setminus z$;
(d) if $x \rightarrow z/y$, then $x \cdot y \rightarrow z$;
(d') if $y \rightarrow x \setminus z$, then $x \cdot y \rightarrow z$;
(e) if $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$.

Here are a few observations about Lambek Calculus:

1. Although Lambek calculues has more rules then AB grammar, but they're actually equivalent: "a set of strings of words forms a categorial language of one type if and only if it forms a categorial language of the other type" [8];
2. The $\rightarrow$ relation is reflexisive (by (a)) and transitive (by (e)), but it's not symmetric, i.e. in general $x \rightarrow y \not\Rightarrow y \rightarrow x$, thus it's not an equivalence relation at all.
3. The dot $(\cdot)$ connective usually doesn't appears in the lexion, in which each word is associated with one or more categories which is made by only basic categories and two connectives slash (/) and backslash (\).
4. Lambek calculus can only be used to decide if a given string of words has a specific category (e. g. the category of a legal sentence, S), the phrase structure, however, is not immediately known.
5. Lambek calculus has a decision procedure through Gentzen's Sequent Calculus and the corresponding Cut-elimination theorem. This was proved by Lambek (1958) [7].
6. Lambek calculus is NP-complete [9], L-complete [10], and Lambek grammars are context-free [11].

Among other issues, to resolve the key limitation that Lambek calculus cannot be used as a language parser, in 1961, Lambek introduced the so-called Non-associative Lambek Calculus [12], or **NL**. There're two major modifications comparing to previous associative Lambek Calculus:

1. The associative rule (b) and (b') have been removed from axiom rules, and all theorems derivated from them are not derivable any more. It's found that such changes also removed some strange non-language sentences which are acceptable before.

2. To decide the category of a sentence, now the first step is to put all the words into a binary tree with the original order respected, i. e. the *bracketing* process, then applications of inference rules are limited at each tree node.

Using as a language parser, non-associative Lambek Calculus works in this way: for any given string of words, if there exists a bracketing such that the resulting category is the target category (e. g. S), then it's grammatically correct and the corresponding binary tree represent the phrase structures, with the category of each node representing the phrase structure at different levels. It's proven that non-associative Lambek Calculus also has a decision procedure, which surprisingly is polynomial. And for product-free **NL** (there's no dots in lexicon), the parsing process is also polynomial. (see Chapter 4.6 of [6] for detailed discussions and links tooriginal papers)

Beside **L** and **NL**, there's also **NLP** in which the commutativity of products is respect based on **NL**, i. e. $x \cdot y = y \cdot x$. With such assumption, the resulting grammar is quite similar to the original Ajdukiewicz grammar. When both commutativity and associativity are respected, the resulting Lambek calculus is called **LP**.

1.3 Categorial grammars and universal grammar

Chomsky believes that there's an innate *universal grammar* in human mind, and when children start learning languages from very limited vocabulary and extremely incomplete corpus (where machine learning is impossible to converge), what they actually do is to adapt this universal grammar to their mother languages. However, from Chomsky's phrase structure theory and the related grammars based on rewriting rules, terminals and non-terminals, we can't see any evidence for the existence of Universal gammars.

On the other side, in Categorial grammars we see two separated parts:

1. Language independent part: the inference rules for categories;
2. Language dependent part: the lexicon.

Therefore, if we treat the first part at "unversal grammar", what remains to be learn by children (and machine learning program), is just the lexion which associates words to their syntactic categories (which then indicate how a word gets used in this language) and their meanings. This connection (to universal grammars) has made Category Grammars quite attractive. Maybe the language parsing process based on categorial grammars are more natural and close to the natural language processing in human mind.

1.4 Categorial Type Logics

Since natural language is not context-free, while either **L** and **NL** can only handle context-free languages, people seek ways to extend Lambek Calculus to support context-sensitive languages. On the other side, none of **L**, **NL** and **NLP** is perfect for all languages, thus people seek ways to combine different Lambek calculus and use them for different portions of the target language.

One such attempts is the so-called *Multimodal Lambek Calculus*. In this calculus system, there're many copies of category connectives, each associated with a mode index i:

$$L ::= P \quad | \quad (L/_i L) \quad | \quad (L \backslash_i L) \quad | \quad L \cdot_i L$$

Different versions of Lambek Calculus were identified with different mode identifiers, and properties like commutativity and associativity are respected for only connectives of certain modes. Then there's structure rules to bring different modes together. (see Chapter 5 of [6]).

Even further extension includes the unary connectives from linear logics: $\Diamond$ and $\Box$. With all such things, the resulting calculus system has so many rules and operators, and goes far beyond Lambek Calculus, has a new name called *Categorial Type Logics (CTL)*.

Categorial Type Logics has the folloing characteristics:

1. From the view of proof theory, there're still decision procedures, and the cut-elimination theorem can still be proved. However, to find the proofs for category derivations, more complex algorithms must be used (e.g. proof nets) and the time complexity is at least NP-complete.
2. From the view of model theory, CTL were proved to be sound and complete.
3. Learning CTL grammars from corpus has no known results. Given the fact that the structure of categories now contains more connectives in different modes, the learning process is much harder than the normal uni-modal Lambek calculus.

Thus CTL is not quite practical so far, as one of the main authors said in his book: *"So the big open question for multimodal categorial grammars is to find the 'right' combination of structural rules both from the point of view of being able to make the relevant linguistic generalizations and from the point of view of computational complexity."*. (page. 184 of [6])

In this paper, we have tried another possibility to extend Lambek Calculus. We think **NL** is good enough as a basis, because it's as simple as rewriting rules in context-free grammars, and it has known polinomial parsing alghrithm and reasonable learning algorithms. To handle certain context-sentive portions of the target language, we use restrictive *extensions* to extend the **NL** inference rules. These extensions, when satisfying certain restrictions, won't break the validity of existing parsing and learning algorithms while it's possible to handle context-sensitive languages.

1.5 Trusted software and theorem provers

Software cannot be trusted in general, their outputs may be wrong due to either issues in their program code, or issues in the development platform in which they were written. On the other side, either operating systems or development platform (including programming languages) evolve quickly. Any program, once being written, must be maintained continuously, otherwise it soon becomes unusable.

Therefore, we try to convince the audiences the following principles:

1. For any research task in computer science, the last solution is to write a whole new software. Because once the research is done (or paper is published), the software may not be well maintained any more, and in a few years it's dead.
2. Choose programming languages which has long history (at least 10 years), with more than one stable implemenations.
3. Whenever possible, generate program code from higher level tools instead of writing them directly.

On the other side, to gurantee the correctness of algorithms or their implementations, it's preferred to use theorem provers. Many theorem proving software have the ability to generte program code from proved theorems. Coq [2] and Isabelle [3] are such examples.

Sometimes, the theorem prover itself is written as an extension to the development platform, and in this case user can continue extending the theorem prover to make the whole platform into any application software. ACL2 [4] and HOL4 [5] are two notable examples written in this way.

1.6 Why Higher Order Logic?

Comparing to other theorem provers, the Higher Order Logic (HOL) family of theorem provers (HOL4, HOL light, Isabelle) has relatively simple logic foundation: just simple typed lambda calculus with type variables. Comparing to the logic foundation of Coq (Calculus of Inductive Construction, CIC), higier order logic is easier to understand and use, because it has few primitive derivation rules, and students who has just finished studying λ-calculus (typed and untyped) could easily get started with HOL.

HOL has also a small and verified kernel. And when HOL is implemented in Standard ML language, which is the case of HOL4 and Isabelle, the programming platform itself could be also formally verified [6]

Therefore, we choose HOL4 for several reasons:

1. It's a theorem prover well maintained and with long history (30 years). Wigh high chances it will continue living for next 30 years, so is the proof scripts we wrote in it today.
2. The logical foundation of HOL4 is easy to understand, and the primitive inference rule set is small and clear.
3. HOL4 has a rich builtin theory library and a very large set of example code library for reference purposes.
4. The software is written as a natural extension to Standard ML programming language. The theorem prover, the proof scripts, and the tacticals, they're all written in Standard ML. In theory, user can build any customized software upon the theorem prover. Besides, the CakeML project has demonstrated how to write a formally verified programming language compiler and interpreter in HOL4.

[2] `https://coq.inria.fr`
[3] `http://isabelle.in.tum.de`
[4] `http://www.cs.utexas.edu/users/moore/acl2/`
[5] `https://hol-theorem-prover.org`
[6] The CakeML project (`https://cakeml.org`) represents one such efforts.

5. The type variable in HOL4 types are especially suited for expresing syntactic categories in Categorial Grammars. (this will be explained in next section).

Some more differences between HOL and Coq will be discussed later, in section 6.

2 Lambek Calculus in HOL4

In this project, we have implemented Lambek Calculus in HOL4. The exact variant of Lambek Calculus is **NL** (non-associative) with arbitrary extensions. Three deduction systems are implemented: (Axiomatic) Syntactic Calculus, Natural Deduction, and Gentzen's Sequent Calculus. [7]

This work is based on an implementation of Lambek Calculus [8] in Coq proof assistant. In fact, our work so far can be considered as a partial porting of the Coq-based proof scripts: most definitions and theorems are from previous work, with necessary modifications. Here are some major differences:

1. Whenever a relation can be defined as a reflexitive transivive closure (RTC) of another relation, instead of define it manuall as in Coq, now we use HOL's builtin relationTheory to define it and get the related theorems.
2. Coq's built-in support on proofs are not directly portable to HOL, therefore we defined whole new data structures and fill the gaps.

2.1 Syntactic Categories

In HOL, syntactic categories are defined by an algebraic datatype *Form* with a type variable α:

```
Datatype ‘Form = At ’a | Slash Form Form | Backslash Form Form | Dot Form Form‘
```

In this way, all theorems we proved are actually theorem schemas in which there's always a type variable. In practice, user can either define another data structure with enumerated categories ($S, n, np, \ldots$), or directly use strings. For example, in later mode, the basic category "S" can be simply represented as `At` "S", and category "S/np" will be `Slash (At "S") (At "np")` or `At` "S" `/` `At` "np" when grammar supports are enabled.

2.2 Syntactic Calculus

(Axiomatic) Syntactic Calculus is a theory about the `arrow` relation which indicates that a string of words having the first category will also have the seoncd category, it's reflexitive and transitive. In HOL4, such a inductive relation can be easily defined by `Hol_reln` [13] command:

```
val (arrow_rules, arrow_ind, arrow_cases) = Hol_reln ‘
    (!X A. arrow X A A) /\
    (!X A B C. arrow X (Dot A B) C ==> arrow X A (Slash C B)) /\
    (!X A B C. arrow X A (Slash C B) ==> arrow X (Dot A B) C) /\
    (!X A B C. arrow X (Dot A B) C ==> arrow X B (Backslash A C)) /\
    (!X A B C. arrow X B (Backslash A C) ==> arrow X (Dot A B) C) /\
    (!X A B C. arrow X A B /\ arrow X B C ==> arrow X A C) /\
    (!(X :’a arrow_extension) A B. X A B ==> arrow X A B) ‘;
```

which is then broken into the following separated rules:

`one`:
$\vdash$ `arrow` $X\ A\ A$
`beta`:
$\vdash$ `arrow` $X\ (A \cdot B)\ C \Rightarrow$ `arrow` $X\ A\ (C\ /\ B)$
`beta'`:
$\vdash$ `arrow` $X\ A\ (C\ /\ B) \Rightarrow$ `arrow` $X\ (A \cdot B)\ C$
`gamma`:
$\vdash$ `arrow` $X\ (A \cdot B)\ C \Rightarrow$ `arrow` $X\ B\ (A \setminus C)$
`gamma'`:
$\vdash$ `arrow` $X\ B\ (A \setminus C) \Rightarrow$ `arrow` $X\ (A \cdot B)\ C$

[7] Project code can be found at `https://github.com/binghe/informatica-public/tree/master/lambek`
[8] `https://github.com/coq-contribs/lambek`

`comp:`
$\vdash \texttt{arrow}\ X\ A\ B \land \texttt{arrow}\ X\ B\ C \Rightarrow \texttt{arrow}\ X\ A\ C$
`arrow_plus:`
$\vdash X\ A\ B \Rightarrow \texttt{arrow}\ X\ A\ B$

Noticed that, the relation `arrow` is a 3-ary relation with the type "α `arrow_extension ->` α `arrow_extension`", in which the type abbreviation "arrow_extension" is defined as

```
type_abbrev ("arrow_extension", ``:'a Form -> 'a Form -> bool``);
```

An arrow extension defined some extra properties beyond the basic rules. This connection was made by the `arrow_plus` theorem. Thus for any arrow extension X, `arrow` X can be considered as a normal 2-ary relation between two categories with the type "α `Form`".

Since arrow extensions are nothing but normal relations, the union of two such relations can be considered as the "sum" of two arrow extensions, and the subset/superset of relations can be considered as restrictions (or further extensions) to an arrow extension. Then, the arrow extensions of the four different Lambek Calculus, **NL**, **L**, **NLP** and **LP** are defined respectively:

$\vdash \texttt{NL =}$
$\vdash (\forall A\ B\ C.\ \texttt{L}\ (A \cdot (B \cdot C))\ (A \cdot B \cdot C)) \land$
$\quad \forall A\ B\ C.\ \texttt{L}\ (A \cdot B \cdot C)\ (A \cdot (B \cdot C))$
$\vdash \texttt{NLP}\ (A \cdot B)\ (B \cdot A)$
$\vdash \texttt{LP = add\_extension NLP L}$

in which `EMPTY_REL` represents an empty relation, and `add_extension` equals to the union of relations. Thus **LP** is nothing but a sum of **NLP** and **L**.

Then we have proved some common theorems for all Lambek Calculus (i. e. for whatever arrow extensions):

`Dot_mono_right:`
$\vdash \texttt{arrow}\ X\ B'\ B \Rightarrow \texttt{arrow}\ X\ (A \cdot B')\ (A \cdot B)$
`Dot_mono_left:`
$\vdash \texttt{arrow}\ X\ A'\ A \Rightarrow \texttt{arrow}\ X\ (A' \cdot B)\ (A \cdot B)$
`Dot_mono:`
$\vdash \texttt{arrow}\ X\ A\ C \land \texttt{arrow}\ X\ B\ D \Rightarrow \texttt{arrow}\ X\ (A \cdot B)\ (C \cdot D)$
`Slash_mono_left:`
$\vdash \texttt{arrow}\ X\ C'\ C \Rightarrow \texttt{arrow}\ X\ (C'\ /\ B)\ (C\ /\ B)$
`Slash_antimono_right:`
$\vdash \texttt{arrow}\ X\ B'\ B \Rightarrow \texttt{arrow}\ X\ (C\ /\ B)\ (C\ /\ B')$
`Backslash_antimono_left:`
$\vdash \texttt{arrow}\ X\ A\ A' \Rightarrow \texttt{arrow}\ X\ (A' \setminus C)\ (A \setminus C)$
`Backslash_mono_right:`
$\vdash \texttt{arrow}\ X\ C'\ C \Rightarrow \texttt{arrow}\ X\ (A \setminus C')\ (A \setminus C)$

The monotonity of `arrow` relation with respect to arrow extensions is proved too (by induction on all basic arrow rules):

`mono_X:`
$\vdash \texttt{arrow}\ X\ A\ B \Rightarrow \texttt{extends}\ X\ X' \Rightarrow \texttt{arrow}\ X'\ A\ B$

For Lambek extensions supporting permutations, the following theorems are proved:

`pi:`
$\vdash \texttt{extends NLP}\ X \Rightarrow \forall A\ B.\ \texttt{arrow}\ X\ (A \cdot B)\ (B \cdot A)$
`pi_NLP:`
$\vdash \texttt{arrow NLP}\ (A \cdot B)\ (B \cdot A)$
`pi_LP:`
$\vdash \texttt{arrow LP}\ (A \cdot B)\ (B \cdot A)$

For Lambek extensions supporting associativity, the following theorems are proved:

`alfa:`
$\vdash \texttt{extends L}\ X \Rightarrow \forall A\ B\ C.\ \texttt{arrow}\ X\ (A \cdot (B \cdot C))\ (A \cdot B \cdot C)$

```
alfa':
 ⊢ extends L X ⇒ ∀A B C. arrow X (A · B · C) (A · (B · C))
alfa_L:
 ⊢ arrow L (A · (B · C)) (A · B · C)
alfa'_L:
 ⊢ arrow L (A · B · C) (A · (B · C))
alfa_LP:
 ⊢ arrow LP (A · (B · C)) (A · B · C)
alfa'_LP:
 ⊢ arrow LP (A · B · C) (A · (B · C))
```

2.3 Associative Lambek Calculus

For the original associative Lambek Calculus, we have proved all arrow theorems mentioned in Lambek (1958) [7]. All these theorems are named as `L_` plus single letters (with optional prim). The first five are **L** axioms (but actually they're proved from previous definition on the arrow relation:

```
L_a:  ⊢ arrow L x x
L_b:  ⊢ arrow L (x · y · z) (x · (y · z))
L_b': ⊢ arrow L (x · (y · z)) (x · y · z)
L_c:  ⊢ arrow L (x · y) z ⇒ arrow L x (z / y)
L_c': ⊢ arrow L (x · y) z ⇒ arrow L y (x \ z)
L_d:  ⊢ arrow L x (z / y) ⇒ arrow L (x · y) z
L_d': ⊢ arrow L y (x \ z) ⇒ arrow L (x · y) z
L_e:  ⊢ arrow L x y ∧ arrow L y z ⇒ arrow L x z
```

Based on these axioms, the rest **L** theorems about arrow relations can easily be proved by automatic first-order proof searching:

```
L_f:  ⊢ arrow L x (x · y / y)
L_g:  ⊢ arrow L (z / y · y) z
L_h:  ⊢ arrow L y ((z / y) \ z)
L_i:  ⊢ arrow L (z / y · (y / x)) (z / x)
L_j:  ⊢ arrow L (z / y) (z / x / (y / x))
L_k:  ⊢ arrow L (x \ y / z) (x \ (y / z))
L_k': ⊢ arrow L (x \ (y / z)) (x \ y / z)
L_l:  ⊢ arrow L (x / y / z) (x / (z · y))
L_l': ⊢ arrow L (x / (z · y)) (x / y / z)
L_m:  ⊢ arrow L x x′ ∧ arrow L y y′ ⇒ arrow L (x · y) (x′ · y′)
L_n:  ⊢ arrow L x x′ ∧ arrow L y y′ ⇒ arrow L (x / y′) (x′ / y)
```

Finally, the monotone theorems specific to **L** are proved:

```
L_dot_mono_r:
 ⊢ arrow L B B′ ⇒ arrow L (A · B) (A · B′)
L_dot_mono_l:
 ⊢ arrow L A A′ ⇒ arrow L (A · B) (A′ · B)
L_slash_mono_l:
 ⊢ arrow L C C′ ⇒ arrow L (C / B) (C′ / B)
L_slash_antimono_r:
 ⊢ arrow L B B′ ⇒ arrow L (C / B′) (C / B)
L_backslash_antimono_l:
 ⊢ arrow L A A′ ⇒ arrow L (A′ \ C) (A \ C)
L_backslash_mono_r:
 ⊢ arrow L C C′ ⇒ arrow L (A \ C) (A \ C′)
```

The main problem of axiomatic syntactic calculus is that, for complicated categories, it's not easy to "find" the proofs. Automatic proof searching also doesn't work here, because the searching space is infinite. Further more, the arrow relation is not a reduction, because there's no congruence here: if a sub-formula is known to have another category, we can't just replace that sub-formula and expect the whole formula still has the same category. Thus in practice, we have to use other deduction systems which is more convinence for manual proofs or automatic proofs. But for what ever other deduction systems, to be useful they must be proved to be equivalent with Syntactic calculus.

2.4 Natural Deduction for Lambek Calculus

Natural deduction was first invented by Dag Prawitz [14] as a non-semantic approach to derive propositional logic formulae.[9] From a structure view, Natural deduction is nothing but a group of inference rules, each concerns about the *introduction* or *eliminatiob* of a logic connective. Natural deduction is indeed natural: it's successfully adopted in hand-proofs and also the primitive deduction systems of many theorem provers.

Natural deduction has two styles, the original Prawitz style and the Gentzen style (which is borrowed from Genzen's Sequent Calculus, we'll talk about this in next section). The two styles are not totally equivalent in expressive power, sometimes the Prawitz style is unnatural to express certain logic extensions. Thus, in Lambek Calculus, most of time, only the Gentzen style of Natural Deduction is used, and when we're talking "Natural deduction" in this paper, we always refer to its Gentzen style.

The basic unit in Natural Deduction (in Gentzen style) is based on the concept of *Sequent* which represents a statement of the calculus in the following form:

$$A_1, A_2, \ldots, A_n \vdash C$$

If we think the whole statement as a theorem, then C is the conclusion, and $A_1, A_2, \ldots, A_n$ is a list of hypotheses or assumptions in which their orders are irrelevant. In terminalogy of sequents, such a list is called *the antecedent, or the hypotheses of the statement.* The purpose of inference rules is to derive new statements from existing statements.

To adopt Natural Deduction for Lambek Calculus, especially the non-associative version, two steps are done here:

1. The arrow relation between two categories, i.e. $A \longrightarrow B$, is now represented by logic theorem $A \Rightarrow B$, and then a sequent $A \vdash B$, which means by assuming A we can prove B.
2. A sentence as a string of categories of each words, is now represented as the antecedents $A_1, A_2, \ldots, A_n$ of a sequent. However, now the order is essential (for Lambek calculus without **P**). And for Lambek calculus based on **NL**, the binary bracketing is also necessary.

To represent sentence structures as binary trees, a new data structure called *Term* is now defined in HOL4:

```
Datatype ‘Term = OneForm (’a Form) | Comma Term Term‘;
```

Notice the similarity between the `Comma` connective between two Terms and the `Dot` connective between two Forms. In fact, most of times they're convertable from each others, but it's `Comma` representing the boundary of categories of words. To translate a Term into Form, a recursive function called `deltaTranslation` is defined as follows[10]:

```
⊢ (∀f. deltaTranslation (OneForm f) = f) ∧
  ∀t₁ t₂.
    deltaTranslation (Comma t₁ t₂) =
    deltaTranslation t₁ · deltaTranslation t₂
```

Simiar to arrow extensions, which is a 2-ary relation between Terms, now we have another kind of extensions, called Sequent extensions or *gentzen_extension*, which is again abbreviated type as 2-ary relation between Terms:

```
type_abbrev ("gentzen_extension", ‘‘:’a Term -> ’a Term -> bool‘‘);
```

Sequent extensions for **L**, **NL**, **NLP** and **LP** are defined as follows:

```
⊢ NL_Sequent =
⊢ NLP_Sequent (Comma A B) (Comma B A)
⊢ (∀A B C.
     L_Sequent (Comma A (Comma B C)) (Comma (Comma A B) C)) ∧
  ∀A B C.
    L_Sequent (Comma (Comma A B) C) (Comma A (Comma B C))
⊢ LP_Sequent = add_extension NLP_Sequent L_Sequent
```

[9] the semantic approach is to build a truth table to explicitly check if the given formula is true for all possible combinations of boolean values for each variables

[10] The reverse translation is not unique, and it's not needed to have such translations.

To define the inference rules for Natural Deduction, we still need one more device: the ability to *replace* the sub-formula of a Term into another Term:

```
⊢ (∀$F_1$ $F_2$. replace $F_1$ $F_2$ $F_1$ $F_2$) ∧
  (∀$Gamma_1$ $Gamma_2$ $Delta$ $F_1$ $F_2$.
     replace $Gamma_1$ $Gamma_2$ $F_1$ $F_2$ ⇒
     replace (Comma $Gamma_1$ $Delta$) (Comma $Gamma_2$ $Delta$) $F_1$
       $F_2$) ∧
  ∀$Gamma_1$ $Gamma_2$ $Delta$ $F_1$ $F_2$.
    replace $Gamma_1$ $Gamma_2$ $F_1$ $F_2$ ⇒
    replace (Comma $Delta$ $Gamma_1$) (Comma $Delta$ $Gamma_2$) $F_1$ $F_2$
```

Therefore when `replace` A B C D is true, it means in Term A, there's a sub-formula C, and by replacing it with D, we obtain B. In many papers, the Term A is actually $A[C]$, and B is written as $A[D]$.

Related to `replace`, if the goal is to just replace some Commas into Dots, there's a simplified relation called `replaceCommaDot`. To define it, we first define its one-step version is called `replaceCommaDot1`:

```
⊢ replace $T_1$ $T_2$ (Comma (OneForm $A$) (OneForm $B$))
    (OneForm ($A \cdot B$)) ⇒
  replaceCommaDot1 $T_1$ $T_2$
```

Then `replaceCommaDot` is nothing but the RTC of `replaceCommaDot1`:

```
⊢ replaceCommaDot = replaceCommaDot1$^*$
```

To make these replace operators actually useful, we have proved many theorems about them (some have complicated proofs):

```
replaceTransitive':
⊢ replaceCommaDot $T_1$ $T_2$ ∧ replaceCommaDot $T_2$ $T_3$ ⇒
  replaceCommaDot $T_1$ $T_3$
replaceCommaDot_rules:
⊢ (∀$T$. replaceCommaDot $T$ $T$) ∧
  (∀$T_1$ $T_2$ $A$ $B$.
     replace $T_1$ $T_2$ (Comma (OneForm $A$) (OneForm $B$))
       (OneForm ($A \cdot B$)) ⇒
     replaceCommaDot $T_1$ $T_2$) ∧
  (∀$T_1$ $T_2$ $T_3$ $A$ $B$.
     replaceCommaDot $T_1$ $T_2$ ∧
     replace $T_2$ $T_3$ (Comma (OneForm $A$) (OneForm $B$))
       (OneForm ($A \cdot B$)) ⇒
     replaceCommaDot $T_1$ $T_3$) ∧
  ∀$T_1$ $T_2$ $T_3$ $A$ $B$.
    replace $T_1$ $T_2$ (Comma (OneForm $A$) (OneForm $B$))
      (OneForm ($A \cdot B$)) ∧ replaceCommaDot $T_2$ $T_3$ ⇒
    replaceCommaDot $T_1$ $T_3$
replaceMonoRight:
⊢ replaceCommaDot $T_1$ $T_2$ ⇒
  replaceCommaDot (Comma $T_1$ $T_3$) (Comma $T_2$ $T_3$)
replaceMonoLeft:
⊢ replaceCommaDot $T_1$ $T_2$ ⇒
  replaceCommaDot (Comma $T_3$ $T_1$) (Comma $T_3$ $T_2$)
replaceMono:
⊢ replaceCommaDot $T_1$ $T_2$ ∧ replaceCommaDot $T_3$ $T_4$ ⇒
  replaceCommaDot (Comma $T_1$ $T_3$) (Comma $T_2$ $T_4$)
replaceTranslation:
⊢ replaceCommaDot $T$ (OneForm (deltaTranslation $T$))
replace_inv1:
⊢ replace (OneForm $C$) $Gamma'$ (OneForm $X$) $Delta$ ⇒
  ($Gamma'$ = $Delta$) ∧ ($X$ = $C$)
```

```
replace_inv2:
 ⊢ replace (Comma Gamma1 Gamma2) Gamma' (OneForm X) Delta ⇒
   (∃ G.
      (Gamma' = Comma G Gamma2) ∧
      replace Gamma1 G (OneForm X) Delta) ∨
   ∃ G.
     (Gamma' = Comma Gamma1 G) ∧
     replace Gamma2 G (OneForm X) Delta
doubleReplace:
 ⊢ replace Gamma Gamma' T1 T2 ⇒
   ∀ Gamma2 A T3.
     replace Gamma' Gamma2 (OneForm A) T3 ⇒
     (∃ G.
        replace Gamma G (OneForm A) T3 ∧
        replace G Gamma2 T1 T2) ∨
     ∃ G.
       replace T2 G (OneForm A) T3 ∧ replace Gamma Gamma2 T1 G
replaceSameP:
 ⊢ replace T1 T2 T3 T4 ⇒
   ∀ G. ∃ G'. replace T1 G' T3 G ∧ replace G' T2 G T4
replaceTrans:
 ⊢ replace T1 T2 T3 T4 ⇒
   replace T3 T4 T5 T6 ⇒
   replace T1 T2 T5 T6
```

The replace theorems about three deduction systems are more difficult to prove:

```
replace_arrow:
 ⊢ replace Gamma Gamma' T1 T2 ⇒
   arrow X (deltaTranslation T2) (deltaTranslation T1) ⇒
   arrow X (deltaTranslation Gamma') (deltaTranslation Gamma)
replace_arrow':
 ⊢ replace Gamma Gamma' T1 T2 ⇒
   arrow X (deltaTranslation T2) (deltaTranslation T1) ⇒
   arrow X (deltaTranslation Gamma) C ⇒
   arrow X (deltaTranslation Gamma') C
replaceGentzen:
 ⊢ replace Gamma Gamma' Delta Delta' ⇒
   gentzenSequent E Delta' (deltaTranslation Delta) ⇒
   gentzenSequent E Gamma' (deltaTranslation Gamma)
replaceGentzen':
 ⊢ replace Gamma Gamma' Delta Delta' ∧
   gentzenSequent E Delta' (deltaTranslation Delta) ∧
   gentzenSequent E Gamma C ⇒
   gentzenSequent E Gamma' C
replaceNatDed:
 ⊢ replace Gamma Gamma' Delta Delta' ⇒
   natDed E Delta' (deltaTranslation Delta) ⇒
   natDed E Gamma' (deltaTranslation Gamma)
```

Now we use `natDed` E $Gamma$ A to represent $Gamma \vdash A$ under extension E. The type of `natDed` is "α `gentzen_extension ->` α `Term ->` α `Form -> bool`". And the inference rules about Natural Deduction on Lambek Calculus are defined by an inductive relation and then broken into the following separated rules:

```
NatAxiom:
 ⊢ natDed E (OneForm A) A
SlashIntro:
 ⊢ natDed E (Comma Gamma (OneForm B)) A ⇒
   natDed E Gamma (A / B)
```

`BackslashIntro:`
$\vdash$ `natDed` E `(Comma (OneForm` B`)` $Gamma$`)` $A \Rightarrow$
  `natDed` E $Gamma$ `(`B `\` A`)`
`DotIntro:`
$\vdash$ `natDed` E $Gamma$ $A \wedge$ `natDed` E $Delta$ $B \Rightarrow$
  `natDed` E `(Comma` $Gamma$ $Delta$`)` `(`$A \cdot B$`)`
`SlashElim:`
$\vdash$ `natDed` E $Gamma$ `(`A `/` B`)` $\wedge$ `natDed` E $Delta$ $B \Rightarrow$
  `natDed` E `(Comma` $Gamma$ $Delta$`)` A
`BackslashElim:`
$\vdash$ `natDed` E $Gamma$ $B \wedge$ `natDed` E $Delta$ `(`B `\` A`)` $\Rightarrow$
  `natDed` E `(Comma` $Gamma$ $Delta$`)` A
`DotElim:`
$\vdash$ `replace` $Gamma$ $Gamma'$ `(Comma (OneForm` A`) (OneForm` B`))` $Delta \wedge$
  `natDed` E $Delta$ `(`$A \cdot B$`)` $\wedge$ `natDed` E $Gamma$ $C \Rightarrow$
  `natDed` E $Gamma'$ C
`NatExt:`
$\vdash$ `replace` $Gamma$ $Gamma'$ $Delta$ $Delta' \wedge E$ $Delta$ $Delta' \wedge$
  `natDed` E $Gamma$ $C \Rightarrow$
  `natDed` E $Gamma'$ C

These natural deduction rules are quite similar with those arrow rules of Syntactic Calculus. And the elimination rules `SlashElim` and `BackslashElim` are even more close to AB grammar rules. Besides, `replace` operations only appear in `DotElim` and `NatExt` rules. All these characteristics made natural deduction easier to use for proving facts about Lambek Calculus.

We have proved a few theorems as simplified version of above primitive rules, sometimes it's easier to use them instead of the primitive rules:

`NatAxiomGeneralized:`
$\vdash$ `natDed` E $Gamma$ `(deltaTranslation` $Gamma$`)`
`DotElimGeneralized:`
$\vdash$ `replaceCommaDot` $Gamma$ $Gamma' \Rightarrow$
  `natDed` E $Gamma$ $C \Rightarrow$
  `natDed` E $Gamma'$ C
`NatTermToForm:`
$\vdash$ `natDed` E $Gamma$ $C \Rightarrow$
  `natDed` E `(OneForm (deltaTranslation` $Gamma$`))` C
`NatExtSimpl:`
$\vdash$ E $Gamma$ $Gamma' \wedge$ `natDed` E $Gamma$ $C \Rightarrow$ `natDed` E $Gamma'$ C

The main problem for Natural Deduction is the lacking of decision procedures. When proving any theorem in backward way, one has to "guess" many unknown variables to correctly apply those elimination rules. To see this, let's take a look at one elimination rules, e. g. `SlashElim`:

$\vdash$ `natDed` E $Gamma$ `(`A `/` B`)` $\wedge$ `natDed` E $Delta$ $B \Rightarrow$
  `natDed` E `(Comma` $Gamma$ $Delta$`)` A

In this rule, variable B only appears in the antecedents but the conclusion, and when applying this rule from backwards, we have to guess out the value of B, and different values of B may completely change the rest proofs (only the correct guess can lead to a success proof). Thus, automatic proof searching based on these Natural deduction rules are impossible. But if our final goal is to get a langauge parser, we have to have a more reasonable set of rules in which there's essential no guess (this is the so-called "sub-formula property" in sequent calculus, we'll get to this property later).

Thus, to make both manual proving and automatic proof searching possible, now we represent the final solution for the inference system of Lambek Calculus: the Gentzen's Sequent Calculus.

2.5 Gentzen's Sequent Calculus for Lambek Calculus

Sequent Calculus was original introduced by Gerhard Gentzen in two German papers [15] [16] written in 1935. It's a great achievements in the proof theory of classical and intuitionistic logic. To understnad its difference with Natural Deduction, it's better to watch directly the inference rules for Lambek Calculus. In our implementation, the Sequent Calculus is defined through an inductive definition of the relation `gentzenSequent` which has the same type as previous `natDed` relation. Here are its inference rules:

```
SeqAxiom:
 ⊢ gentzenSequent E (OneForm A) A
RightSlash:
 ⊢ gentzenSequent E (Comma Gamma (OneForm B)) A ⇒
   gentzenSequent E Gamma (A / B)
RightBackslash:
 ⊢ gentzenSequent E (Comma (OneForm B) Gamma) A ⇒
   gentzenSequent E Gamma (B \ A)
RightDot:
 ⊢ gentzenSequent E Gamma A ∧ gentzenSequent E Delta B ⇒
   gentzenSequent E (Comma Gamma Delta) (A · B)
LeftSlash:
 ⊢ replace Gamma Gamma′ (OneForm A)
     (Comma (OneForm (A / B)) Delta) ∧
   gentzenSequent E Delta B ∧ gentzenSequent E Gamma C ⇒
   gentzenSequent E Gamma′ C
LeftBackslash:
 ⊢ replace Gamma Gamma′ (OneForm A)
     (Comma Delta (OneForm (B \ A))) ∧
   gentzenSequent E Delta B ∧ gentzenSequent E Gamma C ⇒
   gentzenSequent E Gamma′ C
LeftDot:
 ⊢ replace Gamma Gamma′ (Comma (OneForm A) (OneForm B))
     (OneForm (A · B)) ∧ gentzenSequent E Gamma C ⇒
   gentzenSequent E Gamma′ C
CutRule:
 ⊢ replace Gamma Gamma′ (OneForm A) Delta ∧
   gentzenSequent E Delta A ∧ gentzenSequent E Gamma C ⇒
   gentzenSequent E Gamma′ C
SeqExt:
 ⊢ replace Gamma Gamma′ Delta Delta′ ∧ E Delta Delta′ ∧
   gentzenSequent E Gamma C ⇒
   gentzenSequent E Gamma′ C
```

Comparing with Natural Deduction rules, the following differences are notable:

1. For each of the three connectives (·, / and \), there're Left and Right rules for them; For Natural deduction, instead they're Introduction and Elimination rules.
2. Except for the different relation name, the rule `SeqAxiom` is the same as `NatAxiom`, `RightSlash` is the same as `SlashIntro`, `RightBackslash` is the same as `BackslashIntro`, `RightDot` is the same as `DotIntro`, and finally the rule `SeqExt` is the same as `NatExt`.
3. All Left rules are new, and they all make use of the `replace` predicates. And the result of these rules is to introduce one of the three connectives in the antecedents *without changing the conclusion.*
4. The rule `CutRule` is new in Sequent Calculus, it's not present in Natural Deduction, nor can be proved in Natural Deduction.

Beside above observations, there's one more important fact:

– All rules except for `CutRule` satisfy the so-called "sub-formula property", that is, all variables in the antecedents are sub-formulae of the conclusion.

To see why this is true, let's take a deeper look at `LeftSlash`:

```
 ⊢ replace Gamma Gamma′ (OneForm A)
     (Comma (OneForm (A / B)) Delta) ∧
   gentzenSequent E Delta B ∧ gentzenSequent E Gamma C ⇒
   gentzenSequent E Gamma′ C
```

If we consider the meaning of `replace`, we may rewrite this rule into the following form:

$$(\text{LeftSlash})\ \frac{\mathit{Delta} \vdash B \qquad \mathit{Gamma}[A] \vdash C}{\mathit{Gamma}[(A/B, \mathit{Delta})] \vdash C}$$

To some extents, this rule is just saying $A/B \cdot B \to A$ as in AB grammar. What's more important is the fact that, all variables appearing in the conclusion, i.e. *Gamma*, A, B, *Delta* and C, are variables borrowing from antecedents. The same fact is true for all other `gentzenSequent` rules, except for `CutRule`:

$$(\text{CutRule})\ \frac{\mathit{Delta} \vdash A \qquad \mathit{Gamma}[A] \vdash C}{\mathit{Gamma}[\mathit{Delta}] \vdash C}$$

in which the variable A appears only in antecedents but conclusion.

The sub-formula property is extremely useful for automatic proof searching, because there's nothing to guess when applying these rules from backwards.

Based on above primitive inference rules, we have proved a large amount of derived theorems which is true for all Lambek calculus (i. e. extended from **NL**):

```
SeqAxiomGeneralized:
 ⊢ gentzenSequent E Gamma (deltaTranslation Gamma)
LeftDotSimpl:
 ⊢ gentzenSequent E (Comma (OneForm A) (OneForm B)) C ⇒
   gentzenSequent E (OneForm (A · B)) C
LeftDotGeneralized:
 ⊢ replaceCommaDot T1 T2 ⇒
   gentzenSequent E T1 C ⇒
   gentzenSequent E T2 C
SeqTermToForm:
 ⊢ gentzenSequent E Gamma C ⇒
   gentzenSequent E (OneForm (deltaTranslation Gamma)) C
LeftSlashSimpl:
 ⊢ gentzenSequent E Gamma B ∧ gentzenSequent E (OneForm A) C ⇒
   gentzenSequent E (Comma (OneForm (A / B)) Gamma) C
LeftBackslashSimpl:
 ⊢ gentzenSequent E Gamma B ∧ gentzenSequent E (OneForm A) C ⇒
   gentzenSequent E (Comma Gamma (OneForm (B \ A))) C
CutRuleSimpl:
 ⊢ gentzenSequent E Gamma A ∧ gentzenSequent E (OneForm A) C ⇒
   gentzenSequent E Gamma C
DotRightSlash':
 ⊢ gentzenSequent E (OneForm A) (C / B) ⇒
   gentzenSequent E (OneForm (A · B)) C
DotRightBackslash':
 ⊢ gentzenSequent E (OneForm B) (A \ C) ⇒
   gentzenSequent E (OneForm (A · B)) C
SeqExtSimpl:
 ⊢ E Gamma Gamma' ∧ gentzenSequent E Gamma C ⇒
   gentzenSequent E Gamma' C
application:
 ⊢ gentzenSequent E (OneForm (A / B · B)) A
application':
 ⊢ gentzenSequent E (OneForm (B · B \ A)) A
RightSlashDot:
 ⊢ gentzenSequent E (OneForm (A · C)) B ⇒
   gentzenSequent E (OneForm A) (B / C)
RightBackslashDot:
 ⊢ gentzenSequent E (OneForm (B · A)) C ⇒
   gentzenSequent E (OneForm A) (B \ C)
coApplication:
 ⊢ gentzenSequent E (OneForm A) (A · B / B)
```

```
coApplication':
 ⊢ gentzenSequent E (OneForm A) (B \ (B · A))
mono_E:
 ⊢ gentzenSequent E Gamma A ⇒
   extends E E' ⇒
   gentzenSequent E' Gamma A
monotonicity:
 ⊢ gentzenSequent E (OneForm A) B ∧
   gentzenSequent E (OneForm C) D ⇒
   gentzenSequent E (OneForm (A · C)) (B · D)
isotonicity:
 ⊢ gentzenSequent E (OneForm A) B ⇒
   gentzenSequent E (OneForm (A / C)) (B / C)
isotonicity':
 ⊢ gentzenSequent E (OneForm A) B ⇒
   gentzenSequent E (OneForm (C \ A)) (C \ B)
antitonicity:
 ⊢ gentzenSequent E (OneForm A) B ⇒
   gentzenSequent E (OneForm (C / B)) (C / A)
antitonicity':
 ⊢ gentzenSequent E (OneForm A) B ⇒
   gentzenSequent E (OneForm (B \ C)) (A \ C)
lifting:
 ⊢ gentzenSequent E (OneForm A) (B / A \ B)
lifting':
 ⊢ gentzenSequent E (OneForm A) ((B / A) \ B)
```

For Lambek Calculus extended from **L**,which supports associativity, we have proved some extra theorems:

```
LextensionSimpl:
 ⊢ extends L_Sequent E ∧
   gentzenSequent E (Comma T1 (Comma T2 T3)) C ⇒
   gentzenSequent E (Comma (Comma T1 T2) T3) C
LextensionSimpl':
 ⊢ extends L_Sequent E ∧
   gentzenSequent E (Comma (Comma T1 T2) T3) C ⇒
   gentzenSequent E (Comma T1 (Comma T2 T3)) C
LextensionSimplDot:
 ⊢ extends L_Sequent E ∧
   gentzenSequent E (OneForm (A · (B · C))) D ⇒
   gentzenSequent E (OneForm (A · B · C)) D
LextensionSimplDot':
 ⊢ extends L_Sequent E ∧
   gentzenSequent E (OneForm (A · B · C)) D ⇒
   gentzenSequent E (OneForm (A · (B · C))) D
mainGeach:
 ⊢ extends L_Sequent E ⇒
   gentzenSequent E (OneForm (A / B)) (A / C / (B / C))
mainGeach':
 ⊢ extends L_Sequent E ⇒
   gentzenSequent E (OneForm (B \ A)) ((C \ B) \ C \ A)
secondaryGeach:
 ⊢ extends L_Sequent E ⇒
   gentzenSequent E (OneForm (B / C)) ((A / B) \ (A / C))
secondaryGeach':
 ⊢ extends L_Sequent E ⇒
   gentzenSequent E (OneForm (C \ B)) (C \ A / B \ A)
composition:
```

$\vdash$ `extends L_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(A$ / $B \cdot (B$ / $C)))$ $(A$ / $C)$
`composition'`:
$\vdash$ `extends L_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(C \backslash B \cdot B \backslash A))$ $(C \backslash A)$
`restructuring`:
$\vdash$ `extends L_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(A \backslash B$ / $C))$ $(A \backslash (B$ / $C))$
`restructuring'`:
$\vdash$ `extends L_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(A \backslash (B$ / $C)))$ $(A \backslash B$ / $C)$
`currying`:
$\vdash$ `extends L_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(A$ / $(B \cdot C)))$ $(A$ / C / $B)$
`currying'`:
$\vdash$ `extends L_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(A$ / C / $B))$ $(A$ / $(B \cdot C))$
`decurrying`:
$\vdash$ `extends L_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $((A \cdot B) \backslash C))$ $(B \backslash A \backslash C)$
`decurrying'`:
$\vdash$ `extends L_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(B \backslash A \backslash C))$ $((A \cdot B) \backslash C)$

For Lambek Calculus extended from **NLP**, which supports both commutativity we have proved the following extra theorems:

`NLPextensionSimpl`:
$\vdash$ `extends NLP_Sequent` $E \land$ `gentzenSequent` E `(Comma` T_1 $T_2)$ $C \Rightarrow$
  `gentzenSequent` E `(Comma` T_2 $T_1)$ C
`NLPextensionSimplDot`:
$\vdash$ `extends NLP_Sequent` $E \land$
  `gentzenSequent` E `(OneForm` $(A \cdot B))$ $C \Rightarrow$
  `gentzenSequent` E `(OneForm` $(B \cdot A))$ C
`permutation`:
$\vdash$ `extends NLP_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $A)$ $(B \backslash C) \Rightarrow$
  `gentzenSequent` E `(OneForm` $B)$ $(A \backslash C)$
`exchange`:
$\vdash$ `extends NLP_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(A$ / $B))$ $(B \backslash A)$
`exchange'`:
$\vdash$ `extends NLP_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(B \backslash A))$ $(A$ / $B)$
`preposing`:
$\vdash$ `extends NLP_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $A)$ $(B$ / $(B$ / $A))$
`postposing`:
$\vdash$ `extends NLP_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $A)$ $((A \backslash B) \backslash B)$

For Lambek Calculus extended from **LP**, which supports both commutativity and associativity, we have proved the following theorems:

`mixedComposition`:
$\vdash$ `extends LP_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(A$ / $B \cdot C \backslash B))$ $(C \backslash A)$
`mixedComposition'`:
$\vdash$ `extends LP_Sequent` $E \Rightarrow$
  `gentzenSequent` E `(OneForm` $(B$ / $C \cdot B \backslash A))$ $(A$ / $C)$

3 Equivalences between three deduction systems

So far we have introduced three deduction systems for Lambek Calculus: (axiomatic) syntactic calculus, natural deduction and gentzen's sequent calculus. The convincing of syntactic calculus comes from intuition and the fact that all axiom rules are extremely simple, while the correctness of natural deduction rules and sequent calculus rules is not that clear. Before fully switching to use natural deduction or sequent calculus as the alternative deduction system, we have to prove the equivalences between them and the original syntactic calculus.

3.1 Equivalence between Syntactic Calculus and Sequent Calculus

By equivalence, basically we want to know if any theorem about categories in `arrow` relation has also a proof in natural deduction and sequent calculus, and conversely if any theorems in latter systems has also a proof in syntactac calculus. In another words, the set of theorems about syntactic categories proven in these three deduction systems are exactly the same. However, there're two difficulties we must resolve first:

1. the corresponce between arrow extensions (from Form to Form) and gentzen extensions (from Term to Term) must be defined and proved.
2. There's unique translation from Term to Form, but the other direction is not unique.

The first problem are handled in two directions: from arrow to gentzen extensions, then from genzen to arrow extensions. In the first direction we have the following definition:

```
⊢ arrowToGentzenExt X E ⇔
    ∀A B. X A B ⇒ gentzenSequent E (OneForm A) B
```

Noticed that, this is not a function mapping any arrow extension to a gentzen extension, instead we need to use `gentzenSequent` relation to handle the `OneForm` quoting of the first parameter of arrow extension. Based on this definition, we have proved the correspondences between arrow and gentzen extensions for **NL**, **L**, **NLP**, **LP**:

```
NLToNL_Sequent:
 ⊢ arrowToGentzenExt NL NL_Sequent
NLPToNLP_Sequent:
 ⊢ arrowToGentzenExt NLP NLP_Sequent
LToL_Sequent:
 ⊢ arrowToGentzenExt L L_Sequent
LPToLP_Sequent:
 ⊢ arrowToGentzenExt LP LP_Sequent
```

The other direction is more subtle. From gentzen extension to arrow extension, since the translation from Term to Form is unique, the relationship between two kind of extensions can be defined just by characteristics of themselves:

```
⊢ gentzenToArrowExt E X ⇔
    ∀T₁ T₂.
      E T₁ T₂ ⇒ X (deltaTranslation T₂) (deltaTranslation T₁)
```

However, noticed the interesting part: instead of saying E T_1 T_2 ⇒ X `(deltaTranslation` T_1`)` `(deltaTranslation` we must use swap the order of `deltaTranslation` T_1 and `deltaTranslation` T_2. The reason seems connected with the `CutRule`, and without defing like this we can't prove the equivalece between the three deduction systems. However, we do can prove correspondences between arrow and gentzen extensions for **NL**, **L**, **NLP**, **LP** and other general results using this definition:

```
NL_SequentToNL:
 ⊢ gentzenToArrowExt NL_Sequent NL
NLP_SequentToNLP:
 ⊢ gentzenToArrowExt NLP_Sequent NLP
L_SequentToL:
 ⊢ gentzenToArrowExt L_Sequent L
LP_SequentToLP:
 ⊢ gentzenToArrowExt LP_Sequent LP
```

Also noticed that, in the definition of `gentzenToArrowExt`, when the gentzen extension contains more contents (i.e. more pairs of Terms satisfy the relation) then what's required by arrow extension, the whole definition is still satisfied. To actually define a function which precisely translate a gentzen extension into unique arrow extension and make sure the resulting arrow extension satisfy the definition of `gentzenToArrowExt`, we can define this function like this:

```
⊢ ToArrowExt E =
  CURRY
    {(deltaTranslation y,deltaTranslation x) |
     (x,y) ∈ UNCURRY E}
```

Here we used HOL4's set theory package, and the use of `CURRY` and `UNCURRY` is to converse between standard mathemics definition of relations (with type "α `Form` $\cdot$ α `Form -> bool`") and relations in higher order logics (with type "α `arrow_extension`". We can prove that, the output of this function indeed satisfy the definition of `gentzenToArrowExt`:

```
gentzenToArrowExt_thm:
⊢ gentzenToArrowExt E (ToArrowExt E)
```

With all above devices, now the equivalences between (axiomatic) syntactic calculus and gentzen's Sequent calculus and their common extensions have been proved by the following theorems:

```
arrowToGentzen:
⊢ arrow X A B ⇒
  arrowToGentzenExt X E ⇒
  gentzenSequent E (OneForm A) B
arrowToGentzenNL:
⊢ arrow NL A B ⇒ gentzenSequent NL_Sequent (OneForm A) B
arrowToGentzenNLP:
⊢ arrow NLP A B ⇒ gentzenSequent NLP_Sequent (OneForm A) B
arrowToGentzenL:
⊢ arrow L A B ⇒ gentzenSequent L_Sequent (OneForm A) B
arrowToGentzenLP:
⊢ arrow LP A B ⇒ gentzenSequent LP_Sequent (OneForm A) B

gentzenToArrow:
⊢ gentzenToArrowExt E X ∧ gentzenSequent E Gamma A ⇒
  arrow X (deltaTranslation Gamma) A
NLGentzenToArrow:
⊢ gentzenSequent NL_Sequent Gamma A ⇒
  arrow NL (deltaTranslation Gamma) A
NLPGentzenToArrow:
⊢ gentzenSequent NLP_Sequent Gamma A ⇒
  arrow NLP (deltaTranslation Gamma) A
LGentzenToArrow:
⊢ gentzenSequent L_Sequent Gamma A ⇒
  arrow L (deltaTranslation Gamma) A
LPGentzenToArrow:
⊢ gentzenSequent LP_Sequent Gamma A ⇒
  arrow LP (deltaTranslation Gamma) A
```

This means, if we translated all Comma into Dots, gentzen's Sequent Calculus actually doesn't prove anything new beyond the Syntactic Calculus, although it has more rules, especially the CutRule.

3.2 Equivalence between Natural Deduction and Sequent Calculus

Surprisely, the equivalance between Natural Deduction and Sequent Calculus requires extra assumptions about properties on gentzen extensions. Generally speaking, without any restriction on gentzen extension, Gentzen's sequent calculus has a larger theorem set, therefore is stronger than Natural Deduction.

From Natural Deduction to Sequence Calculus, i.e. the easy direction, we can prove the following theorem by induction on the structure of `gentzenSequent` relation:

natDedToGentzen:
$\vdash$ natDed E $Gamma$ C $\Rightarrow$ gentzenSequent E $Gamma$ C

The other direction is more difficult to prove, and it contains an extra property called condCutExt about the extensions. First we present directly the following important result:

gentzenToNatDed:
$\vdash$ gentzenSequent E $Gamma$ C $\Rightarrow$ condCutExt E $\Rightarrow$ natDed E $Gamma$ C

its the formal proof of this theorem is the longest proof in our LambekTheory, and to successfully prove it, many lemmas are needed.

The definition of condCutExt used in above theorem is defined as follows:

$\vdash$ condCutExt E $\iff$
  $\forall\, Gamma\ T_1\ T_2\ A\ Delta$.
    $E\ T_1\ T_2 \Rightarrow$
    replace T_2 $Gamma$ (OneForm A) $Delta$ $\Rightarrow$
    $\exists\, Gamma'$.
      $E\ Gamma'\ Gamma$ $\wedge$ replace T_1 $Gamma'$ (OneForm A) $Delta$

to understand the precise meaning of this definition, we may consider the meaning of the replace operator and rewrite above definition into the following mathematic definition:

$$E\ \ T_1[A]\ \ T_2[A] \Longrightarrow \exists T_1[Delta].\ \ E\ \ T_1[Delta]\ \ T_2[Delta]$$

This seems like a kind of completeness of the arrow extension: whenever we replace any OneForm term into another Term in the pairs satisfying the gentzen extension, the resulting pairs must also satisfy this extension. However, whenever the definition of extension concerns only about Commas, like in L_-Sequent_def and others, this condition is naturally satisfied: (although the formal proofs are quite long for some of them)

conditionOKNL:
$\vdash$ condCutExt NL_Sequent
conditionOKNLP:
$\vdash$ condCutExt NLP_Sequent
conditionOKL:
$\vdash$ condCutExt L_Sequent

Besides, we have proved that, if two gentzen extensions both satisfy above condCutExt property, so is their sum relation:

condAddExt:
$\vdash$ condCutExt E $\wedge$ condCutExt E' $\Rightarrow$
  condCutExt (add_extension E E')

The formal proofs of gentzenToNatDed also depends on the following lemmas which themselves have long proofs:

CutNatDed:
$\vdash$ natDed E $Gamma$ C $\Rightarrow$
  condCutExt E $\Rightarrow$
  natDed E $Delta$ A $\Rightarrow$
  $\forall\, Gamma'$.
    replace $Gamma$ $Gamma'$ (OneForm A) $Delta$ $\Rightarrow$ natDed E $Gamma'$ C
natDedComposition:
$\vdash$ condCutExt E $\wedge$ natDed E $Gamma$ F_1 $\wedge$
  natDed E (OneForm F_1) F_2 $\Rightarrow$
  natDed E $Gamma$ F_2

Finally, we can merge the two direction together and get the following equivalence theorems about natural deduction and gentzen's sequent calculus for Lambek Calculus (both general and special cases), although they're not used anywhere so far:

gentzenEqNatDed:
$\vdash$ condCutExt $E \Rightarrow$
(gentzenSequent E $Gamma$ C $\iff$ natDed E $Gamma$ C)
NLgentzenEqNatDed:
$\vdash$ gentzenSequent NL_Sequent $Gamma$ C $\iff$
natDed NL_Sequent $Gamma$ C
LgentzenEqNatDed:
$\vdash$ gentzenSequent L_Sequent $Gamma$ C $\iff$ natDed L_Sequent $Gamma$ C
NLPgentzenEqNatDed:
$\vdash$ gentzenSequent NLP_Sequent $Gamma$ C $\iff$
natDed NLP_Sequent $Gamma$ C

3.3 Equivalence between Syntactic Calculus and Natural Deduction

Combining results from previous two sections, the equivalence between Syntactic Calculus and Natural Deduction can be easily proved with gentzen's Sequent Calculus as intermediate steps:

natDedToArrow:
$\vdash$ gentzenToArrowExt E X $\Rightarrow$
natDed E $Gamma$ A $\Rightarrow$
arrow X (deltaTranslation $Gamma$) A
natDedToArrow_E:
$\vdash$ natDed E $Gamma$ A $\Rightarrow$
arrow (ToArrowExt E) (deltaTranslation $Gamma$) A
arrowToNatDed:
$\vdash$ condCutExt E $\wedge$ arrowToGentzenExt X E $\wedge$ arrow X A B $\Rightarrow$
natDed E (OneForm A) B

There seems no way to get equivalence theorems between Syntactic Calculus and Natural Deduction/Sequent Calculus, because of the translations between Terms and Forms.

4 A proof-theoretic formalization of Sequent Calculus

Above formalizations for Lambek Calculus may have provided a good basis for proving theorems about categories using any of the three deduction systems, but one can only do this manually. For the following two purposes, the current work is not enough:

1. Proving theorems about Sequent Calculus proofs, e. g. the cut-elimination theorem (for any Sequent Calculus proof, there exists a corresponding proof without using the CutRule).
2. Finding and generating Sequent Calculus proofs programmatically.

For any of above purpose, we need a data structure to represent a whole proof. HOL theorem prover has already provided data structures to represent terms and theorems, but there's no built-in facility to represent a whole proof. In another word, a "proof" is not a first-class object in HOL.

However, Coq has built-in support for first-class proof object. In Coq, it's possible to define the "degree" of a gentzenSequent proof like this:

```
Fixpoint degreeProof (Atoms : Set) (Gamma : Term Atoms)
 (B : Form Atoms) (E : gentzen_extension) (p : gentzenSequent E Gamma B)
 {struct p} : nat :=
  match p with
  | Ax _ => 0
  | RightSlash _ _ _ H => degreeProof H
  | RightBackSlash _ _ _ H => degreeProof H
  | RightDot _ _ _ _ H1 H2 => max (degreeProof H1) (degreeProof H2)
  | LeftSlash _ _ _ _ _ _ R H1 H2 => max (degreeProof H1) (degreeProof H2)
  | LeftBackSlash _ _ _ _ _ _ R H1 H2 =>
     max (degreeProof H1) (degreeProof H2)
  | LeftDot _ _ _ _ _ R H => degreeProof H
  | CutRule _ _ _ A _ R H1 H2 =>
     max (max (degreeFormula A) (degreeProof H1)) (degreeProof H2)
  | SequentExtension _ _ _ _ _ E R H => degreeProof H
  end.
```

In HOL, it's impossible to defined the same thing directly, because `gentzenSequent` E $Gamma$ B has the type `bool`. To fill the gaps, we have to model a proof by a proof tree "object" by using a datatype definition. Our datatype definition is based on similar modelling work in Isabelle/HOL for Display Logic [17], then all other related inductive relation definitions and theorems are new. (But once the gaps are filled, those theorems in Coq has also the same internal structure in HOL.

4.1 Proof objects

A proof object in Sequent Calculus is represented as a datatype called *Dertree*. A *Dertree* has at least a sequent as its head, and by applying a rule it's derived from one or more other sequents. Thus *Dertree* is nothing but a tree of sequent with each node identified by a rule name. A *Dertree* as a proof can also be "unfinished", and in this case it contains only a (unproved) sequent, nothing else:

```
Datatype `Sequent = Sequent ('a gentzen_extension) ('a Term) ('a Form)`;
Datatype `Rule = SeqAxiom
               | RightSlash | RightBackslash | RightDot
               | LeftSlash  | LeftBackslash  | LeftDot
               | CutRule    | SeqExt`;
Datatype `Dertree = Der ('a Sequent) Rule (Dertree list)
                  | Unf ('a Sequent)`;
```

Here the datatype `Sequent` is just a simple container of three inner objects: a gentzen extension, a Term and a Form. The datatype `Rule` is just an atomic symbol, which takes values from 9 possible rule names as primitive Sequent Calculus rules. So in our formal system, a sequent `gentzenSequent` E $Gamma$ B has the type `bool`, there's no way to access its internat structure, and when it can appears along as a theorem. On the other side a sequent `Sequent` E $Gamma$ B has the type α `Sequent`, it's just a container with accessors and its correctness has to be proved separately.

With above data structures, the following accessors are defined to access their internal structures:

```
⊢ (∀seq v₀ v₁. head (Der seq v₀ v₁) = seq) ∧
  ∀seq. head (Unf seq) = seq
⊢ (concl (Unf (Sequent E Delta A)) = A) ∧
  (concl (Der (Sequent E Delta A) v₀ v₁) = A)
⊢ (prems (Unf (Sequent E Delta A)) = Delta) ∧
  (prems (Der (Sequent E Delta A) v₀ v₁) = Delta)
⊢ (exten (Unf (Sequent E Delta A)) = E) ∧
  (exten (Der (Sequent E Delta A) v₀ v₁) = E)
```

And now we can define `degreeProof` as follows:

```
⊢ (∀S. degreeProof (Der S SeqAxiom []) = 0) ∧
  (∀S H. degreeProof (Der S RightSlash [H]) = degreeProof H) ∧
  (∀S H.
     degreeProof (Der S RightBackslash [H]) = degreeProof H) ∧
  (∀S H₂ H₁.
     degreeProof (Der S RightDot [H₁; H₂]) =
     MAX (degreeProof H₁) (degreeProof H₂)) ∧
  (∀S H₂ H₁.
     degreeProof (Der S LeftSlash [H₁; H₂]) =
     MAX (degreeProof H₁) (degreeProof H₂)) ∧
  (∀S H₂ H₁.
     degreeProof (Der S LeftBackslash [H₁; H₂]) =
     MAX (degreeProof H₁) (degreeProof H₂)) ∧
  (∀S H. degreeProof (Der S LeftDot [H]) = degreeProof H) ∧
  (∀S H. degreeProof (Der S SeqExt [H]) = degreeProof H) ∧
  ∀S H₂ H₁.
    degreeProof (Der S CutRule [H₁; H₂]) =
    MAX (degreeFormula (concl H₁))
      (MAX (degreeProof H₁) (degreeProof H₂))
```

Another closely related concept is the `degreeFormula` which assign each Form as integer as its degree:

```
⊢ (∀C. degreeFormula (At C) = 1) ∧
  (∀F1 F2.
     degreeFormula (F1 / F2) =
     SUC (MAX (degreeFormula F1) (degreeFormula F2))) ∧
  (∀F1 F2.
     degreeFormula (F1 \ F2) =
     SUC (MAX (degreeFormula F1) (degreeFormula F2))) ∧
  ∀F1 F2.
    degreeFormula (F1 · F2) =
    SUC (MAX (degreeFormula F1) (degreeFormula F2))
```

The concept of sub-formula between two Forms that we mentioned several times in previous sections, is now formally defined as an inductive relation:

```
⊢ (∀A. subFormula A A) ∧
  (∀A B C. subFormula A B ⇒ subFormula A (B / C)) ∧
  (∀A B C. subFormula A B ⇒ subFormula A (C / B)) ∧
  (∀A B C. subFormula A B ⇒ subFormula A (B \ C)) ∧
  (∀A B C. subFormula A B ⇒ subFormula A (C \ B)) ∧
  (∀A B C. subFormula A B ⇒ subFormula A (B · C)) ∧
  ∀A B C. subFormula A B ⇒ subFormula A (C · B)
```

And we have also proved many theorems about `subFormula`:

```
subAt:
⊢ subFormula A (At a) ⇒ (A = At a)
subSlash:
⊢ subFormula A (B / C) ⇒
  (A = B / C) ∨ subFormula A B ∨ subFormula A C
subBackslash:
⊢ subFormula A (B \ C) ⇒
  (A = B \ C) ∨ subFormula A B ∨ subFormula A C
subDot:
⊢ subFormula A (B · C) ⇒
  (A = B · C) ∨ subFormula A B ∨ subFormula A C
subFormulaTrans:
⊢ subFormula A B ⇒ subFormula B C ⇒ subFormula A C
```

The sub-formula between a Form and a Term is called `subFormTerm`, which is defined inductively upon `subFormula`:

```
⊢ (∀A B. subFormula A B ⇒ subFormTerm A (OneForm B)) ∧
  (∀A T1 T2. subFormTerm A T1 ⇒ subFormTerm A (Comma T1 T2)) ∧
  ∀A T1 T2. subFormTerm A T1 ⇒ subFormTerm A (Comma T2 T1)
```

Then we have proved several important theorems about `subFormTerm`, most concering about the `replace` operator:

```
oneFormSubEQ:
⊢ subFormTerm A (OneForm B) ⟺ subFormula A B
comSub:
⊢ subFormTerm f (Comma T1 T2) ⇒
  subFormTerm f T1 ∨ subFormTerm f T2
subReplace1:
⊢ replace T1 T2 T3 T4 ⇒ subFormTerm f T3 ⇒ subFormTerm f T1
subReplace2:
⊢ replace T1 T2 T3 T4 ⇒ subFormTerm f T4 ⇒ subFormTerm f T2
subReplace3:
⊢ replace T1 T2 T3 T4 ⇒
  subFormTerm X T1 ⇒
  subFormTerm X T2 ∨ subFormTerm X T3
```

4.2 Derivations of proof tree

So how can we (manually) construct a proof for any theorem in Sequent calculus of Lambek Calculus and prove the resulting `Dertree` object is indeed a valid proof for this theorem? Unfortunately previous Coq proof scripts didn't give any hint, for this part the author has built everything needed from the ground.

The idea comes from beta-reduction in λ-Calculus. First we define the *one-step* derivation from any unfinished `Dertree` by applying one rule which is equivalent with one of `gentzenSequent` rules:

```
⊢ (∀E A.
     derivOne (Unf (Sequent E (OneForm A) A))
       (Der (Sequent E (OneForm A) A) SeqAxiom [])) ∧
  (∀E Gamma A B.
     derivOne (Unf (Sequent E Gamma (A / B)))
       (Der (Sequent E Gamma (A / B)) RightSlash
          [Unf (Sequent E (Comma Gamma (OneForm B)) A)])) ∧
  (∀E Gamma A B.
     derivOne (Unf (Sequent E Gamma (B \ A)))
       (Der (Sequent E Gamma (B \ A)) RightBackslash
          [Unf (Sequent E (Comma (OneForm B) Gamma) A)])) ∧
  (∀E Gamma Delta A B.
     derivOne (Unf (Sequent E (Comma Gamma Delta) (A · B)))
       (Der (Sequent E (Comma Gamma Delta) (A · B)) RightDot
          [Unf (Sequent E Gamma A);
           Unf (Sequent E Delta B)])) ∧
  (∀E Gamma Gamma′ Delta A B C.
     replace Gamma Gamma′ (OneForm A)
       (Comma (OneForm (A / B)) Delta) ⇒
     derivOne (Unf (Sequent E Gamma′ C))
       (Der (Sequent E Gamma′ C) LeftSlash
          [Unf (Sequent E Gamma C);
           Unf (Sequent E Delta B)])) ∧
  (∀E Gamma Gamma′ Delta A B C.
     replace Gamma Gamma′ (OneForm A)
       (Comma Delta (OneForm (B \ A))) ⇒
     derivOne (Unf (Sequent E Gamma′ C))
       (Der (Sequent E Gamma′ C) LeftBackslash
          [Unf (Sequent E Gamma C);
           Unf (Sequent E Delta B)])) ∧
  (∀E Gamma Gamma′ A B C.
     replace Gamma Gamma′ (Comma (OneForm A) (OneForm B))
       (OneForm (A · B)) ⇒
     derivOne (Unf (Sequent E Gamma′ C))
       (Der (Sequent E Gamma′ C) LeftDot
          [Unf (Sequent E Gamma C)])) ∧
  (∀E Delta Gamma Gamma′ A C.
     replace Gamma Gamma′ (OneForm A) Delta ⇒
     derivOne (Unf (Sequent E Gamma′ C))
       (Der (Sequent E Gamma′ C) CutRule
          [Unf (Sequent E Gamma C);
           Unf (Sequent E Delta A)])) ∧
  ∀E Gamma Gamma′ Delta Delta′ C.
    replace Gamma Gamma′ Delta Delta′ ∧ E Delta Delta′ ⇒
    derivOne (Unf (Sequent E Gamma′ C))
      (Der (Sequent E Gamma′ C) SeqExt
         [Unf (Sequent E Gamma C)])
```

The second step is to define "structure rules" as a new inductive relation `deriv` based on `derivOne`. With this relation, one can repeatly apply one-step derivation for all the unfinished sub-proofs in the derivation tree, until all leaves are finished:

```
⊢ (∀D1 D2. derivOne D1 D2 ⇒ deriv D1 D2) ∧
  (∀S R D1 D1'.
     deriv D1 D1' ⇒ deriv (Der S R [D1]) (Der S R [D1'])) ∧
  (∀S R D1 D1' D.
     deriv D1 D1' ⇒
     deriv (Der S R [D1; D]) (Der S R [D1'; D])) ∧
  (∀S R D2 D2' D.
     deriv D2 D2' ⇒
     deriv (Der S R [D; D2]) (Der S R [D; D2'])) ∧
  ∀S R D1 D1' D2 D2'.
    deriv D1 D1' ∧ deriv D2 D2' ⇒
    deriv (Der S R [D1; D2]) (Der S R [D1'; D2'])
```

The last step is to define the chain of derivations as a reduction, so that any two derivation trees are related with each other. This is done by defining a new relation `Deriv` as the reflexitive transitive closure (RTC) of `deriv`:

```
⊢ Deriv = deriv*
```

Merging all above definitions together, we have proved all needed theorems for contructing a whole new proof tree, either manually or automatically:

```
(One step rules)
DerivSeqAxiom:
⊢ Deriv (Unf (Sequent E (OneForm A) A))
    (Der (Sequent E (OneForm A) A) SeqAxiom [])
DerivRightSlash:
⊢ Deriv (Unf (Sequent E Gamma (A / B)))
    (Der (Sequent E Gamma (A / B)) RightSlash
       [Unf (Sequent E (Comma Gamma (OneForm B)) A)])
DerivRightBackslash:
⊢ Deriv (Unf (Sequent E Gamma (B \ A)))
    (Der (Sequent E Gamma (B \ A)) RightBackslash
       [Unf (Sequent E (Comma (OneForm B) Gamma) A)])
DerivRightDot:
⊢ Deriv (Unf (Sequent E (Comma Gamma Delta) (A · B)))
    (Der (Sequent E (Comma Gamma Delta) (A · B)) RightDot
       [Unf (Sequent E Gamma A); Unf (Sequent E Delta B)])
DerivLeftSlash:
⊢ replace Gamma Gamma' (OneForm A)
    (Comma (OneForm (A / B)) Delta) ⇒
  Deriv (Unf (Sequent E Gamma' C))
    (Der (Sequent E Gamma' C) LeftSlash
       [Unf (Sequent E Gamma C); Unf (Sequent E Delta B)])
DerivLeftBackslash:
⊢ replace Gamma Gamma' (OneForm A)
    (Comma Delta (OneForm (B \ A))) ⇒
  Deriv (Unf (Sequent E Gamma' C))
    (Der (Sequent E Gamma' C) LeftBackslash
       [Unf (Sequent E Gamma C); Unf (Sequent E Delta B)])
DerivLeftDot:
⊢ replace Gamma Gamma' (Comma (OneForm A) (OneForm B))
    (OneForm (A · B)) ⇒
  Deriv (Unf (Sequent E Gamma' C))
    (Der (Sequent E Gamma' C) LeftDot
       [Unf (Sequent E Gamma C)])
DerivCutRule:
⊢ replace Gamma Gamma' (OneForm A) Delta ⇒
  Deriv (Unf (Sequent E Gamma' C))
    (Der (Sequent E Gamma' C) CutRule
        [Unf (Sequent E Gamma C); Unf (Sequent E Delta A)])
DerivSeqExt:
 ⊢ replace Gamma Gamma' Delta Delta' ∧ E Delta Delta' ⇒
    Deriv (Unf (Sequent E Gamma' C))
      (Der (Sequent E Gamma' C) SeqExt
        [Unf (Sequent E Gamma C)])

(Structure rules)
DerivOne:
 ⊢ Deriv D₁ D₁' ⇒ Deriv (Der S R [D₁]) (Der S R [D₁'])
DerivLeft:
 ⊢ Deriv D₁ D₁' ⇒ Deriv (Der S R [D₁; D]) (Der S R [D₁'; D])
DerivRight:
 ⊢ Deriv D₂ D₂' ⇒ Deriv (Der S R [D; D₂]) (Der S R [D; D₂'])
DerivBoth:
 ⊢ Deriv D₁ D₁' ⇒
    Deriv D₂ D₂' ⇒
    Deriv (Der S R [D₁; D₂]) (Der S R [D₁'; D₂'])

(RTC rules)
Deriv_refl:
 ⊢ Deriv x x
Deriv_trans:
 ⊢ Deriv x y ∧ Deriv y z ⇒ Deriv x z
```

A `Dertree` is a proof if all its leaves are finished sub-proofs. It has to be defined as another inductive unary relation:

```
 ⊢ (∀S R. Proof (Der S R [])) ∧
    (∀S R D. Proof D ⇒ Proof (Der S R [D])) ∧
    ∀S R D₁ D₂. Proof D₁ ∧ Proof D₂ ⇒ Proof (Der S R [D₁; D₂])
```

Finally we have proved the following theorem which partially guaranteed the correctness of `Deriv` rules. It says for each true sequent, there's a finished proof tree which is derivable from a unfinished Dertree having that sequent as head:[11]

```
gentzenToDeriv:
 ⊢ gentzenSequent E Gamma A ⇒
    ∃D. Deriv (Unf (Sequent E Gamma A)) D ∧ Proof D
```

4.3 Cut-free proofs

In Gentzen's original Sequent Calculus for intuitionistic propositional logic, the so-called *cut-elimination theorem* (Hauptsatz) stands at the central position. Cut-elimination theorem for Lambek Calculus was proved by Lambek for **L** [7] and **NL** [12].

Roughly speaking, this important theorem said that the Cut rule is admissible. In other words, the Cut rule doesn't increase the set of theorems provable from other rules. It's this theorem which guaranteed the existence of decision procedures, because all other rules has the sub-formula property, which is essential for automatic proof searching.

In our project, due to the complexity and large amount of preparation before reaching the Cut-elimination theorem, it's not formally proved yet. Nor the original Coq work has done this proof.[12]

A *cut-free* proof is a proof (Dertree) without using the `CutRule` of gentzen's Sequent Calculus. One way to define cut-free proofs is simply through the `degreeProof` property:

```
 ⊢ CutFreeProof p ⟺ (degreeProof p = 0)
```

[11] The other direction remains unproved: for any sequent, if it leads to a finished Dertree, then it must be a true sequent. We leave this difficult theorem to future work.

[12] to our knowledge, the Cut-elimination theorem for Lambek calculus is never formally verified. This topic along may become another paper after this project, since it's big enough.

This is because, in the definition of `degreeProof`, only the `CutRule` has a non-zero degree, while all other rules has zero degrees. So if the entire Dertree has zero degree, it must not contain any CutRule. Saying the same thing in another way, that's the following theorem:

```
notCutFree:
 ⊢ replace $T_1$ $T_2$ (OneForm A) D ∧ ($p_1$ = Sequent E D A) ∧
   ($p_2$ = Sequent E $T_1$ C) ⇒
   ¬CutFreeProof (Der _ CutRule [Der $p_1$ _ _; Der $p_2$ _ _])
```

The next important concept is *sub-proof.* A proof q is a sub-proof of another proof p if and only if p can be found in one leave of the Dertree of p. The one-step version of this relation is defined as another inductive relation `subProofOne`. Here we omitted its long definition but only show one example, the two LeftSlash cases:

```
ls1:
 ⊢ ($p_0$ = Sequent E Gamma' C) ∧
   ($p_1$ = Der (Sequent E Delta B) R D) ∧
   ($p_2$ = Der (Sequent E Gamma C) R D) ∧
   replace Gamma Gamma' (OneForm A)
     (Comma (OneForm (A / B)) Delta) ⇒
   subProofOne $p_1$ (Der $p_0$ LeftSlash [$p_1$; $p_2$])
ls2:
 ⊢ ($p_0$ = Sequent E Gamma' C) ∧
   ($p_1$ = Der (Sequent E Delta B) R D) ∧
   ($p_2$ = Der (Sequent E Gamma C) R D) ∧
   replace Gamma Gamma' (OneForm A)
     (Comma (OneForm (A / B)) Delta) ⇒
   subProofOne $p_2$ (Der $p_0$ LeftSlash [$p_1$; $p_2$])
```

Based on `subProofOne`, now the full version `subProof` is just a RTC of `subProofOne`: (there's no structure rules here)

```
 ⊢ subProof = subProofOne*
```

Finally we have proved an important sub-formula property if we already have a Cut-free proof. The one-step version (`subFormulaPropertyOne`) is the longest proofs we met in the whole project, the full version is then provable by doing induction on the one-step version.

```
subFormulaPropertyOne:
 ⊢ subProofOne q p ⇒
   extensionSub (exten p) ⇒
   CutFreeProof p ⇒
   ∀x.
     subFormTerm x (prems q) ∨ subFormula x (concl q) ⇒
     subFormTerm x (prems p) ∨ subFormula x (concl p)
subFormulaPropertyOne':
 ⊢ (p = Der (Sequent E $Gamma_1$ B) _ _) ⇒
   (q = Der (Sequent E $Gamma_2$ C) _ _) ⇒
   extensionSub E ⇒
   subProofOne q p ⇒
   CutFreeProof p ⇒
   subFormTerm x $Gamma_2$ ∨ subFormula x C ⇒
   subFormTerm x $Gamma_1$ ∨ subFormula x B
subFormulaProperty:
 ⊢ subProof q p ⇒
   extensionSub (exten p) ⇒
   CutFreeProof p ⇒
   ∀x.
     subFormTerm x (prems q) ∨ subFormula x (concl q) ⇒
     subFormTerm x (prems p) ∨ subFormula x (concl p)
subFormulaProperty':
```

```
⊢ (p = Der (Sequent E Gamma₁ B) _ _) ⇒
  (q = Der (Sequent E Gamma₂ C) _ _) ⇒
  extensionSub E ⇒
  subProof q p ⇒
  CutFreeProof p ⇒
  subFormTerm x Gamma₂ ∨ subFormula x C ⇒
  subFormTerm x Gamma₁ ∨ subFormula x B
```

Basically these theorems said, if we have a cut-free proof, then any sub-formula in its sub-proofs (either in antecedents or conclusion of the sequent) is also a sub-formula of the sequent for whole proof. In another word, NO new formula appears during the proof searching process, or NO need to guess anything! So, if the cut-free proof indeed exists for *every sequent proof using CutRule*, then we do have the decision procedure!

The last thing to explain, is the meaning of `extensionSub` appearing in above theorems. The purpose is to make sure the gentzen extension in the related Lambek Calculus has also the sub-formula property:

```
⊢ extensionSub E ⇔
  ∀ Form T₁ T₂.
    E T₁ T₂ ⇒ subFormTerm Form T₁ ⇒ subFormTerm Form T₂
```

5 Examples

Our first example demonstrates how to use the Lambek Calculus formulization as a toolkit for manual proving derivations about categories of sentences. Suppose we have the following minimal Italian lexicon:

word	**category**
cosa	$S/(S/np)$
guarda	S/inf
passare	inf/np

And the goal is to check if "cosa guarda passare" is an Italian sentance (and then parse it). There're only two ways to bracketing the three words "cosa", "guarda" and "passare":

1. (("cosa", "guarda"), "passare");
2. ("cosa", ("guarda", "passare")).

The first way leads to nothing, while the second way (as also a parsing tree) can be proved to have the category S in either Natural Deduction or Sequent Calculus:

```
⊢ natDed L_Sequent
    (Comma (OneForm (At "S" / (At "S" / At "np")))
       (Comma (OneForm (At "S" / At "inf"))
          (OneForm (At "inf" / At "np")))) (At "S")
⊢ gentzenSequent L_Sequent
    (Comma (OneForm (At "S" / (At "S" / At "np")))
       (Comma (OneForm (At "S" / At "inf"))
          (OneForm (At "inf" / At "np")))) (At "S")
```

Here is the proof tree in Natural Deduction with **L** extension:

$$\dfrac{\dfrac{S/(S/np) \vdash S/(S/np) \qquad \dfrac{\dfrac{\dfrac{S/inf \vdash S/inf \qquad \dfrac{inf/np \vdash inf/np \qquad np \vdash np}{inf/np, np \vdash inf}\,/_e}{S/inf, (inf/np, np) \vdash S}\,/_e}{(S/inf, inf/np), np \vdash S}\,\text{L\_Sequent}}{S/inf, inf/np \vdash S/np}\,/_i}{S/(S/np), (S/inf, inf/np) \vdash S}\,/_e}{(\text{“cosa”}, (\text{“guarda”}, \text{“passare”})) \vdash S}\,\text{Lex}$$

And the proof tree in Sequent Calculus with **L** extension:

$$\dfrac{\dfrac{S\vdash S \qquad \dfrac{\dfrac{\dfrac{\dfrac{S\vdash S \quad \mathit{inf}\vdash \mathit{inf}}{S/\mathit{inf}, \mathit{inf}\vdash S}\,/_L \qquad \mathit{np}\vdash \mathit{np}}{S/\mathit{inf}, (\mathit{inf}/\mathit{np}, \mathit{np})\vdash S}\,/_L}{(S/\mathit{inf}, \mathit{inf}/\mathit{np}), \mathit{np}\vdash S}\,\text{L\_Sequent}}{S/\mathit{inf}, \mathit{inf}/\mathit{np}\vdash S/\mathit{np}}\,/_R}{S/(S/\mathit{np}), (S/\mathit{inf}, \mathit{inf}/\mathit{np})\vdash S}\,/_L}{(\text{“cosa”}, (\text{“guarda”}, \text{“passare”}))\vdash S}\,\text{Lex}$$

If we compare the two proof trees, we can see that, the Sequent Calculus proof is “smaller” in the sense that, all applications of `SeqAxiom` are based on sub-formulae of lower level formulae in the proof tree (in this case they’re all basic categories, but it’s not always like this). To see the advantages of Sequent Calculus more clearly, for the first time we give the formal proof scripts in HOL4 for above two theorems:

```
val cosa_guarda_passare_natDed = store_thm (
   "cosa_guarda_passare_natDed",
  ``natDed L_Sequent (Comma (OneForm ^cosa)
                            (Comma (OneForm ^guarda) (OneForm ^passare)))
                       (At "S")``,
    MATCH_MP_TAC SlashElim
 >> EXISTS_TAC ``(At "S") / (At "np")`` (* guess 1 *)
 >> CONJ_TAC (* 2 sub-goals here *)
 >| [ (* goal 1 *)
      REWRITE_TAC [NatAxiom],
      (* goal 2 *)
      MATCH_MP_TAC SlashIntro \\
      MATCH_MP_TAC NatExtSimpl \\
      EXISTS_TAC ``(Comma (OneForm (At "S" / At "inf"))
                          (Comma (OneForm (At "inf" / At "np"))
                                 (OneForm (At "np"))))`` \\
      CONJ_TAC >- REWRITE_TAC [L_Sequent_rules] \\
      MATCH_MP_TAC SlashElim \\
      EXISTS_TAC ``At "inf"`` \\ (* guess 2 *)
      CONJ_TAC >| (* 2 sub-goals here *)
      [ (* goal 2.1 *)
        REWRITE_TAC [NatAxiom],
        (* goal 2.2 *)
        MATCH_MP_TAC SlashElim \\
        EXISTS_TAC ``At "np"`` \\ (* guess 3 *)
        REWRITE_TAC [NatAxiom] ] ]);

val cosa_guarda_passare_gentzenSequent = store_thm (
   "cosa_guarda_passare_gentzenSequent",
  ``gentzenSequent L_Sequent (Comma (OneForm ^cosa)
                                    (Comma (OneForm ^guarda) (OneForm ^passare)))
                               (At "S")``,
    MATCH_MP_TAC LeftSlashSimpl
 >> CONJ_TAC (* 2 sub-goals here *)
 >| [ (* goal 1 *)
      MATCH_MP_TAC RightSlash \\
      MATCH_MP_TAC SeqExtSimpl \\
      EXISTS_TAC ``(Comma (OneForm (At "S" / At "inf"))
                          (Comma (OneForm (At "inf" / At "np"))
                                 (OneForm (At "np"))))`` \\
      CONJ_TAC >- REWRITE_TAC [L_Sequent_rules] \\
      MATCH_MP_TAC LeftSlashSimpl \\
      CONJ_TAC >| (* 2 sub-goals here *)
      [ (* goal 1.1 *)
        MATCH_MP_TAC LeftSlashSimpl \\
        REWRITE_TAC [SeqAxiom],
        (* goal 1.2 *)
        REWRITE_TAC [SeqAxiom] ],
      (* goal 2 *)
      REWRITE_TAC [SeqAxiom] ]);
```

In the first proof based on Natural Deduction, after rule applications of `SlashElim`, `SlashElim` and the second `SlashElim`, the parameter of `EXISTS_TAC` tactical must be correctly guessed. Since there're infinite possibilities, automatica proof searching will fail with Natural Deduction rules. But in the second proof based on Sequent Calculus, there's no such guess at all. Automatic proof searching algorithm could just try each possible rules (the number is finite) and then does the same strategy for each searching branches, since the formulae at next levels always become smaller, the proof searching process will definitely terminate. (And there're even better algorithms with polinomial time complexity)

The next example is to demonstrates how to manually construct a proof tree object for above sentence and prove that the Dertree is indeed a valid proof.

At the beginning we have the following unfinished Dertree:

```
val r0 =
  ``(Unf (Sequent L_Sequent (Comma (OneForm (At "S" / (At "S" / At "np")))
                                   (Comma (OneForm (At "S" / At "inf"))
                                          (OneForm (At "inf" / At "np"))))
                            (At "S")))``;
```

If we try to manually expand this Dertree into the final proof tree according to above manual proof, at the next step we could have a new Dertree like this:

```
val r1 =
  ``(Der (Sequent L_Sequent (Comma (OneForm (At "S" / (At "S" / At "np")))
                                   (Comma (OneForm (At "S" / At "inf"))
                                          (OneForm (At "inf" / At "np"))))
                            (At "S"))
         LeftSlash
      [ (Unf (Sequent L_Sequent (OneForm (At "S")) (At "S"))) ;
        (Unf (Sequent L_Sequent (Comma (OneForm (At "S" / At "inf"))
                                       (OneForm (At "inf" / At "np")))
                                (At "S" / At "np"))) ])``;
```

And we can prove this new Dertree is derived from the last one:

```
⊢ Deriv
     (Unf
        (Sequent L_Sequent
           (Comma (OneForm (At “S” / (At “S” / At “np”)))
              (Comma (OneForm (At “S” / At “inf”))
                 (OneForm (At “inf” / At “np”)))) (At “S”)))
     (Der
        (Sequent L_Sequent
           (Comma (OneForm (At “S” / (At “S” / At “np”)))
              (Comma (OneForm (At “S” / At “inf”))
                 (OneForm (At “inf” / At “np”)))) (At “S”))
        LeftSlash
        [Unf (Sequent L_Sequent (OneForm (At “S”)) (At “S”));
         Unf
           (Sequent L_Sequent
              (Comma (OneForm (At “S” / At “inf”))
                 (OneForm (At “inf” / At “np”)))
              (At “S” / At “np”))])
```

If we repeat this process and manually expand the Dertree while prove the new Dertree is derived from the last Dertree, finally the transitivity of `Deriv` relation will let us prove that, the final finished Dertree is indeed a valid proof for the original unfinished Dertree, and thus it's indeed a valid proof for the original Sequent theorem. We omited the intermediate steps and show only the proof script of the final step:

```
val r0_to_final = store_thm (
   "r0_to_final", ``Deriv ^r0 ^r_final``,
    ASSUME_TAC r0_to_r1''
 >> ASSUME_TAC (derivToDeriv r1_to_r2)
 >> ASSUME_TAC (derivToDeriv r2_to_r3)
 >> ASSUME_TAC (derivToDeriv r3_to_r4)
 >> ASSUME_TAC (derivToDeriv r4_to_r5)
 >> ASSUME_TAC (derivToDeriv r5_to_r6)
 >> ASSUME_TAC (derivToDeriv r6_to_final)
 >> REPEAT (IMP_RES_TAC Deriv_trans));
```

And the actual proved theorem:

```
⊢ Deriv
    (Unf
       (Sequent L_Sequent
          (Comma (OneForm (At “S” / (At “S” / At “np”)))
             (Comma (OneForm (At “S” / At “inf”))
                (OneForm (At “inf” / At “np”)))) (At “S”)))
    (Der
       (Sequent L_Sequent
          (Comma (OneForm (At “S” / (At “S” / At “np”)))
             (Comma (OneForm (At “S” / At “inf”))
                (OneForm (At “inf” / At “np”)))) (At “S”))
       LeftSlash
       [Der (Sequent L_Sequent (OneForm (At “S”)) (At “S”))
          SeqAxiom [];
        Der
          (Sequent L_Sequent
             (Comma (OneForm (At “S” / At “inf”))
                (OneForm (At “inf” / At “np”)))
             (At “S” / At “np”)) RightSlash
          [Der
             (Sequent L_Sequent
                (Comma
                   (Comma (OneForm (At “S” / At “inf”))
                      (OneForm (At “inf” / At “np”)))
                   (OneForm (At “np”))) (At “S”)) SeqExt
             [Der
                (Sequent L_Sequent
                   (Comma (OneForm (At “S” / At “inf”))
                      (Comma (OneForm (At “inf” / At “np”))
                         (OneForm (At “np”)))) (At “S”))
                LeftSlash
                [Der
                   (Sequent L_Sequent
                      (Comma (OneForm (At “S” / At “inf”))
                         (OneForm (At “inf”))) (At “S”))
                   LeftSlash
                   [Der
                      (Sequent L_Sequent (OneForm (At “S”))
                         (At “S”)) SeqAxiom [];
                    Der
                      (Sequent L_Sequent (OneForm (At “inf”))
                         (At “inf”)) SeqAxiom []];
                 Der
                   (Sequent L_Sequent (OneForm (At “np”))
                      (At “np”)) SeqAxiom []]]]])
```

The final Dertree is quite long and hard to read, but it does contain all necessary information about the details of the proof. If there's an automatic proof searching algorithm, in theory we can implement it either as a special tactical for proving theorems about relation `gentzenSequent`, or as a program taking an initial Dertree and produce a finished Dertree with a related theorem as above one. If we need a language parser instead, then the useful output will be the bracketed binary tree, together with categories at each node of the tree.

6 Differences between HOL and Coq

There're essential differences between HOL and Coq for many logical definitions that we have ported from Coq, although they looks similiar.

6.1 Inductive datatypes and relations

In Coq, both inductive data types and relations are defined as Inductive sets. For instance, the definition of type `Form` and the `arrow` relation for Syntactic Calculus:

```
Inductive Form (Atoms : Set) : Set :=
  | At : Atoms -> Form Atoms
  | Slash : Form Atoms -> Form Atoms -> Form Atoms
  | Dot : Form Atoms -> Form Atoms -> Form Atoms
  | Backslash : Form Atoms -> Form Atoms -> Form Atoms.

Inductive arrow (Atoms : Set) : Form Atoms -> Form Atoms -> Set :=
  | one : forall A : Form Atoms, arrow A A
  | comp : forall A B C : Form Atoms, arrow A B -> arrow B C -> arrow A C
  | beta :
      forall A B C : Form Atoms, arrow (Dot A B) C -> arrow A (Slash C B)
  | beta' :
      forall A B C : Form Atoms, arrow A (Slash C B) -> arrow (Dot A B) C
  | gamma :
      forall A B C : Form Atoms,
      arrow (Dot A B) C -> arrow B (Backslash A C)
  | gamma' :
      forall A B C : Form Atoms,
      arrow B (Backslash A C) -> arrow (Dot A B) C
  | arrow_plus : forall A B : Form Atoms, X A B -> arrow A B.
```

while in HOL, although they're still inductive defintions, but they're handled differently: the former is defined by `Define`, and latter is defined by `Hol_reln`:

```
val _ = Datatype ‘Form = At ’a | Slash Form Form | Backslash Form Form | Dot Form Form‘;

val (arrow_rules, arrow_ind, arrow_cases) = Hol_reln ‘
    (!X A. arrow X A A) /\                                          (* one *)
    (!X A B C. arrow X (Dot A B) C ==> arrow X A (Slash C B)) /\     (* beta *)
    (!X A B C. arrow X A (Slash C B) ==> arrow X (Dot A B) C) /\     (* beta' *)
    (!X A B C. arrow X (Dot A B) C ==> arrow X B (Backslash A C)) /\ (* gamma *)
    (!X A B C. arrow X B (Backslash A C) ==> arrow X (Dot A B) C) /\ (* gamma' *)
    (!X A B C. arrow X A B /\ arrow X B C ==> arrow X A C) /\        (* comp *)
    (!(X :'a arrow_extension) A B. X A B ==> arrow X A B) ‘;         (* arrow_plus *)
```

There's no magic behind HOL's datatype definition, because what the `Datatype` does is to prove a series of theorems which completely characteries the type `Form`:

```
Form_TY_DEF:
 ⊢ ∃rep.
     TYPE_DEFINITION
       (λa₀'.
          ∀'Form' .
            (∀a₀'.
               (∃a.
                  a₀' =
                  (λa.
                     ind_type$CONSTR 0 a (λn. ind_type$BOTTOM))
                    a) ∨
               (∃a₀ a₁.
                    ($a_0'$ =
                     ($\lambda\, a_0\ a_1$.
                        ind_type$CONSTR (SUC 0) ARB
                          (ind_type$FCONS $a_0$
                             (ind_type$FCONS $a_1$
                                ($\lambda\, n$. ind_type$BOTTOM)))) $a_0\ a_1$) $\land$
                    $'Form'\ a_0$ $\land$ $'Form'\ a_1$) $\lor$
                 ($\exists\, a_0\ a_1$.
                    ($a_0'$ =
                     ($\lambda\, a_0\ a_1$.
                        ind_type$CONSTR (SUC (SUC 0)) ARB
                          (ind_type$FCONS $a_0$
                             (ind_type$FCONS $a_1$
                                ($\lambda\, n$. ind_type$BOTTOM)))) $a_0\ a_1$) $\land$
                    $'Form'\ a_0$ $\land$ $'Form'\ a_1$) $\lor$
                 ($\exists\, a_0\ a_1$.
                    ($a_0'$ =
                     ($\lambda\, a_0\ a_1$.
                        ind_type$CONSTR (SUC (SUC (SUC 0))) ARB
                          (ind_type$FCONS $a_0$
                             (ind_type$FCONS $a_1$
                                ($\lambda\, n$. ind_type$BOTTOM)))) $a_0\ a_1$) $\land$
                    $'Form'\ a_0$ $\land$ $'Form'\ a_1$) $\Rightarrow$
                 $'Form'\ a_0'$) $\Rightarrow$
              $'Form'\ a_0'$) $rep$
Form_case_def:
 $\vdash$ ($\forall\, a\ f\ f_1\ f_2\ f_3$. Form_CASE (At $a$) $f\ f_1\ f_2\ f_3$ = $f\ a$) $\land$
   ($\forall\, a_0\ a_1\ f\ f_1\ f_2\ f_3$.
      Form_CASE ($a_0$ / $a_1$) $f\ f_1\ f_2\ f_3$ = $f_1\ a_0\ a_1$) $\land$
   ($\forall\, a_0\ a_1\ f\ f_1\ f_2\ f_3$.
      Form_CASE ($a_0$ \ $a_1$) $f\ f_1\ f_2\ f_3$ = $f_2\ a_0\ a_1$) $\land$
   $\forall\, a_0\ a_1\ f\ f_1\ f_2\ f_3$. Form_CASE ($a_0\cdot a_1$) $f\ f_1\ f_2\ f_3$ = $f_3\ a_0\ a_1$
Form_size_def:
 $\vdash$ ($\forall f\ a$. Form_size $f$ (At $a$) = 1 + $f\ a$) $\land$
   ($\forall f\ a_0\ a_1$.
      Form_size $f$ ($a_0$ / $a_1$) =
      1 + (Form_size $f\ a_0$ + Form_size $f\ a_1$)) $\land$
   ($\forall f\ a_0\ a_1$.
      Form_size $f$ ($a_0$ \ $a_1$) =
      1 + (Form_size $f\ a_0$ + Form_size $f\ a_1$)) $\land$
   $\forall f\ a_0\ a_1$.
     Form_size $f$ ($a_0\cdot a_1$) =
     1 + (Form_size $f\ a_0$ + Form_size $f\ a_1$)
Form_11:
 $\vdash$ ($\forall\, a\ a'$. (At $a$ = At $a'$) $\iff$ ($a = a'$)) $\land$
   ($\forall\, a_0\ a_1\ a_0'\ a_1'$.
      ($a_0$ / $a_1$ = $a_0'$ / $a_1'$) $\iff$ ($a_0 = a_0'$) $\land$ ($a_1 = a_1'$)) $\land$
   ($\forall\, a_0\ a_1\ a_0'\ a_1'$.
      ($a_0$ \ $a_1$ = $a_0'$ \ $a_1'$) $\iff$ ($a_0 = a_0'$) $\land$ ($a_1 = a_1'$)) $\land$
   $\forall\, a_0\ a_1\ a_0'\ a_1'$.
     ($a_0\cdot a_1 = a_0'\cdot a_1'$) $\iff$ ($a_0 = a_0'$) $\land$ ($a_1 = a_1'$)
Form_Axiom:
 $\vdash$ $\exists fn$.
     ($\forall\, a$. $fn$ (At $a$) = $f_0\ a$) $\land$
     ($\forall\, a_0\ a_1$. $fn$ ($a_0$ / $a_1$) = $f_1\ a_0\ a_1$ ($fn\ a_0$) ($fn\ a_1$)) $\land$
     ($\forall\, a_0\ a_1$. $fn$ ($a_0$ \ $a_1$) = $f_2\ a_0\ a_1$ ($fn\ a_0$) ($fn\ a_1$)) $\land$
     $\forall\, a_0\ a_1$. $fn$ ($a_0\cdot a_1$) = $f_3\ a_0\ a_1$ ($fn\ a_0$) ($fn\ a_1$)
Form_case_cong:
```

$$\begin{array}{l}
\vdash (M = M') \land (\forall a.\ (M' = \mathtt{At}\ a) \Rightarrow (f\ a = f'\ a)) \land \\
\quad (\forall a_0\ a_1.\ (M' = a_0\ /\ a_1) \Rightarrow (f_1\ a_0\ a_1 = f_1'\ a_0\ a_1)) \land \\
\quad (\forall a_0\ a_1.\ (M' = a_0 \setminus a_1) \Rightarrow (f_2\ a_0\ a_1 = f_2'\ a_0\ a_1)) \land \\
\quad (\forall a_0\ a_1.\ (M' = a_0 \cdot a_1) \Rightarrow (f_3\ a_0\ a_1 = f_3'\ a_0\ a_1)) \Rightarrow \\
\quad (\mathtt{Form\_CASE}\ M\ f\ f_1\ f_2\ f_3 = \mathtt{Form\_CASE}\ M'\ f'\ f_1'\ f_2'\ f_3')
\end{array}$$

`Form_distinct:`

$$\begin{array}{l}
\vdash (\forall a_1\ a_0\ a.\ \mathtt{At}\ a \neq a_0\ /\ a_1) \land (\forall a_1\ a_0\ a.\ \mathtt{At}\ a \neq a_0 \setminus a_1) \land \\
\quad (\forall a_1\ a_0\ a.\ \mathtt{At}\ a \neq a_0 \cdot a_1) \land \\
\quad (\forall a_1'\ a_1\ a_0'\ a_0.\ a_0\ /\ a_1 \neq a_0' \setminus a_1') \land \\
\quad (\forall a_1'\ a_1\ a_0'\ a_0.\ a_0\ /\ a_1 \neq a_0' \cdot a_1') \land \\
\quad \forall a_1'\ a_1\ a_0'\ a_0.\ a_0 \setminus a_1 \neq a_0' \cdot a_1'
\end{array}$$

`Form_induction:`

$$\begin{array}{l}
\vdash (\forall a.\ P\ (\mathtt{At}\ a)) \land (\forall F'\ F_0.\ P\ F' \land P\ F_0 \Rightarrow P\ (F'\ /\ F_0)) \land \\
\quad (\forall F'\ F_0.\ P\ F' \land P\ F_0 \Rightarrow P\ (F' \setminus F_0)) \land \\
\quad (\forall F'\ F_0.\ P\ F' \land P\ F_0 \Rightarrow P\ (F' \cdot F_0)) \Rightarrow \\
\quad \forall F'.\ P\ F'
\end{array}$$

`Form_nchotomy:`

$$\begin{array}{l}
\vdash (\exists a.\ F'F' = \mathtt{At}\ a) \lor (\exists F'\ F_0.\ F'F' = F'\ /\ F_0) \lor \\
\quad (\exists F'\ F_0.\ F'F' = F' \setminus F_0) \lor \exists F'\ F_0.\ F'F' = F' \cdot F_0
\end{array}$$

where the constant `TYPE_DEFINITION` is defined in the theory `bool` by:

$$\begin{array}{l}
\vdash \mathtt{TYPE\_DEFINITION} = \\
\quad (\lambda P\ rep. \\
\qquad (\forall x'\ x''.\ (rep\ x' = rep\ x'') \Rightarrow (x' = x'')) \land \\
\qquad \forall x.\ P\ x \iff \exists x'.\ x = rep\ x')
\end{array}$$

With all above theorems, any theorem in which the type `Form` is used, can be handled by combining above theorems with other related theorems, although most of time, some HOL's tacticals can benefit from these generated theorems implicitly. In Coq, datatypes are handled in black-box like ways: user has no direct access to any theorem related to datatype themselves.

Similarily, an induction relation `arrow` in HOL is just defined by some generated theorems:

`arrow_def:`

$$\begin{array}{l}
\vdash \mathtt{arrow} = \\
\quad (\lambda a_0\ a_1\ a_2. \\
\qquad \forall arrow'. \\
\qquad\quad (\forall a_0\ a_1\ a_2. \\
\qquad\qquad (a_2 = a_1) \lor \\
\qquad\qquad (\exists B\ C.\ (a_2 = C\ /\ B) \land arrow'\ a_0\ (a_1 \cdot B)\ C) \lor \\
\qquad\qquad (\exists A\ B.\ (a_1 = A \cdot B) \land arrow'\ a_0\ A\ (a_2\ /\ B)) \lor \\
\qquad\qquad (\exists A\ C.\ (a_2 = A \setminus C) \land arrow'\ a_0\ (A \cdot a_1)\ C) \lor \\
\qquad\qquad (\exists A\ B.\ (a_1 = A \cdot B) \land arrow'\ a_0\ B\ (A \setminus a_2)) \lor \\
\qquad\qquad (\exists B.\ arrow'\ a_0\ a_1\ B \land arrow'\ a_0\ B\ a_2) \lor a_0\ a_1\ a_2 \Rightarrow \\
\qquad\qquad arrow'\ a_0\ a_1\ a_2) \Rightarrow \\
\qquad\quad arrow'\ a_0\ a_1\ a_2)
\end{array}$$

`arrow_rules:`

$$\begin{array}{l}
\vdash (\forall X\ A.\ \mathtt{arrow}\ X\ A\ A) \land \\
\quad (\forall X\ A\ B\ C.\ \mathtt{arrow}\ X\ (A \cdot B)\ C \Rightarrow \mathtt{arrow}\ X\ A\ (C\ /\ B)) \land \\
\quad (\forall X\ A\ B\ C.\ \mathtt{arrow}\ X\ A\ (C\ /\ B) \Rightarrow \mathtt{arrow}\ X\ (A \cdot B)\ C) \land \\
\quad (\forall X\ A\ B\ C.\ \mathtt{arrow}\ X\ (A \cdot B)\ C \Rightarrow \mathtt{arrow}\ X\ B\ (A \setminus C)) \land \\
\quad (\forall X\ A\ B\ C.\ \mathtt{arrow}\ X\ B\ (A \setminus C) \Rightarrow \mathtt{arrow}\ X\ (A \cdot B)\ C) \land \\
\quad (\forall X\ A\ B\ C.\ \mathtt{arrow}\ X\ A\ B \land \mathtt{arrow}\ X\ B\ C \Rightarrow \mathtt{arrow}\ X\ A\ C) \land \\
\quad \forall X\ A\ B.\ X\ A\ B \Rightarrow \mathtt{arrow}\ X\ A\ B
\end{array}$$

`arrow_strongind:`

$$\begin{array}{l}
\vdash (\forall X\ A.\ arrow'\ X\ A\ A) \land \\
\quad (\forall X\ A\ B\ C. \\
\qquad \mathtt{arrow}\ X\ (A \cdot B)\ C \land arrow'\ X\ (A \cdot B)\ C \Rightarrow \\
\qquad arrow'\ X\ A\ (C\ /\ B)) \land \\
\quad (\forall X\ A\ B\ C.
\end{array}$$

```
      arrow X A (C / B) ∧ arrow′ X A (C / B) ⇒
      arrow′ X (A · B) C) ∧
   (∀X A B C.
      arrow X (A · B) C ∧ arrow′ X (A · B) C ⇒
      arrow′ X B (A \ C)) ∧
   (∀X A B C.
      arrow X B (A \ C) ∧ arrow′ X B (A \ C) ⇒
      arrow′ X (A · B) C) ∧
   (∀X A B C.
      arrow X A B ∧ arrow′ X A B ∧ arrow X B C ∧ arrow′ X B C ⇒
      arrow′ X A C) ∧ (∀X A B. X A B ⇒ arrow′ X A B) ⇒
   ∀a0 a1 a2. arrow a0 a1 a2 ⇒ arrow′ a0 a1 a2
arrow_ind:
⊢ (∀X A. arrow′ X A A) ∧
   (∀X A B C. arrow′ X (A · B) C ⇒ arrow′ X A (C / B)) ∧
   (∀X A B C. arrow′ X A (C / B) ⇒ arrow′ X (A · B) C) ∧
   (∀X A B C. arrow′ X (A · B) C ⇒ arrow′ X B (A \ C)) ∧
   (∀X A B C. arrow′ X B (A \ C) ⇒ arrow′ X (A · B) C) ∧
   (∀X A B C. arrow′ X A B ∧ arrow′ X B C ⇒ arrow′ X A C) ∧
   (∀X A B. X A B ⇒ arrow′ X A B) ⇒
   ∀a0 a1 a2. arrow a0 a1 a2 ⇒ arrow′ a0 a1 a2
arrow_cases:
⊢ arrow a0 a1 a2 ⇔
   (a2 = a1) ∨ (∃B C. (a2 = C / B) ∧ arrow a0 (a1 · B) C) ∨
   (∃A B. (a1 = A · B) ∧ arrow a0 A (a2 / B)) ∨
   (∃A C. (a2 = A \ C) ∧ arrow a0 (A · a1) C) ∨
   (∃A B. (a1 = A · B) ∧ arrow a0 B (A \ a2)) ∨
   (∃B. arrow a0 a1 B ∧ arrow a0 B a2) ∨ a0 a1 a2
```

With all above theorems, any theorem in which the relation `arrow` is used, can be handled by combining above theorems with other related theorems.

Here is the essential differences between Coq and HOL that we have observed on inductive relations: In HOL, terms like `arrow` $X\ A\ B$ has type `bool`; while in Coq, its type is `Set`.

Here is the consequence: in HOL, in terms like `arrow` $X\ A\ B\ \wedge$ `arrow` $X\ B\ C$ or `arrow` $X\ A\ B \Rightarrow$ `arrow` $X\ A$ they're normal logical connectives between boolean values (first is "and", second is "implies"). But in Coq, logical connectives never appears between two `Set`s. Instead, theorems like `A / B ==> C` were always represented as `A -> B -> C` in which `->` serves as logical implication but actually has more complex meanings.

6.2 Further on logical connectives

In HOL, basic Boolean connectives and first-order logic quantifiers like "forall", "exists", "and", "or", "not" and even "true", "false", are all defined as λ terms:

```
⊢ T ⇔ ((λx. x) = (λx. x))
⊢ (∀) = (λP. P = (λx. T))
⊢ (∃) = (λP. P ((ε) P))
⊢ (∧) = (λt1 t2. ∀t. (t1 ⇒ t2 ⇒ t) ⇒ t)
⊢ (∨) = (λt1 t2. ∀t. (t1 ⇒ t) ⇒ (t2 ⇒ t) ⇒ t)
⊢ F ⇔ ∀t. t
⊢ (¬) = (λt. t ⇒ F)
⊢ (∃!) = (λP. (∃) P ∧ ∀x y. P x ∧ P y ⇒ (x = y))
```

Since they're all λ terms, any facts about their relationship can be proved within the framework of λ-calculus, using β-reductions and other basic deduction rules. That's so clear!

While in Coq, it's surprised to notice that, the quantifier `forall` is something quite primitive: it's a keyword in Coq. While there's no `exists` keyword at all. And to express the existence of something in any theorem, user must write it as a lambda function. For instance, our `replace_inv2` theorem in HOL contains two existence quantifier variables:

```
⊢ replace (Comma Gamma1 Gamma2) Gamma′ (OneForm X) Delta ⇒
  (∃ G.
     (Gamma′ = Comma G Gamma2) ∧
     replace Gamma1 G (OneForm X) Delta) ∨
  ∃ G.
    (Gamma′ = Comma Gamma1 G) ∧
    replace Gamma2 G (OneForm X) Delta
```

The meaning of above theorem is quite clear just by reading it. While in Coq, the same theorem must be expressed in this strange way:

```
Lemma replace_inv2 :
 forall (Gamma1 Gamma2 Gamma' Delta : Term Atoms) (X : Form Atoms),
 replace (Comma Gamma1 Gamma2) Gamma' (OneForm X) Delta ->
 sigS
   (fun Gamma'1 : Term Atoms =>
    {x_ : replace Gamma1 Gamma'1 (OneForm X) Delta |
    Gamma' = Comma Gamma'1 Gamma2}) +
 sigS
   (fun Gamma'2 : Term Atoms =>
    {x_ : replace Gamma2 Gamma'2 (OneForm X) Delta |
    Gamma' = Comma Gamma1 Gamma'2}).
```

please notice that, how logical "and" and "or" must be expressed as `|` and `+` between `Sets` in Coq. The author hope these examples could convince the readers that, Coq is very unnatural for representing logical theorems.

6.3 On Coq's built-in supports of proofs

In previous section, we have mentioned that, Coq has built-in supports on "proofs". This fact seems coming from the fact that, in Coq, each logical theorems has essentially the type `Set` which is indeed a mathematical set, and each element in such sets is one possible "proof" for that theorem! This is really a convenient feature when people need to prove results about proof themselves, but the drawback is, most of such theorems has large amount of variables.

For instance, our last theorem in `CutFreeTheory` is `subFormulaProperty`, which has the following representation in HOL:

```
⊢ subProof q p ⇒
  extensionSub (exten p) ⇒
  CutFreeProof p ⇒
  ∀ x.
    subFormTerm x (prems q) ∨ subFormula x (concl q) ⇒
    subFormTerm x (prems p) ∨ subFormula x (concl p)
```

This theorem was actually ported from Coq, from the following theorem:

```
Lemma subFormulaProperty :
 forall (Atoms : Set) (Gamma1 Gamma2 : Term Atoms)
   (B C x : Form Atoms) (E : gentzen_extension)
   (p : gentzenSequent E Gamma1 B) (q : gentzenSequent E Gamma2 C),
 extensionSub Atoms E ->
 subProof q p ->
 CutFreeProof p ->
 subFormTerm x Gamma2 \/ subFormula x C ->
 subFormTerm x Gamma1 \/ subFormula x B.
```

Now let's count how many variables are used in above theorem in Coq: `Atoms`, `Gamma1`, `Gamma2`, `B`, `C`, `x`, `E`, `p`, `q`, 9 variables totally. While in the HOL version, 3 variables are just enough (two "proofs" plus one Term).

Once we have proved above HOL theorem, we can also easily prove the following theorem which has the same amount of variables as Coq:

```
⊢ (p = Der (Sequent E Gamma1 B) _ _) ⇒
  (q = Der (Sequent E Gamma2 C) _ _) ⇒
  extensionSub E ⇒
  subProof q p ⇒
  CutFreeProof p ⇒
  subFormTerm x Gamma2 ∨ subFormula x C ⇒
  subFormTerm x Gamma1 ∨ subFormula x B
```

but the other direction is not easy (maybe just impossible): if we have already this last theorem in HOL, no way to get the previous theorem with only 3 variables.

Nevertheless, for Coq's built-in supports of proof, HOL users can prove the same theorems, although they have to invent the concept and structure of "proofs" from almost ground, and case by case. But the author thinks, it's quite fair to say that, all theorems provers have exactly the same set of theorems that they're capable to prove. So the remain question is how to choose between them. Most of time, it's just a matter of personal preferences, experiences and environment requirements (e.g. when you were doing formalization studies in France, you have to use Coq and OCaml, which are both invented by Franch people).

7 Future directions

From the view of theorem proving, the following theorems are worth to prove in the future:

1. Cut-elimination theorem for Lambek Calculus with arbitrary sequent extensions.
2. Lambek calculus **L** is context-free. [11]

The cut-elimination theorem can be proved directly using the existing framework in our CutFreeTheory, while for the second goal, a full treatment of Lambek Calculus in model-theoretic approach with many new foundamental definitions and theorems must be done first.

From the view of language parsing, the following goals remain to be finished:

1. Implement an automatic proof searching algorithm for Sequent Calculus as a HOL tactical.
2. Implement the same algorithm for generating first-class proof trees (Dertree).
3. Implement a language parser as ML functions which generates both parsing trees and validation theorems.
4. Design a lexicon data structure for holding large amount of words, each with more than one categories.
5. Design and implement machine learning algorithms to build a large enough lexicon for Italian language.

8 Conclusions

In this project, we have implemented a rather complete proof-theoretical formalization of Lambek Calculus (non-associative, with arbitrary extensions).

The current status is enough as a tool kit for manually proving category theorems in three deduction systems of Lambek Calculus: Axiomatic Syntactic Calculus, Natural Deduction (in Gentzen style) and Gentzen's Sequent Calculus. It can also be considered as a base framework for further formalization of more deep theorems of Lambek Calculus, e.g. the cut-elimination theorem of Gentzen's Sequent Calculus of Lambek Calculus.

This work is based on the Lambek Calculus formalization in Coq, by Houda ANOUN and Pierre Casteran in 2002-2003. For the modules that we have migrated from Coq, we improved definitions and proved many new theorems. For the proof-theoretic formalization of Sequent Calculus proofs, our work (first-class proof trees in HOL, including related derivation definitions and theorems) is completely new.

Thanks to Prof. Fabio Tambrini, who has introduced Lambek Calculus to the author in his NLP course at University of Bologna.

References

1. Chomsky, N.: On certain formal properties of grammars. Information and Computation **2** (1959) 137–167
2. Chomsky, N.: Syntactic Structures. Walter de Gruyter (January 2002)

3. Napoli, D.J., Burzio, L.: Italian Syntax: A Government-Binding Approach. Language **64**(1) (1988) 130
4. Ajdukiewicz, K.: Syntactic Connexion (1936). In: The Scientific World-Perspective and Other Essays, 1931–1963. Springer Netherlands, Dordrecht (1978) 118–139
5. Bar-Hillel, Y.: A quasi-arithmetical notation for syntactic description. Language **29**(1) (1953) 47
6. Moot, R., Retore, C.: The Logic of Categorial Grammars. Volume 6850 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
7. Lambek, J.: The mathematics of sentence structure. The American Mathematical Monthly **65**(3) (1958) 154–170
8. Cohen, J.M.: The equivalence of two concepts of categorial grammar. Information and Control **10**(5) (1967) 475–484
9. Pentus, M.: Lambek calculus is NP-complete. Theoretical Computer Science **357**(1-3) (July 2006) 186–201
10. Pentus, M.: Lambek calculus is L-complete. Institute for Logic (1993)
11. Pentus, M.: Lambek grammars are context free. In: Eighth Annual IEEE Symposium on Logic in Computer Science (Montreal, PQ, 1993). IEEE Comput. Soc. Press, Los Alamitos, CA (1993) 429–433
12. Lambek, J.: On the calculus of syntactic types. Proceedings of Symposia in Applied Mathematics **12** (1961) 166–178
13. Melham, T.F.: A Package for Inductive Relation Definitions in HOL. (January 2017) 1–10
14. Prawitz, D.: Natural deduction. A proof-theoretical study. Acta Universitatis Stockholmiensis. Stockholm Studies in Philosophy, No. 3. Almqvist & Wiksell, Stockholm (1965)
15. Gentzen, G.: Untersuchungen über das logische Schlie en. I. Mathematische Zeitschrift **39**(1) (1935) 176–210
16. Gentzen, G.: Untersuchungen über das logische Schlie en. II. Mathematische Zeitschrift **39**(1) (1935) 405–431
17. Dawson, J.E., Goré, R.: Machine-checked Cut-elimination for Display Logic. (2006)