Typelogical Proof Nets in Python

Graphical Lambek-Grishin Calculus

Sjoerd Dost
3481603

March 14, 2013

Bachelor thesis Cognitive Artificial Intelligence

Utrecht University
Supervisor: Prof. dr. Michael Moortgat

Contents

Chapter 1

Introduction

1.1 A Computational Approach to Natural Language

In Artificial Intelligence one of the main topics is natural language processing. A key issue is the balance between expressivity and complexity. We would like to formalise natural language for use by computers, in such a way that the system is expressive enough and not too complex. The right trade-off between the two is itself a delicate field of study.

The more expressive a language is, the more sentences can be formulated with it. This is a rough interpretation: in formal language we look at the different syntactic patterns that can be expressed. We must keep in mind though that the more expressive a language is, the more difficult it can be to understand it. If we want to add extra expressivity to a language, we will eventually need to add more complexity. This is the essential trade-off between expressivity and complexity.

So what do we know about the complexity of natural language? The general consensus is that natural language should be polynomially parsable. Parsing a sentence should not possibly take extremely long, relative to the length of the sentence. A model for natural language should adhere to this restriction to be feasible, in accordance with psychological research of human language use. In 1956, Noam Chomsky introduced a hierarchy of formal languages [1]. This hierarchy orders formal languages by their computational complexity. Starting at regular languages and growing all the way to the recursively enumerable languages, the Chomsky hierarchy has been expanded on for more than 50 years now. We will look for the computational complexity of natural language in this hierarchy.

In this thesis we show a logical approximation of language and a system that can work with it. The approximation is a calculus with certain rules: the Lambek-Grishin calculus. The system is a theorem prover: a program that can prove whether the calculus accepts a certain 'sentence'. By building a prover for the calculus we show that this is an approximation we can actually use. We introduce the calculus in a hierarchy of complexity. We then show the theory underlying the theorem prover. Finally in the appendix we give the entire source code of the prover, which can also be found at `https://github.com/deosjr/Scriptie`.

1.2 The Chomsky Hierarchy

First we take a look at formal languages and their relation to natural language. Instead of looking at individual languages we look at several classes of languages. All languages in such a class are of equal computational complexity. We start by looking at context-free languages, followed by context-sensitive languages. Both are defined in [1]. After concluding that natural language is not best described by either, we look at an intermediate area in the hierarchy. The aim is to find a class of language that corresponds closely to natural language in terms of expressivity and computational complexity. The structure of this overview very roughly corresponds to the chronological order of research in this field. See [6] for an extended overview.

1.2.1 Context-free languages

The first area of the hierarchy to be considered is the context-free (CF) area. Context-free languages can describe many syntactic patterns found in natural language. They can be described using context-free grammars (CFGs) that are easily definable. When crossing dependencies were identified in some natural languages it became apparent that CFGs are not powerful enough to capture the entirety of natural language. These crossing dependencies, found in Dutch but most convincingly shown in Swiss German [15], can be shown to be beyond CFG.

...das	met d'chind	em Hans es huus	lönd	hälfe aastriiche
...that we	the children Hans	the house	let	help paint

'...that we let the children help Hans paint the house.'

Since CSG's can't describe these dependencies, natural language is shown to be more expressive than CFG's can ever be. We have to search higher up in the Chomsky hierarchy.

1.2.2 Context-sensitive languages

The next step in the hierarchy as originally stated is that of the context-sensitive (CS) languages. Whilst crossing dependencies can be analysed with context-sensitive grammars (CSGs), some structures definable using CSGs have convincingly been shown to be beyond natural language. For example, the language $\{a^{2^n} | n \in \mathbb{N}\}$ defines a pattern that grows exponentially, which is something we have not found in natural language. CS is therefore too expressive to approximate natural language with. Context-sensitive languages are also not all polynomially parsable. This means CS is too complex as well and definately not a good approximation.

We have found that context-free grammars are too weak to model natural language with, and context-sensitive grammars are too strong. The next logical step is to define an area in between; a class of languages that is stronger than context-free but weaker than context-sensitive.

1.2.3 Mildly context-sensitive languages

In 1985 Aravind Joshi characterised a class of languages between context-free and context-sensitive, calling it mildly context-sensitive (MCS). [5]. It is defined as follows (taken from [6]):

Definition 1.1 ***Mild context-sensitivity***

1. *A set $\mathcal{L}$ of languages is* mildly context-sensitive *iff*
   (a) *$\mathcal{L}$ contains all context-free languages*
   (b) *$\mathcal{L}$ can describe cross-serial dependencies:*
   There is an $n \geq 2$ such that $\{w^k | w \in T^*\} \in \mathcal{L}$ for all $k \leq n$.
   (c) *The languages in $\mathcal{L}$ are polynomially parsable, i.e., $\mathcal{L} \subset PTIME$.*
   (d) *The languages in $\mathcal{L}$ have the* constant growth property.
2. *A formalism F is* mildly context-sensitive *iff the set $\{L | L = L(G)$ for some $G \in F\}$ is mildly context-sensitive.*

The first constraint (a) tells us that the class of mildly context-sensitive languages includes that of the context-free languages. The second shows what we want to capture beyond context-free: crossing dependencies. Note that crossing dependencies can only be captured up to a certain degree: not all dependencies can be motivated from the study of natural languages. The third constraint captures our intuition that natural languages should not be too hard to parse. This also places mild context-sensitive languages in a subclass of the context-sensitive, since the decidability problem for CSGs is PSPACE complete. For a language to have the bounded growth property means the length of words in the language grows linearly, when ordered by length.

As we can see mild context-sensitivity is precisely defined as the area in which we expect to find natural language. The hypothesis is that the MCS class would be appropriate for the analysis of the syntactic patterns occurring in natural language. Mildly context-sensitive languages are expressive enough (a, b) and not too complex (c, d). Formalisms in MCS include Tree-adjoining grammar (TAG), Multiple Context-free grammar (MCFG) and Combinatorial Categorial grammar (CCG).

1.3 Typelogical Grammar

In this thesis we study a formalism with a lower bound in the mildly context-sensitive area, Lambek-Grishin calculus (LG). It is a categorial grammar in the typelogical framework. The typelogical perspective allows us to import techniques from logical proof theory, notably proof nets. The Curry-Howard correspondence gives us an interface between syntax and semantics. A theorem prover for Lambek-Grishin calculus had not yet been implemented, to our knowledge. In 2002 Richard Moot introduced Grail, a prover in Prolog for multimodal Lambek calculus. An extension for LG was given in [13], but was not implemented. For more on this interactive parser, see [11]. In this thesis we give an implementation in Python for graphical LG.

We illustrate a typical categorial grammar by comparison with a context-free grammar, which is a rewrite grammar. A context-free grammar G is defined as the set $\{N, T, P, S\}$. N and T are its non-terminal and terminal symbols, respectively. We will call its terminal symbols 'words' and series of words 'sentences'. This might seem confusing as we usually use the term 'word' for what we now call a sentence. We try to be consistent in our term usage and will use the above terms more intuitively in later discussion. The set P gives us rules to rewrite a non-terminal symbol. S is a special non-terminal, the start symbol. Given a sentence $x : \{x = w_1, w_2 \ldots w_n$ with $w_i \in$T$\}$, the grammar will accept x if and only if $S \Rightarrow^* x$. That is, a sentence x is only accepted by the grammar if there is a series of rules in P that rewrites S to x. A categorial grammar G' gives us a lexicon L and inference rules R. It accepts the same sentence x if and only if $A_1, A_2 \ldots A_n \vdash s$ is provable in natural deduction using inference rules given in R. Here A_i is the type given to w_i by L and s is the type of a sentence. In general categorial grammar can prove sequents of the form $A_1, A_2 \ldots A_n \vdash B_1, B_2 \ldots B_m$. This means that given a categorial framework, providing a grammar for a certain language is only a matter of formulating the correct lexicon.

1.3.1 Lambek systems

The Lambek calculus [7] defines its types using the following atomic types and operators:

$$\textbf{Types: } A, B ::= p \mid A \otimes B \mid A/B \mid B\backslash A$$

where A and B are (possibly complex) types and p is atomic. Intuitively the operators are defined as follows: A/B is of type A if a type B can be found to the right of it. Similarly, $B\backslash A$ is of type A given a type B directly to its right. The $\otimes$ operator indicates concatenation of types, allowing types to be found next to each other to satisfy conditions for the previously named operators.

Lambek calculus provides us with the first link between categorial grammars and the Chomsky hierarchy: it is equivalent to context-free grammar. This equivalence is easily proven from CFG to Lambek grammar; equivalence in opposite direction is known as the Chomsky conjecture [2], proven by Pentus in [14]. Since Lambek-Grishin calculus is an extension of the Lambek calculus, its expressivity must be at least context-free.

LG essentially adds another set of operators which mirror the original operators of Lambek calculus. These operators adhere to the same kind of rules the originals adhere to, and the intuition for using them is the same. That is, A/B is of type A given that we find a type B concatenated with $\otimes$ to the right of it. $B \obslash A$ is of type A if a type B is concatenated via $\oplus$ to its left.

$$\textbf{Types: } A, B ::= p \mid A \otimes B \mid A/B \mid B\backslash A \mid A \oplus B \mid A \oslash B \mid B \obslash A$$

The extra expressivity comes from its extra inference rules (besides those that are dual to the original rules). These so-called *linear distributivity principles* or *interaction rules* translate between the two sets of operators. We have several options to present LG's full rule system. Natural deduction is not a good option since it is not suited for automation. To use the calculus for automatic inference, we choose a sequent calculus approach, since it can be read purely top-down. Sequent calculus' decidability makes it a better choice for automatic proving.

We present LG's inference rules using the notation of [9]. It gives LG in a display logic style (calling it sLG), divided in structural and logical rules (Figures 1.1 and 1.2). These rules will be the foundation of our graphical calculus as well: graphical LG is mostly a translation of these rules to graphs. A translation embedding Tree-adjoining grammars (TAGs) in LG has been shown by Richard Moot in [12]. Since TAG is a mild context-sensitive formalism this places LG's lower bound in our area of interest, instead of at the context-free hierarchy. The upper bound for expressivity of LG is still unknown. For discussion see [8].

$$\frac{A \cdot \$ \cdot B \Rightarrow Y}{A\$B \Rightarrow Y}\ \$L \quad \$ \in \{\otimes, \obslash, \oslash\} \qquad \frac{X \Rightarrow A \cdot \# \cdot B}{X \Rightarrow A\#B}\ \#R \quad \# \in \{\oplus, \backslash, /\}$$

$$\frac{X \Rightarrow A \quad Y \Rightarrow B}{X \cdot \otimes \cdot Y \Rightarrow A \otimes B}\ \otimes R \qquad \frac{A \Rightarrow X \quad B \Rightarrow Y}{A \oplus B \Rightarrow X \cdot \oplus \cdot Y}\ \oplus L$$

$$\frac{X \Rightarrow A \quad B \Rightarrow Y}{A \backslash B \Rightarrow X \cdot \backslash \cdot Y}\ \backslash L \qquad \frac{X \Rightarrow A \quad B \Rightarrow Y}{X \cdot \oslash \cdot Y \Rightarrow A \oslash B}\ \oslash R$$

$$\frac{X \Rightarrow A \quad B \Rightarrow Y}{B / A \Rightarrow Y \cdot / \cdot X}\ /L \qquad \frac{X \Rightarrow A \quad B \Rightarrow Y}{Y \cdot \obslash \cdot X \Rightarrow B \obslash A}\ \obslash R$$

Figure 1.1: Logical rules for LG

$$\frac{}{A \Rightarrow A}\ \text{Ax} \qquad \frac{X \Rightarrow A \quad A \Rightarrow Y}{X \Rightarrow Y}\ \text{Cut}$$

$$\frac{\frac{X \Rightarrow Z \cdot / \cdot Y}{X \cdot \otimes \cdot Y \Rightarrow Z}\ rp}{Y \Rightarrow X \cdot \backslash \cdot Z}\ rp \qquad \frac{\frac{Y \cdot \obslash \cdot Z \Rightarrow X}{Z \Rightarrow Y \cdot \oplus \cdot X}\ drp}{Z \cdot \oslash \cdot X \Rightarrow Y}\ drp$$

$$\frac{X \cdot \otimes \cdot Y \Rightarrow Z \cdot \oplus \cdot W}{Z \cdot \obslash \cdot X \Rightarrow W \cdot / \cdot Y}\ G1 \qquad \frac{X \cdot \otimes \cdot Y \Rightarrow Z \cdot \oplus \cdot W}{Y \cdot \oslash \cdot W \Rightarrow X \cdot \backslash \cdot Z}\ G3$$

$$\frac{X \cdot \otimes \cdot Y \Rightarrow Z \cdot \oplus \cdot W}{Z \cdot \obslash \cdot Y \Rightarrow X \cdot \backslash \cdot W}\ G2 \qquad \frac{X \cdot \otimes \cdot Y \Rightarrow Z \cdot \oplus \cdot W}{X \cdot \oslash \cdot W \Rightarrow Z \cdot / \cdot Y}\ G4$$

Figure 1.2: Structural rules for LG

1.4 Spurious Ambiguity

This concludes the introduction. The next chapter handles graphical calculus for LG, which is the main subject of this thesis. Switching from sequent to graphical calculus has various reasons. However, we have just motivated the use of sequent calculus instead of natural deduction. Although sequent calculus is indeed easier to use for automation, it does not have a feature natural deduction has: a single derivation per interpretation of a sequent. This means that sequent calculus can allow multiple derivations for a single interpretation of a sequent. This is called *spurious ambiguity*. Compare Figures 1.3 and 1.4. Graphical calculus seeks to solve these problems by giving a method of derivation that rewrites graphs and is free of spurious ambiguity. See Figure 1.5 for an example.

$$\frac{\frac{}{np \vdash np}\ Ax \qquad \frac{\frac{}{(np\backslash s)/np \vdash (np\backslash s)/np}\ Ax \qquad \frac{\frac{}{np/n \vdash np/n}\ Ax \qquad \frac{}{n \vdash n}\ Ax}{(np/n) \otimes n \vdash np}\ /E}{((np\backslash s)/np) \otimes ((np/n) \otimes n) \vdash np\backslash s}\ /E}{np \otimes (((np\backslash s)/np) \otimes ((np/n) \otimes n)) \vdash s}\ \backslash E$$

Figure 1.3: Natural deduction proof for $np \otimes (((np\backslash s)/np) \otimes ((np/n) \otimes n)) \vdash s$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np \Rightarrow np}Ax \quad \dfrac{}{n \Rightarrow n}Ax}{np/n \Rightarrow np \cdot / \cdot n}/L}{(np/n) \cdot\otimes\cdot n \Rightarrow np}rp}{(np/n)\otimes n \Rightarrow np}\otimes L \quad \dfrac{\dfrac{}{np \Rightarrow np}Ax \quad \dfrac{}{s \Rightarrow s}Ax}{np\backslash s \Rightarrow np\cdot\backslash\cdot s}\backslash L}{(np\backslash s)/np \Rightarrow (np\cdot\backslash\cdot s)\cdot/\cdot((np/n)\otimes n)}/L}{((np\backslash s)/np)\cdot\otimes\cdot((np/n)\otimes n) \Rightarrow np\cdot\backslash\cdot s}rp}{((np\backslash s)/np)\otimes((np/n)\otimes n) \Rightarrow np\cdot\backslash\cdot s}\otimes L}{np\cdot\otimes\cdot(((np\backslash s)/np)\otimes((np/n)\otimes n)) \Rightarrow s}rp}{np\otimes(((np\backslash s)/np)\otimes((np/n)\otimes n)) \Rightarrow s}\otimes L$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{n \Rightarrow n}Ax \quad \dfrac{\dfrac{\dfrac{\dfrac{}{np\Rightarrow np}Ax \quad \dfrac{\dfrac{}{np\Rightarrow np}Ax \quad \dfrac{}{s\Rightarrow s}Ax}{np\backslash s \Rightarrow np\cdot\backslash\cdot s}\backslash}{(np\backslash s)/np \Rightarrow (np\cdot\backslash\cdot s)\cdot/\cdot np}/L}{((np\backslash s)/np)\cdot\otimes\cdot np \Rightarrow np\cdot\backslash\cdot s}rp}{np \Rightarrow ((np\backslash s)/np)\cdot\backslash\cdot(np\cdot\backslash\cdot s)}rp}{np/n \Rightarrow (((np\backslash s)/np)\cdot\backslash\cdot(np\cdot\backslash\cdot s))\cdot/\cdot n}/L}{(np/n)\cdot\otimes\cdot n \Rightarrow ((np\backslash s)/np)\cdot\backslash\cdot(np\cdot\backslash\cdot s)}rp}{(np/n)\otimes n \Rightarrow ((np\backslash s)/np)\cdot\backslash\cdot(np\cdot\backslash\cdot s)}\otimes L}{((np\backslash s)/np)\cdot\otimes\cdot((np/n)\otimes n) \Rightarrow np\cdot\backslash\cdot s}rp}{((np\backslash s)/np)\otimes((np/n)\otimes n) \Rightarrow np\cdot\backslash\cdot s}\otimes L}{np\cdot\otimes\cdot(((np\backslash s)/np)\otimes((np/n)\otimes n)) \Rightarrow s}rp}{np\otimes(((np\backslash s)/np)\otimes((np/n)\otimes n)) \Rightarrow s}\otimes L$$

Figure 1.4: Two sequent derivations for $np \otimes (((np\backslash s)/np) \otimes ((np/n) \otimes n)) \Rightarrow s$

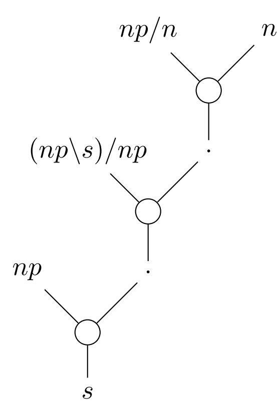


Figure 1.5: $np \otimes (((np\backslash s)/np) \otimes ((np/n) \otimes n)) \Rightarrow s$

Chapter 2

A Graphical Calculus

2.1 Introduction

Graphical calculus for typelogical grammars is based on so-called *proof nets*. Proof nets have been developed to hide a lot of structural rules and to bring back focus on the derivation(s) natural deduction allows for the sequent. They first appeared in 1987 when Jean-Yves Girard introduced proof nets for linear logic [4]. In [13] Richard Moot gives a great overview of extended Lambek calculus in both sequent and graphical form. This system was adapted for Lambek-Grishin calculus in 2012 by Michael Moortgat and Richard Moot [10], in which they add semantics as well. This chapter shows the translation from sequent to graphical calculus. It mostly reiterates from [10], but is essential for understanding the following chapters. First we define the building blocks of our graphs and then we introduce rules for rewriting them.

2.2 Graphs

Proof nets allow us to use graph theory to produce sequent proofs. In order to do so our lexicon cannot just assign types to words but needs to assign graphs. Once words are graphs we can treat them in a graph-theoretical manner and 'compile away' most of the abstract rewriting found in sequent calculus. We start by translating our inference rules to graphs. Logical rules for our operators will define the translation of the operators themselves. Note that we use hypergraphs, graphs with edges that can connect multiple vertices. To be specific, our proof nets will be 3-hypergraphs, in which all edges connect exactly three vertices. Direction is of importance in our graphs, making them harder to draw on the Euclidean plane. For a discussion on drawing these graphs see [3].

The vertices will be labeled with formulas and have two points of connection: up and down. Relative positioning has the following meaning between connected structures A and B: If A is above B, then A is a hypothesis of B. Likewise, if A is below B, then A is a conclusion of B. Although edges simply connect vertices we talk about hypotheses and conclusions of the edge, since it is central in our translation. A vertex that is not the conclusion of anything is called a hypothesis; a vertex that is not the hypothesis of anything is called a conclusion. A vertex connected on both sides is an internal node and has no formula decoration.

Edges are drawn as big circles, which are not to be confused with vertices. They are a direct translation of the logical rules of LG. We distinguish between rules with one premise and rules with two premises. The first are called *cotensors* and are filled in black. The second are called *tensors* and are left white. We will sometimes use the term *tensor link* instead of edge. Note that for Grishin's operators we reverse the premises and conclusions, leading to tensors with one and cotensors with two hypotheses.

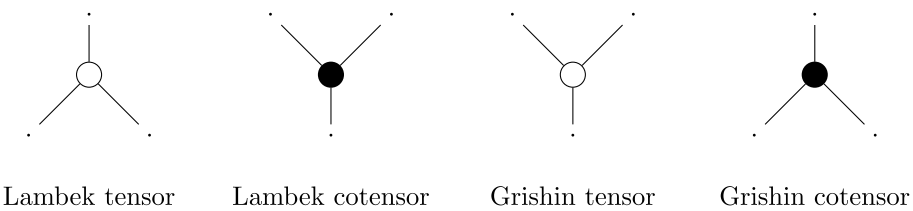


Figure 2.1: Edge layout

Using the graphs in Figure 2.3 we translate types to graphs by 'unfolding' them. We identify the main operator and pick the corresponding edge (depending on whether the formula is a hypothesis or a conclusion). The edge is connected to vertices labeled as in Figure 2.3. A and B are respectively the formulas left and right of the main operator. If A and/or B are complex, we now recursively unfold them. The resulting structure is connected to the main formula via the first edge. The total will therefore always be a connected structure.

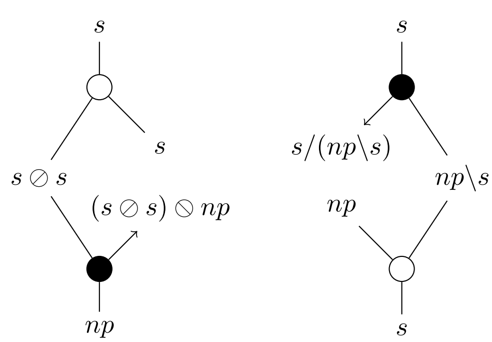


Figure 2.2: Lexical unfolding

We start without a garantee that the sequent is provable. In this case we talk about a proof structure or candidate proof net. We define the proof structure now and leave the definition for the proof net for later. Assume for now that a proof net is a proof structure corresponding to a provable sequent.

Definition 2.1 ***Proof Structure***

1. *A* proof structure *is a 3-hypergraph $\langle V, E\rangle$ such that V is a non-empty set of vertices which can at most once be the hypothesis and at most once be the conclusion of an edge, and E is a set of non-empty subsets of V called edges, as described in Figure 2.3.*

2. *A structure with hypotheses $H_1, \ldots, H_m$ and conclusions $C_1, \ldots, C_n$ is a proof structure of $H_1, \ldots, H_m \Rightarrow C_1, \ldots, C_n$.*

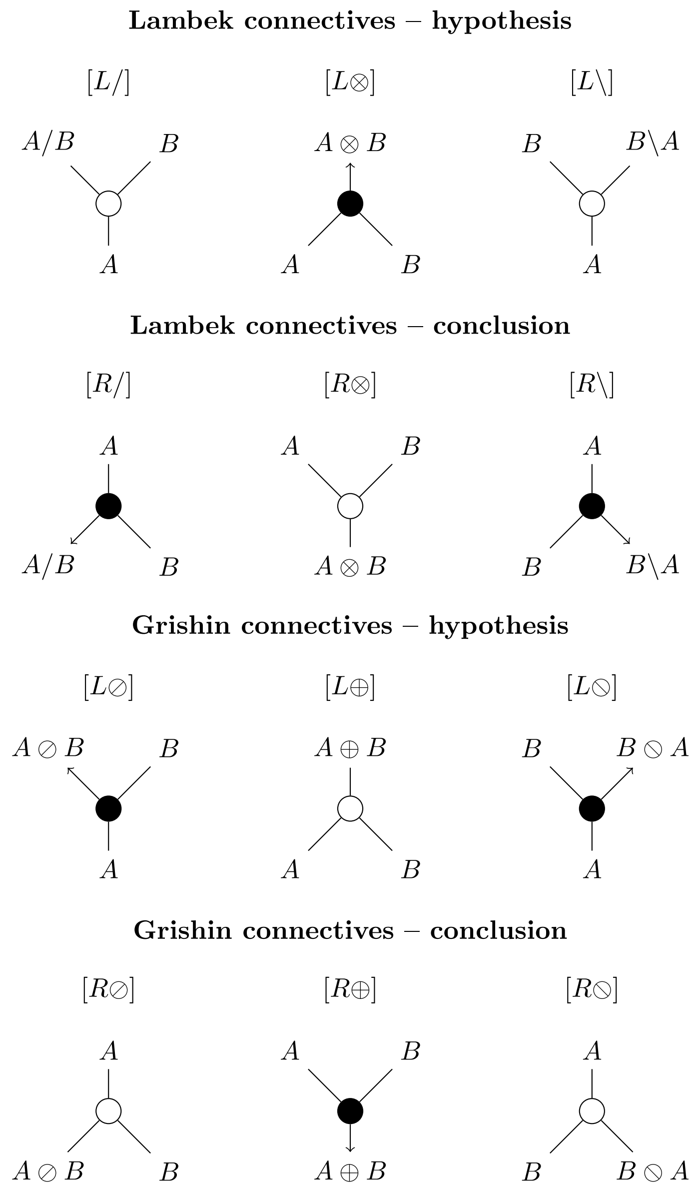


Figure 2.3: Graphical translation of LG's logical rules

Definition 2.2 ***Module***

A module *is a proof structure that is the direct result of lexical unfolding of a single formula.*

We start proving a sequent by unfolding all formulas. If we consider the set of modules corresponding to all formulas in a sequent as a (non-connected) proof structure, we see that this is not yet a proof structure of the sequent. This can easily be verified by looking at Figure 2.3. We need to identify atomic formulas to get a correctly corresponding proof structure. This is done by linking an atomic hypothesis to an atomic conclusion with the same formula decoration. When repeated until no atomic formulas remain the result will be a proof structure of the given sequent. Note that sometimes multiple linkings are possible. In this case each is a candidate proof net.

2.3 Correctness

So far we have only partially made the switch to graphical calculus. We need more than just the logical rules. To complete the translation, we have the following rules, which dictate ways of rewriting the graph. These rules are instrumental in actually proving a sequent. They allow us to rewrite proof structures to proof nets. We now define a proof net, in terms of rules to be explained immediately afterwards.

Definition 2.3 ***Proof Net***

A proof net *is a proof structure that can be contracted to an acyclic, connected structure (a* tree*) containing no cotensors, using only the rules of contraction and interaction as described below.*

Note that we can omit the labeling of internal vertices. In such a case we have an *abstract proof structure*. All rules work on abstract proof structures. Contracting a proof structure and thereby showing it is a proof net equals a correct derivation. Proof of this fundamental principle in the graphical calculus for LG (stated in Theorem 2.4) can be found in [12].

Theorem 2.4 *A proof structure P is a proof net – that is, P converts to a tree T – iff there is a sequent proof of T.*

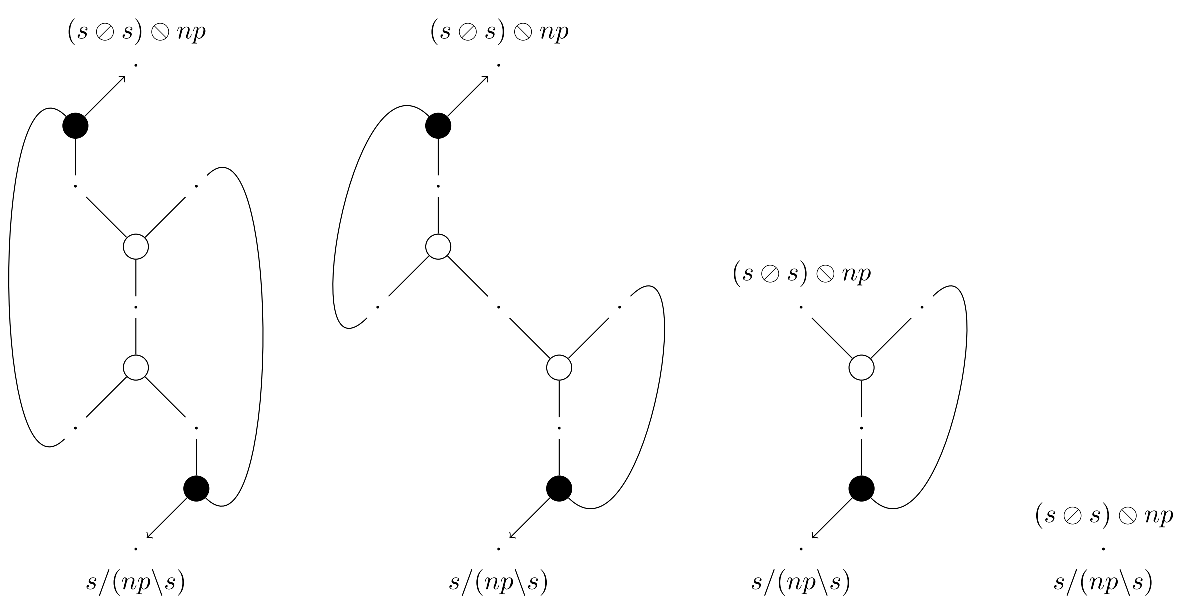


Figure 2.4: $(s \oslash s) \obslash np \Rightarrow s/(np\backslash s)$

2.3.1 Contraction

First we will introduce a set of rules for removing cotensors from our proof structure. These rules are the *contraction rules*. They are abstract proof structures that can contract to a single vertex. These structures can be generalized and can contract even when found as part of a larger structure. In Figure 2.5, showing all six of these structures, the nets are labeled with H and C. These are not necessarily formula labelings: they are possibly structures (so a vertex labeled H in this figure is either internal or a hypothesis). When one of these structures can be identified it can immediately contract to a single vertex labeled H and C. This way the cotensor is removed. The final goal is of course to remove all cotensors, so that we can show the proof structure to be a proof net.

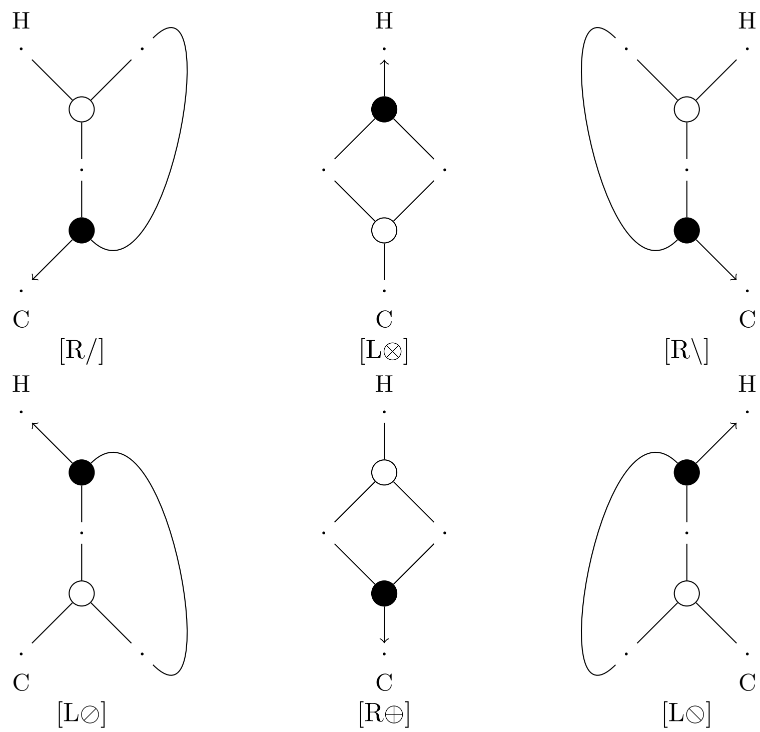


Figure 2.5: Contraction rules

2.3.2 Interaction

The *interaction rules* are ways of rewriting the graph, corresponding to Grishin's interaction principles indicated in Figure 1.2 as $G1$ through $G4$. These rules make it possible to remove cotensors (through contraction) when none of the applicable structures can be found. We rewrite the structure shown in the middle of Figure 2.6 to one of four structures as shown by the arrows. Note that this is a nondeterministic procedure: any four of these structures can be the result of rewriting the same starting structure. The hope is that through (reiterated) rewriting we find a structure on which we can apply contraction. We can generalise the use of interaction and contraction to generalised contraction principles, allowing for any number of tensors between the cotensor and tensor of the structure. After interaction we can always find a contracting configuration in those

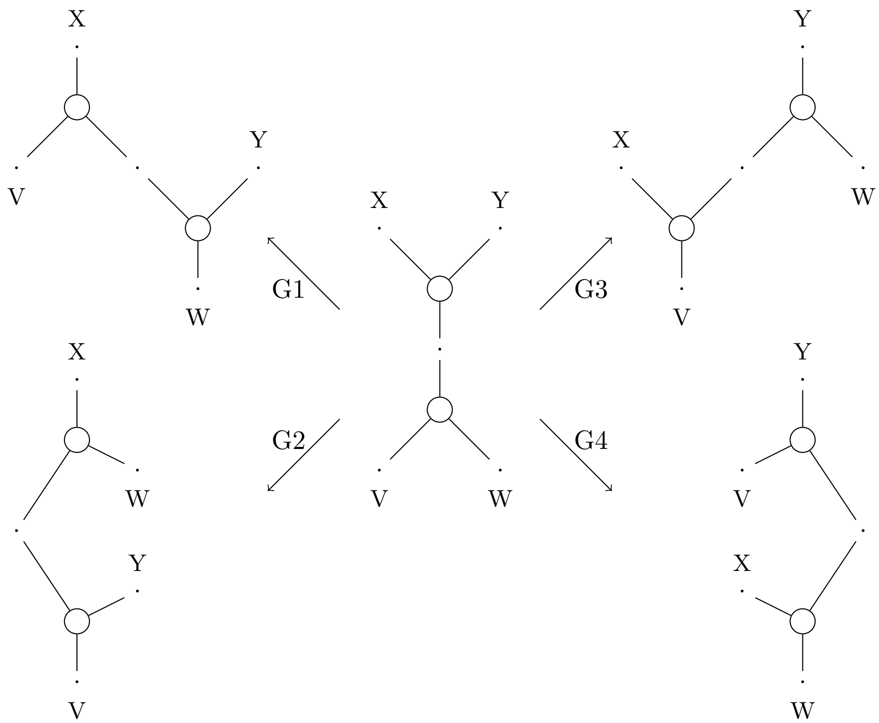


Figure 2.6: Interaction rules

cases. These generalised contractions are not shown but are elaborated upon in 4.5.2.

2.4 Example derivations

We give an example of a derivation using graphical calculus in Figure 2.4. We start with two modules as shown in Figure 2.2. These can be connected in two ways (the *np* in one way, the *s* in two, giving a total of two possibilities). The leftmost structure corresponds to the modules after binding in such a way that the derivation will succeed. Now reading from left to right, we apply interaction and contraction until we find a proof net. The first step is an interaction rule (G1), since we have no configurations for contraction. After applying this rule, we find two configurations to apply contraction on. We first apply [L⦸] and then [R/], giving us a single point. This is trivially a proof net since it contains no cotensors and is connected and acyclic.

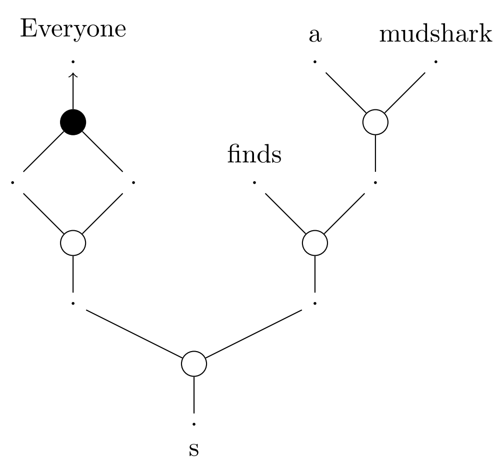
Now that we have seen all that there is to it, let's take another example. This time, we would like to illustrate graphical calculus' approach to spurious ambiguity. We take the example found in Figure 1.4 and give its accompanying proof net in Figure 1.5. It is quite trivially a cotensor-free tree. The ambiguity found in 1.4 is gone: this is the only proof net for the sequent in question. It seems that spurious ambiguity is solved. We must note, however, that our theorem prover allows another net for this sequent. This is because word order in a sentence is not preserved (see chapter 4). The net in Figure 1.5 is the only net for the sequent *with order preserved*.

Chapter 3

Nets and their interpretation

3.1 Relation to sequent proof

Let us revisit the problem of spurious ambiguity. We use the example sentence *"Everyone finds a mudshark"*, combined with a lexicon that assigns the following types to the constituent words respectively: $(np/n) \otimes n$, $(np\backslash s)/np$, np/n and n. A sentence such as this has multiple proofs in sLG, the unfocused sequent approach. The proof net approach of chapter 2 allows for a single derivation of the sequent $(np/n) \otimes n,\ (np\backslash s)/np,\ np/n,\ n \Rightarrow s$. However, we would like to see two derivations, explaining the scope difference of the two interpretations of this sentence. Is there a single mudshark that is found by everyone, or has everyone individually found a mudshark of their own?



$$(1)\ \mu\alpha.(\tfrac{x'z}{\text{subj}}.\langle x' \upharpoonleft (\tilde{\mu}x.\langle \text{det} \upharpoonleft (\tilde{\mu}y.\langle \text{tv} \upharpoonleft ((x\backslash\alpha)/y)\rangle/\text{noun})\rangle/z)\rangle)$$

$$(2)\ \mu\alpha.(\tfrac{x'z}{\text{subj}}.\langle \text{det} \upharpoonleft (\tilde{\mu}y.\langle x' \upharpoonleft (\tilde{\mu}x.\langle \text{tv} \upharpoonleft ((x\backslash\alpha)/y)\rangle/z)\rangle/\text{noun})\rangle)$$

Figure 3.1: Everyone finds a mudshark

In Figure 3.1 we show a proof net (the single net for the above sequent) and the two proof terms associated with it. Figure 3.1 is an example of the output we would like to see from a theorem prover. To avoid confusion between a as a variable and as a determiner, we use the more general "subj tv det noun" in the proof term. These proof terms are compatible with focused proof search for LG, or fLG. They are an encoding of proofs in fLG (which has less of a many-to-one attitude to proofs than sLG). We introduce these terms from a graphical point of view; instead of justifying them from fLG's inference rules, we extend our graphs so we can read these proof terms in a graph-based way.

3.1.1 Types and terms

The term language for our graphs is the same as that for fLG as found in [10]. We distinguish three different types of terms. These are *commands, contexts* and *values.* The full term language differentiates not only between input (represented as variables $x, y, z, \ldots$) and output formulas (represented as covariables $\alpha, \beta, \gamma, \ldots$), but also between these three types. Figure 3.1 gives the full term language in Backus-Naur Form, where commands are labeled c, C, values v, V and contexts e, E.

$$
\begin{array}{l}
v ::= \mu\alpha.C \mid V \;;\; V ::= x \mid v_1 \otimes v_2 \mid v \oslash e \mid e \obslash v \\
e ::= \tilde{\mu}x.C \mid E \;;\; E ::= \alpha \mid e_1 \oplus e_2 \mid v \backslash e \mid e/v \\
c ::= \langle x \upharpoonright E \rangle \mid \langle V \upharpoonright \alpha \rangle \;;\; C ::= c \mid \frac{x\ y}{z}.C \mid \frac{x\ \beta}{z}.C \mid \frac{\beta\ y}{z}.C \mid \frac{\alpha\ \beta}{\gamma}.C \mid \frac{x\ \beta}{\gamma}.C \mid \frac{\beta\ x}{\gamma}.C
\end{array}
$$

Figure 3.2: Term language

In graphical terms, we define these types as follows.

Definition 3.1 ***Value, context, command***

1. *A* value *is either:*
   - *(a) The hypothesis of a tensor*
   - *(b) The positive main formula of a tensor*
   - *(c) A starting formula as found in the sequent*
2. *A* context *is either:*
   - *(a) The conclusion of a tensor*
   - *(b) The negative main formula of a tensor*
3. *A* command *is either:*
   - *(a) The result of cutting a value against a context*
   - *(b) An extension of a command with a cotensor link*

We consider $A \otimes B, A \oslash B$ and $B \obslash A$ to be positive while $A \oplus B, A/B$ and $B\backslash A$ are negative. Atomic formulas have arbitrary polarity: their polarity can be chosen at will, though once determined we must stick with our choice for the entire derivation. A different choice for atom polarity leads to different derivations, although the derivability of a sequent does not depend on this choice.

3.2 Focused proof nets

We extend our graphical calculus in such a way that we can read the corresponding proof term(s) by traversal. Polarity must be defined in our lexicon for our atomic formulas. Complex formulas have a polarity based on their main connective. All we need to do is change the net according to the term language. We don't really change our previous approach: we only add more information to our graph.

3.2.1 Composition Graph

Since proof terms only make sense when associated with proof nets (instead of the more general proof structures), we can assume that a translation will be made from proof nets (not structures) to new nets. Proof terms are computed by a traversal on such a new net, or *composition graph*. The precise translation is defined below (see [10]).

Definition 3.2 ***Composition Graph***

Given a proof net P, the associated composition graph $\text{cg}(P)$ *is obtained as follows.*

1. *All vertices of P with formula label A are expanded into polarised* axiom links*: edges connecting two vertices with formula label A; all links are replaced by the corresponding links of Figure 3.6.*
2. *All vertices labeled with simple formula are assigned atomic terms of the correct type (variable or covariable) and all others are given a term derived from these assignments.*
3. *All axiom links connecting terms of the same type (value or context) are collapsed.*

We talk about an *initial* composition graph before and about a *reduced* composition graph after step 3. An example composition graph for a small proof net can be found in Figure 3.4.

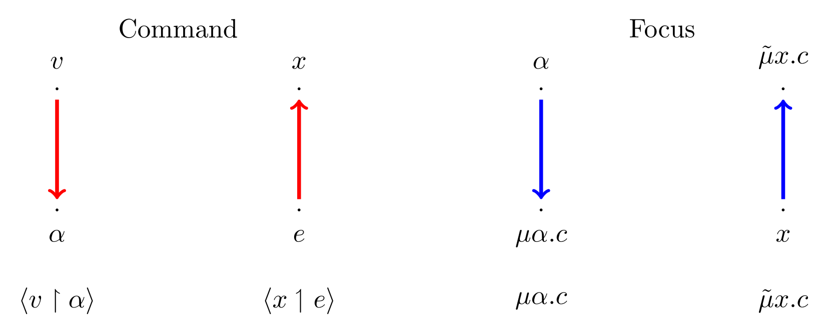


Figure 3.3: Axiom links

Before we explain the actual traversal, we define several parts of the composition graph to be able to refer to them separately. We divide axiom links in four different categories, two of which will be collapsed in step 3 as described above. The interesting cases are those that do not collapse: They link a value to a context or vice-versa. We say the one is a *command link*, the other a *focus link* (see Figure 3.3). The direction of the link is determined by polarity: an arrow from value to context indicates a positive link, the other way around needs a negative formula to function.

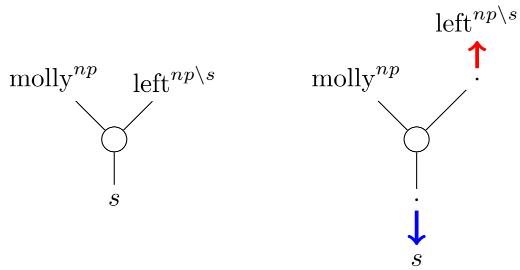


Figure 3.4: Example of producing a composition graph.

Definition 3.3 ***Component***

Given a proof net P, a component *C of P is a maximal subnet of P containing no cotensors.*

From its definition, we can easily see how to obtain a composition graph's components: we simply remove all cotensors. All remaining connected parts are the components of the graph. We have now defined all parts needed to understand term traversal.

3.2.2 Traversal

To read proof terms from focused proof nets we need a structured method of traversing the graph. The following algorithm for term traversal in focused proof nets for LG can be found in [10]. It produces a term given the composition graph *cg(P)* of a proof net P.

1. Compute all components of *cg(P)*, consisting of a set of tensor links with a single main formula. Mark all these links as visited.
2. While *cg(P)* contains unvisited tensor links do the following:
   (a) Follow an unvisited command link attached to a previously calculated maximal subnet, forming a correct command subnet.
   (b) For each cotensor that is doubly connected to the current command subnet, form a new command net including this cotensor. Repeat until no such cotensors can be found.
   (c) Follow a μ or $\tilde{\mu}$ [focus] link to a new vertex, forming a larger value or context subnet.

This algorithm produces a series of links visited, written as $c_1 - \mu_1, ...$ etc. These can be applied to create proof terms.

3.2.3 Problems

In encoding this algorithm for use in the theorem prover we have encountered several difficulties, which we will describe below. All problems described here have been encountered whilst implementing the previously described algorithm. They vary from small remarks on clarity to notes on incompleteness.

First of all, once we have chosen a subnet to start with, we must stick to that subnet for all further steps. This means that the maximal subnet in step 2a can only once be chosen: we must stick with our chosen subnet for the rest of the traversal. This restriction is small but crucial: not adhering to it leads to nonsensical terms. The algorithm also makes no mention of polarity. We can only follow a command or focus link if the polarity of the formula linked is correct. This restriction is needed to bound the number of possible terms.

Furthermore, the algorithm gives us the impression that term traversal is a sequential process, whilst it is actually parallel. As stated just before the original definition of the algorithm, [...] *instead of seeing* ρ [the reduction from proof structure to net] *as a* sequence *of reductions, we can see it as a rooted* tree *of reductions* [...] [10]. In its current state the algorithm does not tell us how to deal with a parallel situation. The algorithm should therefore be adjusted to allow this parallellism to be handled correctly. Figure 3.5 is an example: its associated proof term is of the form $A \otimes B$, where A is (a/b) and B is $(c\backslash d)$. We know there is a proof term, for it is a tautology. Also, we can easily see a series of contractions that result in a proof tree. Still, whichever component we choose to start with, out current algorithm cannot produce the term.

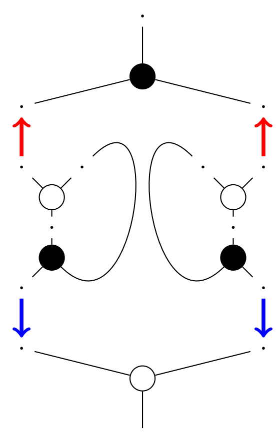


Figure 3.5: Parallellism: $(a/b) \otimes (c\backslash d) \Rightarrow (a/b) \otimes (c\backslash d)$

Unfortunately fixing the entire traversal algorithm is beyond the scope of this thesis. The encoding of the algorithm is therefore not complete. Specifically, it cannot handle parallellism in building the proof term. The version encoded is a modified version of the algorithm above.

1. Compute all components of *cg(P)*, consisting of a set of tensor links with a single main formula. Mark all these links as visited. Choose a component S to start from.
2. While *cg(P)* contains unvisited tensor links do the following:
   (a) Follow an unvisited command link c attached to S, forming a correct command subnet. This can only be done if the c's arrow is outgoing with respect to S. We enlarge S with the subnet attached to it via c.
   (b) For each cotensor that is doubly connected to S, including this cotensor in S. Repeat until no such cotensors can be found.
   (c) Follow a μ or $\tilde{\mu}$ [focus] link just as in step (a), forming a larger subnet S.

If any of these steps (that is, (a) and (c)) cannot be taken but unvisited links remain, the traversal was unsuccesful. If traversal is unsuccesful for all possible starting S, there is no proof term.

Using this updated algorithm, we can compute a proof term methodically for many (but not all) proof nets.

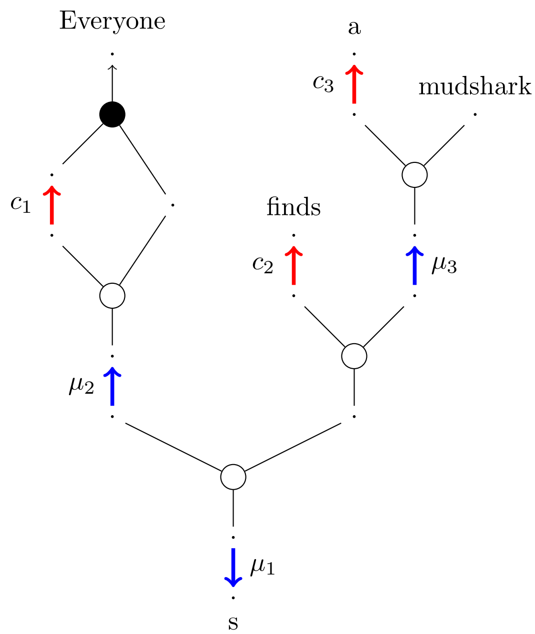


Figure 3.6: Composition graph for *Everyone finds a mudshark*

3.3 Example derivation

Let us see where the two proof terms of Figure 3.1 actually come from. The composition graph of the the proof net shown therein is given in Figure 3.5. We immediately see that there is only one component that is a feasible choice as a starting point: all other components have no outgoing focus links. Therefore we start in this case by choosing the (unique) component consisting of two tensors. Now we repeat part 2 of the algorithm until all links have been visited. We follow an outgoing command link, check for cotensors and follow an outgoing focus link.

At first we have no choice: the only outgoing command link is c_2. But then we have three focus links to choose from. Link μ_1 is not an option, since the resulting subnet has no outgoing commands links. We can choose either μ_2 or μ_3. In this composition graph, this choice determines the rest of the traversal. The endresult is two possible orders of traversal: $c_2 - \mu_3,\ c_3 - \mu_2,\ c_1 - \mu_1$ and $c_2 - \mu_2,\ c_1 - \mu_3,\ c_3 - \mu_1$.
We obtain a proof term by applying function application and abstraction in such an order. Each possible order of traversal therefore encodes a (distinct) proof term.

Chapter 4

Theorem Prover

4.1 Building the Theorem Prover

This chapter deals with the actual code of the theorem prover. It is written in Python 2.7 and can be found in full in the appendix. The code implements several algorithms, each of which is a part of proving a sequent in graphical calculus. Some parts we have adapted from [10], others are original work. Proving a sequent $A \Rightarrow B$ is done by calling `python LGprover.py "A=>B"`, with several possible extra options. These can be found using the `--help` command.

Before we describe the prover itself, a quick word on its performance. We have tested the prover only up to a certain extent. Automatic testing of tautologies $A \Rightarrow A$ with increasing complexity of A has so far returned only positive results. However, we note one major issue. When deriving a sequent $A_1, A_2 \dots A_n \vdash B_1, B_2 \dots B_m$, order is not preserved. Instead, each formula is unfolded and then the linking of input and output axioms between any and all modules is considered. This may lead to, for example, sentences like "subj tv det noun" to be derivable as "det noun tv subj", that is, we have a confusion between subject and object. The prover is insensitive to this distinction: once terms are unfolded all sentence structure is lost. Since both subject and object (determiner plus noun) are noun phrases, the prover cannot distinguish the two. This behaviour is not a bug: it is an actual feature of the prover implemented. Of course, for use with natural language, extra constraints should be considered. Note also that, since we have implemented a brute force approach, performance is definately not optimal.

4.2 Files and Classes

The most important files are LGprover.py, which is the main program, and classes_linear.py, containing all classes. Since we work with hypergraphs, implemented classes for graphs would need to be partially rewritten. Therefore we have built a simple class system from scratch.

We have ProofStructures which most importantly contain a list of Tensors. Proof nets are structures as well, since they are simply structures that can be rewritten to a tensor tree. The Tensor class is split up in OneHypothesis and TwoHypotheses classes. Furthermore we have the Vertex class for vertices and the Link class for links between vertices. Note that these Links are not

Tensors. The file table.py contains a small class which is used in combinatorics. The graph.py file is only used when the argument '`--term`' is used.

4.3 Unfolding

Unfolding is a recursive process. Each step is completely defined by the main connective of the given formula (if any). The code for lexical unfolding and indeed this whole project is an adaptation from code found in [3]. We unfold a single vertex at a time, since the vertex is labeled with a formula. The formula defines the (co)tensor as per Figure 2.3, which is first further unfolded, and then joined to the first vertex in question.

4.4 Pruning

For each possible way of linking the atoms of our modules, we need to consider whether the resulting structure is a proof net. This brute force approach leads to quite the computational overhead. If we can disqualify some possible linkings beforehand, we prune our search space. We prune only very simple configurations which are not derivable. If the number of input and output atoms do not match, we can forget a derivation. Linking a tensor to itself is also not a very good idea. The more possible linkings can be pruned, the less work we need to do afterwards. There are many more pruning checks that can be done.

4.5 Combinatorics

Pruning can still leave a number of possible atom linkings to be tried. We must try to rewrite each of these proof structures to a proof net. Each correct proof net we find is a derivation of a different sequent. In chapter 5.1 we explain why we sometimes have more than one proof net.

4.5.1 Shallow/Deep copy

To show that a proof structure is a proof net we simply take the structure and continuously apply (generalized) contraction. If we cannot apply a rule anymore we are done, and check whether we have a proof net. We have already modified our set of modules (the result of unfolding our sequent) by atom linking to form the structure. Our rewriting will modify it even more. If we want to consider the next possible linking (because the previous has been proved to be (in)correct), we need the set of modules to start from. The simple solution would be to take a copy of the set of modules before considering a possible binding and work on this copy. This raises the following problem.

If we take a shallow copy, the problem is not solved. The result of contraction is a linking between the surrounding parts. These parts are the vertices in our structure which are not copied individually. Our modules will have remembered the previous method, which is unacceptable. On the other hand, making a deep copy of our modules requires a lot of work. In fact, it might be easier to just unfold our set of formulas again. This is exactly what we do for each new possible atom linking that we consider.

4.5.2 Repeated generalised contraction

Contraction is a method of ProofStructures. For each cotensor in the net, we try to contract. If this is possible for a single cotensor, we rewrite the net as per contraction, close the loop, and call the method again. We stop calling the method once none of the cotensors can contract. This way, either all cotensors have contracted, or a cotensor remains that cannot be contracted. The existence of this last cotensor would prove we do not have a proof net. Actually checking whether contraction is possible is a simple case of pattern matching of contraction configurations and parts of the structure.

4.6 Proof net

If none of the rewriting rules can be applied, we check for any remaining cotensors, cycles or unconnected parts. Remaining cotensors are easily detected; connectedness and acyclicity are determined by a traversal of the structure.

4.7 Proof term

We need more information in our nets to do term traversal. Instead of translating the nets (a costly procedure), we stick all extra information onto our classes when creating a structure. This means all proof structures come with a composition graph, even when `--term` is not called. This causes only limited computational overhead.

4.7.1 Traversing the Composition Graph

In order to traverse the graph, we create an abstract representation of it in terms of a simpler graph. Nodes correspond with components and edges with links. Actual traversal order is only calculated on these graphs. Once we have determined the possible orders of traversal, we switch back to the composition graph to calculate the proof terms, since they hold the actual information needed (such as types and variable assignment to formulas).

Chapter 5

Conclusion

5.1 Further research

As described in the previous chapter, the prover does not preserve order of formulas. The prover needs to be adjusted for order to be taken into account. In creating the prover we have not encountered a satisfying method of doing so. Adjusting the prover thusly would make for an interesting extension. There are a lot of pruning strategies that can drastically improve the performance of the prover. Only very simple pruning strategies have been implemented.

The algorithms the theorem prover relies upon are in some cases in need of further specification. These algorithms, especially that for term traversal, have so far been incompletely described. Their full description is an ideal subject for further research.

5.2 Conclusion

The theorem prover created for this thesis is given in Appendices A through G. It is based on solid work on proof nets, most of which is implemented. Its correctness is directly derivable from the correctness of this work. In implementing, some of the underlying theory was found to be not concrete enough. Where possible we have worked around such problems, but some features (such as term traversal) are incomplete due to the lack of theory to draw from. Whether this has hampered the prover, we cannot say for sure. So far testing gives positive results, but we can only accept the prover as correct when proven that its algorithms correspond precisely to the theory.

By building a theorem prover for Lambek-Grishin calculus based on graphical calculus, we have shown that an implementation is indeed possible. More importantly, we hope that it will be used in further research on graphical LG and its characteristics.

Bibliography

[1] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 1956.

[2] Noam Chomsky. Formal properties of grammars. *Handbook of Mathematical Psychology*, 2:323-418, 1963.

[3] Sjoerd Dost. Formalizing the Graphical Notation of Proof Structures for Lambek-Grishin Calculus. *For the course "Logical Methods in Natural Language Processing" by Michael Moortgat. Utrecht University*, 2012.

[4] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1-102, 1987.

[5] Aravind Joshi. Tree Adjoining Grammars: How much context-sensitivity is necessary for characterizing structural descriptions. *Natural Language Processing - Theoretical, Computational and Psychological Perspective*, 1985.

[6] Laura Kallmeyer. *Parsing Beyond Context-Free Grammars*. Springer-Verlag, Berlin Heidelberg, 2010.

[7] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65: 154-170, 1958.

[8] Matthijs Melissen. The generative capacity of the Lambek-Grishin calculus: A new lower bound. *Proceedings 14th Conference on Formal Grammar*, 5591:118-132., 2010.

[9] Michael Moortgat. Symmetric categorial grammar. *Journal of Philosophical Logic*, 38(6):681-710, 2009.

[10] Michael Moortgat and Richard Moot. Proof nets and the categorial flow of information. 2012.

[11] Richard Moot. *http://www.labri.fr/perso/moot/grail.html*. 2002.

[12] Richard Moot. Proof nets for display logic. *Technical report, CNRS and INRIA Futurs*, 2007.

[13] Richard Moot and Christian Retoré. *The Logic of Categorial Grammars*. Springer-Verlag, Berlin Heidelberg, 2012.

[14] Mati Pentus. Lambek grammars are context-free. *Logic in Computer Science*, 1993.

[15] Stuart M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333-343, 1985.

Appendix A

LGprover

```
#!/usr/bin/env python

# LIRa refers to:
# http://www.phil.uu.nl/~moortgat/lmnlp/2012/Docs/contributionLIRA.pdf
# Proofs nets and the categorial flow of information
# Michael Moortgat and Richard Moot

# Algorithm:
# 1) Unfolding
# 2) Pruning
# 3) Combinatorics
# 4) Soundness
# 5) Proof Term

from helper_functions import *
import classes_linear as classes
import argparser
from table import Table
import graph as g
import term

import os, sys
import platform
import itertools


# By default the formula appears in hypothesis position.
def unfold_formula(formula, raw, hypothesis):
    vertex = classes.Vertex(formula, hypothesis)
    structure = classes.ProofStructure(formula, vertex)
    vertex.is_value = True
    vertex.term = term.Atomic_Term(raw)
    if simple_formula(formula):
        structure.add_atom(vertex,hypothesis)
    else:
        vertex.unfold(formula, hypothesis, structure)     # Recursively unfold

    to_remove = []
    for l in structure.links:
        if l.contract():
            to_remove.append(l)
    for l in to_remove:
        structure.links.remove(l)

    # Toggle whole formula
    p = argparser.Parser()
    args = p.get_arguments()
    if args.main:
        vertex.main ='|texttt{{{0}}}'.format(args.main)
```

```
    return structure


def unfold_all(sequentlist, raw):
    classes.vertices = {}
    classes.removed = 0
    classes.next_alpha = 0
    hypotheses = [unfold_formula(x, y, True) for (x,y) in zip(sequentlist[0], raw[0])]
    conclusions = [unfold_formula(x, y, False) for (x,y) in zip(sequentlist[1], raw[1])]
    modules = hypotheses + conclusions
    return modules


def create_composition_graph(sequent, raw, possible_binding):
    # Unfolding (again)
    modules = unfold_all(sequent, raw)

    components = []
    for m in modules:
        components.extend(m.get_components())

    components = [x for x in components if not x == []]

    # Creating the composition graph
    composition_graph = modules[0]
    for m in modules[1:]:
        composition_graph.join(m)
    for b in possible_binding:
        link = classes.Link(b[1],b[0])
        if not link.contract():
            composition_graph.add_link(link)

    command = [l for l in composition_graph.links if l.is_command()]
    mu_comu = [l for l in composition_graph.links if not l.is_command()]

    return composition_graph, components, command, mu_comu


def main():

    p = argparser.Parser()
    args = p.get_arguments()
    if len(args.sequent) != 1:
        p.print_help()
        sys.exit()
    raw_sequent = args.sequent[0]

    lexicon = []
    if args.lexicon:
        lexicon, classes.polarity = build_lexicon(args.lexicon)

    # Parsing the sequent
    raw_sequent = [map(lambda x : x.strip(), y) for y in
                [z.split(",") for z in raw_sequent.split("=>")]]

    if len(raw_sequent) != 2:
        syntax_error()

    sequent = raw_sequent
    if lexicon:
        sequent = [map(lambda y : lookup(y, lexicon), x) for x in raw_sequent]

    # 1) Unfolding
    # Links added as either command or mu/comu

    modules = unfold_all(sequent, raw_sequent)

    # 2) Pruning
    # Checks: atom bijection

    atom_hypotheses = []
    atom_conclusions = []
    for m in modules:
        atom_hypotheses += m.hypotheses
```

```
        atom_conclusions += m.conclusions

    # Van Benthem count / Count Invariance
    if sorted([h.main for h in atom_hypotheses]) != sorted([c.main for c in
        atom_conclusions]):
        no_solutions()

    # Chart of possible atom unification

    chart = {}
    for h in atom_hypotheses:
        if h.main not in chart:
            chart[h.main] = Table(h)
        else:
            chart[h.main].add_hypothesis(h)
    for c in atom_conclusions:
        chart[c.main].add_conclusion(c)

    for t in chart.values():
        t.create_table()

    # Checks: (simple) acyclicity
        t.prune_acyclicity()

    # TODO: Checks: (simple) connectedness
        #t.prune_connectedness()

    # Checks: Co-tensor will never contract
        t.prune_cotensor()

    # TODO: Checks: focusing, mu / comu

    # 3) Combinatorics
    # Creating all possible derivation trees
    for t in chart.values():
        t.combine()

    tables = [t.atom_bindings for t in chart.values()]
    possible_bindings = []
    table_product = list(itertools.product(*tables))
    for product in table_product:
        binding = []
        for b in product:
            binding += b
        possible_bindings += [binding]

    # For each possible binding, create a proof net
    # Shallow / Deep copy problem: unfold every time
    # This is cost-intensive but the easiest way (?)
    # This requires bindings to refer to indices
    # instead of Vertex objects (these are destroyed each unfolding)

    no_solution = True
    # Erase file
    if args.tex:
        f = open('formula.tex', 'w')
        f.close()

    for i in range(0,len(possible_bindings)):
        # Copy problem
        if i > 0:
            modules = unfold_all(sequent, raw_sequent)

        proof_net = modules[0]
        for m in modules[1:]:
            proof_net.join(m)
        for b in possible_bindings[i]:
            link = classes.Link(b[1],b[0])
            if not link.contract():
                proof_net.add_link(link)

        # Checks: (mu / comu) -- command bijection
        if not proof_net.bijection():
            continue
```

```python

        # 4) Soundness
        # Collapse all links, not needed anymore

        for l in proof_net.links:
            l.collapse_link()
        proof_net.links = []

        # Try to contract
        proof_net.contract()

        # If there are cotensors left, this is not a solution
        if [x for x in proof_net.tensors if x.is_cotensor()]:
            print "not a solution"
            continue

        # Check: Connectedness of the whole structure
        # Traversal, checking total connectedness and acyclicity
        # NOTE: Can only be checked on contracted net

        if proof_net.tensors:
            if not proof_net.connected_acyclic():
                continue

        # Print to TeX
        if args.tex:
            proof_net.toTeX(no_solution)

        no_solution = False

        # 5) Proof term
        # TODO: Compostion Graph Traversal
        # NOTE: Can only be done on non-contracted net

        if args.term:

            composition_graph, components, command, mu_comu = create_composition_graph(
                sequent, raw_sequent, possible_bindings[i])

            # Step 1: create matchings
            # TODO: Working assumptions (see graph.py)

            # Create traversal graph
            cotensors = [x for x in composition_graph.tensors if x.is_cotensor()]
            graph = g.Graph(components, cotensors, mu_comu, command)

            # Step 2: Calculate term in order of matching
            matching = graph.match()

            # Step 3: Write to TeX
            graph.to_TeX(matching, composition_graph)

        # For debugging
        # proof_net.print_debug()

    if args.tex and not no_solution:
        # End of document
        f = open('formula.tex', 'a')
        f.write('\end{document}')
        f.close()
        os.system('pdflatex formula.tex')
        if platform.system() == 'Windows':
            os.system('start formula.pdf')
        elif platform.system() == 'Linux':
            os.system('pdfopen --file formula.pdf')
        # Mac OS X ?

    if no_solution:
        no_solutions()

if __name__ == '__main__':
    main()
```

Code/LGprover.py

Appendix B

Classes

```
from helper_functions import *
import argparser
import sys
import pyparsing
import term

drawn = []
texlist = []
vertices = {}
removed = 0
polarity = {}


class ProofStructure(object):

    def __init__(self, formula, vertex):
        self.formula = formula
        self.main = vertex
        self.tensors = []
        self.links = []
        self.order = [0]
        self.hypotheses = []
        self.conclusions = []

    def print_debug(self):
        print ""
        print [x.alpha for x in self.tensors]
        print self.order
        print [(x.top.alpha,x.bottom.alpha) for x in self.links]

    def add_tensor(self, tensor):
        self.tensors.append(tensor)
        tensor.index = len(self.tensors) - 1
        tensor.alpha = len(self.tensors)

    def add_link(self, link):
        self.links.append(link)

    def add_atom(self, atom, hypo):
        if hypo:
            self.conclusions.append(atom)
        else:
            self.hypotheses.append(atom)

    def bijection(self):
        count = 0
        for link in self.links:
            if link.is_command():
                count += 1
```

```
            else:
                count -= 1
        return count == 0

    def join(self, module):
        # Temporary fix on order for printing
        if module.tensors:
            higher_order = [x + len(self.order) for x in module.order]
            for t in module.tensors:
                t.alpha += len(self.order)
            self.order += higher_order
        self.tensors += module.tensors
        self.links += module.links

        del module

    def contract(self):
        contracted = False

        for t in self.tensors:
            if t.is_cotensor():

                (complement, c_main, t_top, s) = t.contractions(self)

                if complement is not None:

                    # Simple contraction, L* and R(*)
                    link = None
                    if not s:
                        if t_top:
                            link = Link(t.arrow, c_main.alpha)
                        else:
                            link = Link(c_main.alpha, t.arrow)

                    # Generalized contraction, R/, R\, L(/) and L(\)
                    else:
                        link2 = None
                        if t_top:
                            link = Link(t.arrow, t.bottom.alpha)
                            link2 = Link(complement.top.alpha, c_main.alpha)
                        else:
                            link = Link(t.top.alpha, t.arrow)
                            link2 = Link(c_main.alpha, complement.bottom.alpha)
                        link2.collapse_link()

                    link.collapse_link()

                    # Removing the tensor
                    a = complement.alpha
                    self.tensors.remove(complement)
                    del complement
                    if a in self.order:
                        self.order.remove(a)
                        for i in range(len(self.order)):
                            if self.order[i] > a:
                                self.order[i] = self.order[i] - 1

                    # Removing the cotensor
                    a = t.alpha
                    self.tensors.remove(t)
                    del t
                    if a in self.order:
                        self.order.remove(a)
                        for i in range(len(self.order)):
                            if self.order[i] > a:
                                self.order[i] = self.order[i] - 1

                    contracted = True
                    break

        if contracted:
            self.contract()

    def connected_acyclic(self):
```

```
        list = []
        for t in self.tensors:
            list.append(t)
        checklist = [(list[0], None)]
        connected_and_acyclic = True

        while checklist:
            (tensor, previous) = checklist[0]
            checklist.pop(0)
            n = tensor.neighbors()

            if previous is not None:
                test = len(n)
                n = [x for x in n if x is not previous]
                if test != (len(n) + 1):
                    # Cycle found
                    connected_and_acyclic = False
                    break
            if tensor not in list:
                # Cycle found
                connected_and_acyclic = False
                break
            list.remove(tensor)
            for t in n:
                checklist.append((t, tensor))

        if list:
            # Disconnected part remains
            connected_and_acyclic = False
        return connected_and_acyclic

    # Determining the components (maximal subgraphs)
    def get_components(self):

        tens = [[x] for x in self.tensors if not x.is_cotensor()]

        if len(tens) < 2:
            return tens

        trial = True
        while trial:
            trial = False
            x = tens[0][0]
            for y in tens[1:]:
                if shortest_path(self,x,y[0]) is not None:
                    tens[0].append(y[0])
                    tens.remove(y)
                    trial = True
            if len(tens) < 2:
                return tens

        return tens

    def toTeX(self, first):
        global texlist, drawn
        drawn = []
        texlist = []

        # Write to formula.tex
        # Header
        f = open('formula.tex', 'a')

        rotate = ""

        if not first:
            f.write("\n")
        else:
            f.write('\documentclass[tikz]{standalone}\n\n')
            f.write('\usepackage{tikz-qtree}\n')
            f.write('\usepackage{stmaryrd}\n')
            f.write('\usepackage{scalefnt}\n')
            f.write('\usepackage{amssymb}\n\n')
            f.write('\\begin{document}\n\n')
```

```
            f.write('\\tikzstyle{mybox} = [draw=red, fill=blue!20, very thick,rectangle, rounded corners, inner sep=10pt, inner ysep=20pt]\n\n')

            # Toggle rotation
            p = argparser.Parser()
            args = p.get_arguments()
            if args.rotate:
                rotate = "rotate=270,"

        # Tikzpicture

        f.write('\\begin{tikzpicture}[')
        f.write(rotate)
        f.write('scale=.8,')
        f.write('cotensor/.style={minimum size=2pt,fill,draw,circle},\n')
        f.write('tensor/.style={minimum size=2pt,fill=none,draw,circle},')
        f.write('sibling distance=1.5cm,level distance=1cm,auto]\n\n')

        x = 0
        y = 0

        if not self.tensors:
            #f.write(self.main.toTeX(x, y, self.main.main, self))
            f.write("\\node at (0,0) [")
            if self.main.hypothesis is not None:
                f.write("label=above:${0}$".format(operators_to_TeX(self.main.hypothesis)))
            if self.main.hypothesis is not None and self.main.conclusion is not None:
                f.write(", ")
            if self.main.conclusion is not None:
                f.write("label=below:${0}$".format(operators_to_TeX(self.main.conclusion)))
            f.write("] {.};\n")

        else:
            # Shuffle self.tensors according to order
            # Trimming order to size instead of
            # losing myself in LaTeX-printing details
            self.order = [x for x in self.order if x < len(self.tensors)]
            self.tensors = map(lambda x: self.tensors[x],self.order)
            previous_tensor = None

        for tensor in self.tensors:

            if previous_tensor is not None:
                (x_adj,y_adj) = adjust_xy(previous_tensor, tensor)
                x += x_adj
                y += y_adj

            f.write('{0} at ({1},{2}) {{}};\n'.format(tensor.toTeX(),x,y))
            f.write(tensor.hypotheses_to_TeX(x, y))
            f.write(tensor.conclusions_to_TeX(x, y))
            y -= 3
            previous_tensor = tensor

        for line in texlist:
            f.write(line)

        for l in self.links:
            f.write(l.draw_line())

        f.write('\n\end{tikzpicture}\n\n')
        f.close()


def adjust_xy(previous, current):
    if isinstance(previous, OneHypothesis):
        if previous.bottomLeft.conclusion is current:
            if isinstance(current, OneHypothesis):
                if current.top.hypothesis is previous:
                    return (-1,1)
            else:
                if current.topRight.hypothesis is previous:
                    return (-2,1)
```

```
        elif previous.bottomRight.conclusion is current:
            if isinstance(current, OneHypothesis):
                if current.top.hypothesis is previous:
                    return (1,1)
            else:
                if current.topLeft.hypothesis is previous:
                    return (2,1)
    else:
        if previous.bottom.conclusion is current:
            if isinstance(current, TwoHypotheses):
                if current.topLeft.hypothesis is previous:
                    return (1,1)
                elif current.topRight.hypothesis is previous:
                    return (-1,1)
    return (0,0)


class Vertex(object):

    def __init__(self, formula=None, hypo=None):
        global vertices, removed
        self.term = None
        self.set_hypothesis(None)
        self.set_conclusion(None)
        self.alpha = len(vertices) + removed
        self.is_value = True      # if False then is_context
        vertices[self.alpha] = self
        if formula is not None:
            self.main = formula
            self.attach(formula, hypo)

    def set_hypothesis(self, hypo):
        self.hypothesis = hypo

    def set_conclusion(self, con):
        self.conclusion = con

    def toTeX(self, x, y, tensor, struc):
        global texlist, drawn
        co = ""
        if tensor is not self.main:
            if tensor.is_cotensor() and tensor.arrow is self.alpha:
                co = "[->]"
            line = "\draw{0} ({1}) -- ({2});\n".format(co,"t"+
                    str(tensor.alpha),"v"+str(self.alpha))
            # TODO: curved links are broken, self.hypo can be a Link
            #if self.internal() and self.conclusion is tensor:
            #    if struc.order.index(tensor.index) != struc.order.index(self.hypothesis.
                index) + 1:
            #        line = self.curved_tentacle(tensor, self.hypothesis)
            texlist.append(line)
            if self.alpha in drawn:
                return ""
            drawn.append(self.alpha)
        label = operators_to_TeX(self.main)
        tex = "\\node ({0}) at ({1},{2}) {{${3}$}};\n".format("v"+str(self.alpha),
                x, y, label)
        return tex

    def curved_tentacle(self, tensor, prev_tensor):
        co = ""
        if tensor.is_cotensor() and tensor.arrow is self.alpha:
            co = "[->]"
        start = "\draw{0} ({1}) ..controls ".format(co, "t"+str(tensor.alpha))
        direction = "west"
        if isinstance(tensor, TwoHypotheses):
            if tensor.topRight is self:
                direction = "east"
        elif isinstance(prev_tensor, OneHypothesis):
            if prev_tensor.bottomRight is self:
                direction = "east"
        controls = "+(north {0}:4) and +(south {0}:4.0)".format(direction)
        end = ".. ({0});\n".format("v"+str(self.alpha))
        line = start + controls + end
```

```
        return line

    def internal(self):
        return ((isinstance(self.hypothesis, Tensor) or
                    isinstance(self.hypothesis, Link)) and
                    (isinstance(self.conclusion, Tensor) or
                    isinstance(self.conclusion, Link)))

    def is_hypothesis(self):
        return (isinstance(self.hypothesis, str) or (self.hypothesis is None))

    def is_conclusion(self):
        return (isinstance(self.conclusion, str) or (self.conclusion is None))

    def is_lexical_item(self):
        return (self.is_hypothesis() and self.is_conclusion())

    def attach(self, label, hypo):
        if hypo:
            self.set_hypothesis(label)
        else:
            self.set_conclusion(label)

    # Important: use of hypo
    # l.top.get_term(False)
    # l.bottom.get_term(True)
    def get_term(self, hypo):
        if isinstance(self.term, term.Connective_Term):
            tensor = None
            if hypo:
                tensor = self.conclusion
            else:
                tensor = self.hypothesis

            if isinstance(tensor, Link) or isinstance(tensor, str) or tensor.is_cotensor
                ():
                self.term = term.Atomic_Term()
                return self.term

            # Now we can assume self.term is a Term object
            left = tensor.left.get_term(tensor.left.hypothesis is tensor)
            right = tensor.right.get_term(tensor.right.hypothesis is tensor)

            self.term = term.Complex_Term(left, self.term, right)

        return self.term

    # This is the source of the recursion
    def unfold(self, formula, hypo, structure, i=None):
        try:
            [left, connective, right] = parse(formula)
        except pyparsing.ParseException:
            syntax_error()
        vertex = Vertex(formula)
        if i is not None:
            self.term = term.Connective_Term(connective)
        vertex.term = term.Connective_Term(connective)
        self.polarity = con_pol(connective)
        vertex.polarity = self.polarity
        if hypo:
            link = Link(self.alpha,vertex.alpha)
        else:
            link = Link(vertex.alpha,self.alpha)
        (premises, geometry, term_geo) = tensor_table[(connective,hypo)]
        if premises == 1:
            t = (OneHypothesis(left, right, geometry, vertex, structure, hypo, i))
        else:
            t = (TwoHypotheses(left, right, geometry, vertex, structure, hypo, i))
        t.term = term_geo
        t.set_left_and_right()
        structure.add_link(link)


class Tensor(object):
```

```
    def __init__(self):
        print "error"

    def toTeX(self):
        co = ''
        if self.is_cotensor():
            co = 'co'
        return '\\node [{0}tensor] ({1})'.format(co,"t"+str(self.alpha))

    def parse_geometry(self, geometry, vertex):
        index = geometry.find("<")
        if index > -1:
            self.arrow = vertex.alpha
        geometry = geometry.replace("<", "")
        return geometry

    def get_lookup(self, left, right, vertex):
        lookup = {
            'f':(Tensor.attach,vertex),
            'l':(Tensor.eval_formula,left),
            'r':(Tensor.eval_formula,right),
            'v':True,
            'e':False
        }
        return lookup

    def set_structure(self, struc, hypo, origin_index):
        if origin_index is not None:
            new = len(struc.order)
            origin_index = struc.order.index(origin_index)
            if hypo:
                struc.order.insert(origin_index + 1,new)
            else:
                struc.order.insert(origin_index,new)
        struc.add_tensor(self)
        self.structure = struc

    def is_cotensor(self):
        return hasattr(self, 'arrow')

    def attach(self, vertex, hypo, is_value, main=True):
        vertex.attach(self, not hypo)
        vertex.is_value = is_value
        if main:
            self.main = vertex
        return vertex

    def eval_formula(self, part, hypo, is_value):
        global polarity
        if simple_formula(part):
            atom = Vertex(part, hypo)
            self.structure.add_atom(atom, not hypo)
            atom.term = term.Atomic_Term()
            if part in polarity:
                atom.polarity = polarity[part]
            else:
                atom.polarity = '-'
            return self.attach(atom, hypo, is_value, False)
        else:
            vertex = Vertex()
            self.attach(vertex, hypo, is_value, False)
            part = part[1:-1]
            vertex.unfold(part, not hypo, self.structure, self.index)
            # Toggle abstract
            p = argparser.Parser()
            args = p.get_arguments()
            if args.abstract:
                vertex.main = "."
            else:
                vertex.main = part
            return vertex

    def get_term(self):
```

```python
        # term has never been evaluated before
        if isinstance(self.term, str):
            t1 = self.left.get_term(self.left.hypothesis is self)
            t2 = self.right.get_term(self.right.hypothesis is self)
            self.term = term.Cotensor_Term(t1, t2, self.main.get_term(self.main.hypothesis is self))
        return self.term

    def neighbors(self):
        n = []
        for h in self.get_hypotheses():
            if isinstance(h.hypothesis, Tensor):
                n.append(h.hypothesis)
        for c in self.get_conclusions():
            if isinstance(c.conclusion, Tensor):
                n.append(c.conclusion)
        return n

    def non_main_connections(self):
        n = []
        for h in self.get_hypotheses():
            if not h is self.main:
                if isinstance(h.hypothesis, Tensor) or isinstance(h.hypothesis, Link):
                    n.append(h.hypothesis)
        for c in self.get_conclusions():
            if not c is self.main:
                if isinstance(c.conclusion, Tensor) or isinstance(c.conclusion, Link):
                    n.append(c.conclusion)
        return n


class OneHypothesis(Tensor):

    def __init__(self, left, right, geometry, vertex, struc, hypo, i):
        Tensor.set_structure(self, struc, hypo, i)
        geometry = Tensor.parse_geometry(self, geometry, vertex)
        lookup = Tensor.get_lookup(self, left, right, vertex)
        (function,arg) = lookup[geometry[0]]
        self.top = function(self, arg, 1, lookup[geometry[3]])
        (function,arg) = lookup[geometry[1]]
        self.bottomLeft = function(self, arg, 0, lookup[geometry[4]])
        (function,arg) = lookup[geometry[2]]
        self.bottomRight = function(self, arg, 0, lookup[geometry[5]])

    def get_hypotheses(self):
        return [self.top]

    def get_conclusions(self):
        return [self.bottomLeft, self.bottomRight]

    def num_hyp(self):
        return 1

    def num_con(self):
        return 2

    def hypotheses_to_TeX(self, x, y):
        return self.top.toTeX(x, y + 1, self, self.structure)

    def conclusions_to_TeX(self, x, y):
        s1 = self.bottomLeft.toTeX(x - 1, y - 1, self, self.structure)
        s2 = self.bottomRight.toTeX(x + 1, y - 1, self, self.structure)
        return s1 + s2

    def replace(self, replace, vertex):
        global vertices, removed
        if self.left is replace:
            self.left = vertex
        if self.right is replace:
            self.tight = vertex
        if self.is_cotensor() and self.arrow == replace.alpha:
            self.arrow = vertex.alpha
        if self.top is replace:
```

```
            self.top = vertex
        elif self.bottomLeft is replace:
            self.bottomLeft = vertex
        elif self.bottomRight is replace:
            self.bottomRight = vertex
        del vertices[replace.alpha]
        removed += 1

    # Can this cotensor contract?
    # If so, return the tensor it contracts with
    def contractions(self, net):
        if isinstance(self.bottomLeft.conclusion, TwoHypotheses):
            t = self.bottomLeft.conclusion
            if not t.is_cotensor():
                if self.bottomLeft is t.topLeft:
                    if self.bottomRight.conclusion is t:
                            # L*
                            return (t, t.bottom, True, [])

                    s = shortest_path(net, self, t)
                    if only_grishin_tensors(s):
                        #R\
                        return (t, t.topRight, False, s)

        if isinstance(self.bottomRight.conclusion, TwoHypotheses):
            t = self.bottomRight.conclusion
            if not t.is_cotensor():
                if self.bottomRight is t.topRight:
                    s = shortest_path(net, self, t)
                    if only_grishin_tensors(s):
                        #R/
                        return (t, t.topLeft, False, s)

        return (None, None, None, None)

    def set_left_and_right(self):
        if self.term[0] is 'l':
            self.left = self.bottomLeft
        if self.term[0] is 'r':
            self.left = self.bottomRight
        if self.term[0] is 't':
            self.left = self.top
        if self.term[1] is 'l':
            self.right = self.bottomLeft
        if self.term[1] is 'r':
            self.right = self.bottomRight
        if self.term[1] is 't':
            self.right = self.top


class TwoHypotheses(Tensor):

    def __init__(self, left, right, geometry, vertex, struc, hypo, i):
        Tensor.set_structure(self, struc, hypo, i)
        geometry = Tensor.parse_geometry(self, geometry, vertex)
        lookup = Tensor.get_lookup(self, left, right, vertex)
        (function,arg) = lookup[geometry[0]]
        self.topLeft = function(self, arg, 1, lookup[geometry[3]])
        (function,arg) = lookup[geometry[1]]
        self.topRight = function(self, arg, 1, lookup[geometry[4]])
        (function,arg) = lookup[geometry[2]]
        self.bottom = function(self, arg, 0, lookup[geometry[5]])

    def get_hypotheses(self):
        return [self.topLeft, self.topRight]

    def get_conclusions(self):
        return [self.bottom]

    def num_hyp(self):
        return 2

    def num_con(self):
        return 1
```

```python

    def hypotheses_to_TeX(self, x, y):
        s1 = self.topLeft.toTeX(x - 1, y + 1, self, self.structure)
        s2 = self.topRight.toTeX(x + 1, y + 1, self, self.structure)
        return s1 + s2

    def conclusions_to_TeX(self, x, y):
        return self.bottom.toTeX(x, y - 1, self, self.structure)

    def replace(self, replace, vertex):
        global vertices, removed
        if self.left is replace:
            self.left = vertex
        if self.right is replace:
            self.tight = vertex
        if self.is_cotensor() and self.arrow == replace.alpha:
            self.arrow = vertex.alpha
        if self.topLeft is replace:
            self.topLeft = vertex
        elif self.topRight is replace:
            self.topRight = vertex
        elif self.bottom is replace:
            self.bottom = vertex
        del vertices[replace.alpha]
        removed += 1

    # Can this cotensor contract?
    # If so, return the tensor it contracts with
    def contractions(self, net):
        if isinstance(self.topLeft.hypothesis, OneHypothesis):
            t = self.topLeft.hypothesis
            if not t.is_cotensor():
                if self.topLeft is t.bottomLeft:
                    if self.topRight.hypothesis is t:
                        # R(*)
                        return (t, t.top, False, [])

                    s = shortest_path(net, self, t)
                    if only_lambek_tensors(s):
                        # L(\)
                        return (t, t.bottomRight, True, s)

        if isinstance(self.topRight.hypothesis, OneHypothesis):
            t = self.topRight.hypothesis
            if not t.is_cotensor():
                if self.topRight is t.bottomRight:
                    s = shortest_path(net, self, t)
                    if only_lambek_tensors(s):
                        # L(/)
                        return (t, t.bottomLeft, True, s)

        return (None, None, None, None)

    def set_left_and_right(self):
        if self.term[0] is 'l':
            self.left = self.topLeft
        if self.term[0] is 'r':
            self.left = self.topRight
        if self.term[0] is 'b':
            self.left = self.bottom
        if self.term[1] is 'l':
            self.right = self.topLeft
        if self.term[1] is 'r':
            self.right = self.topRight
        if self.term[1] is 'b':
            self.right = self.bottom


class Link(object):

    def __init__(self, top, bottom):
        global vertices
        self.top = vertices[top]
        self.bottom = vertices[bottom]
```

```
        self.top.set_conclusion(self)
        self.bottom.set_hypothesis(self)

    def contract(self):
        if self.top.is_value == self.bottom.is_value:
            self.collapse_link()
            return True
        return False

    def collapse_link(self):
        global vertices, removed
        self.top.set_conclusion(self.bottom.conclusion)
        if not isinstance(self.bottom.conclusion, Tensor):
            self.top.term = self.bottom.term
            del vertices[self.bottom.alpha]
            removed += 1
        else:
            self.bottom.term = self.top.term
            self.bottom.conclusion.replace(self.bottom, self.top)

    def is_command(self):
        if self.top.is_value:
            return True
        return False

    # Meaning whether the atomic formula is
    # positive (True) or negative (False)
    def positive(self):
        if self.top.polarity is '+':
            return True
        return False

    def draw_line(self):
        if (self.top.alpha in drawn) and (self.bottom.alpha in drawn):
            top = "v" + str(self.top.alpha)
            bottom = "v" + str(self.bottom.alpha)
            line = "\draw[dotted] ({0}) -- ({1});\n".format(top, bottom)
            return line
        else:
            return ""


# Dijkstra's algorithm
def shortest_path(proofnet, source, target):
    dist = {}
    previous = {}
    q = []

    for t in proofnet.tensors:
        # set distance to functional infinity
        dist[t] = len(proofnet.tensors)
        previous[t] = None
        q.append(t)

    dist[source] = 0

    while q:
        u = q[0]
        for t in q[1:]:
            if dist[t] < dist[u]:
                u = t
        q.remove(u)
        if u is target:
            break

        # This means there are tensors left
        # that are unreachable from source
        if dist[u] == len(proofnet.tensors):
            return None
            break

        n = u.neighbors()
        if u is source and target in n:
            n.remove(target)
```

```

        for v in n:
            if not v in q:
                continue
            alt = dist[u] + 1
            if alt < dist[v]:
                dist[v] = alt
                previous[v] = u

    s = []
    u = previous[target]

    while u in previous:
        if u is not source:
            s.insert(0,u)
        u = previous[u]

    return s


def only_grishin_tensors(path):
    only_grishin = True
    for t in path:
        if t.is_cotensor() or isinstance(t, TwoHypotheses):
            only_grishin = False
            break
    return only_grishin

def only_lambek_tensors(path):
    only_lambek = True
    for t in path:
        if t.is_cotensor() or isinstance(t, OneHypothesis):
            only_lambek = False
            break
    return only_lambek
```

Code/classes_linear.py

Appendix **C**

Helper functions

```
import re
import sys
import pyparsing as p


lexicon = {}


def parse(formula):
    atom = p.Word(p.alphas + "'|{}$")
    operator = p.oneOf("\\ / * (\\) (/) (*)")
    bracket = p.oneOf("( )")
    f = p.OneOrMore(atom | operator | bracket)
    formula = f.parseString(formula)
    check = [len(formula) for i in range(0, len(formula))]
    operators = ["\\", "/", "*", "(\\)", "(/)", "(*)"]
    symmetry = 0
    for i,c in enumerate(formula):
        if c is "(":
            symmetry += 1
        elif c is ")":
            symmetry -= 1
        if symmetry < 0:
            syntax_error()
        if c in operators:
            check[i] = symmetry
    main = check.index(min(check))
    return ["".join(formula[:main]),formula[main],"".join(formula[main+1:])]

# This returns True if the formula contains no connectives.
def simple_formula(formula):
    connectives = re.compile(r"(\*|\\|/|\(\*\)|\(/\)|\(\\\))")
    search = connectives.search(formula)
    return search is None


def operators_to_TeX(string):
    string = string.replace("\\", "\\backslash ")
    string = string.replace("(*)", "\oplus ")
    string = string.replace("*", "\otimes ")
    string = string.replace("(/)", "\oslash ")
    string = string.replace("(\\backslash )", "\obslash ")
    string = string.replace("|", "\\")
    return string


def no_solutions():
    print "\nThere are no solutions"
    sys.exit(1)
```

```python

def syntax_error():
    print "\nSyntax error in formula"
    sys.exit(1)


def lookup(label, lexicon):
    if label in lexicon:
        # Returns first value found for label in lexicon
        # Multiple entries are not supported
        return lexicon[label][0]
    else:
        return label


def build_lexicon(pathfile):
    lex = {}
    pol = {}
    f = open(pathfile)
    for line in f:
        if line[0] != '#' and line[0] != '\n':

            if '=' in line:
                entry = line.split("=")
                label = entry[0].strip()
                polarity = entry[1].strip()
                pol[label] = polarity
            else:
                entry = line.split("::")
                label = entry[0].strip()
                atomic_value = entry[1]
                match = re.search(r'[^\ ]\n$', line)
                if match:
                   atomic_value = atomic_value[:-1]
                if label in lex:
                    lex[label] += atomic_value.strip()
                else:
                    lex[label] = [atomic_value.strip()]
    f.close()
    return lex, pol


tensor_table = {
    # LIRa figure 14
    # (con,hypo):(#premises,geometry,term)
    # geometry: (f)ormula,(l)eft,(r)ight, (<)arrow to previous,
    # (v)alue, (e)context
    # term: (t)op, (b)ottom, (l)eft, (r)ight
    # "lr" with 2 premises meaning that the
    # entire term is topleft - connective - topright

    # Fusion connectives - hypothesis
    ("/",1):(2,"frleve","br"),
    ("*",1):(1,"f<lrvvv","lr"),
    ("\\",1):(2,"lfrvee","lb"),
    # Fusion connectives - conclusion
    ("/",0):(1,"lf<reev","tr"),
    ("*",0):(2,"lrfvvv","lr"),
    ("\\",0):(1,"rlf<eve","lt"),
    # Fission connectives - hypothesis
    ("(/)",1):(2,"f<rlvev","br"),
    ("(*)",1):(1,"flreee","lr"),
    ("(\\)",1):(2,"lf<revv","lb"),
    # Fission connectives - conclusion
    ("(/)",0):(1,"lfrvve","tr"),
    ("(*)",0):(2,"lrf<eee","lr"),
    ("(\\)",0):(1,"rlfvev","lt")
}


def con_pol(connective):
    c = {
    "/":'-',
```

```
    "*":'+',
    "\\":'-',
    "(/)":'+',
    "(*)":'-',
    "(\\)":'+'
    }
    return c[connective]

def term2tex(x):
    translation = {
        "mu":"\\mu",
        "comu":"\\tilde{\\mu}",
        "/|":"\\upharpoonleft",
        "|`":"\\upharpoonright",
        "<":"\\langle",
        ">":"\\rangle",
        '\\':"\\backslash",
        "(*)":"\oplus",
        "*":"\otimes",
        "(/)":"\oslash",
        "(\\)":"\obslash"
        }
    if x in translation:
        return translation[x]
    return x

def substitute_term(subs, part, term):
    for x in subs:
        if x in part:
            insertion = ['('] + term + [')']
            index = part.index(x)
            part = part[:index] + insertion + part[index+1:]
            break   # Because more than one substitution is not possible, right?
    return part
```

Code/helper_functions.py

Appendix D

Table

```python
# np    1   2   3
#  4    T   T   F
#  5    F   T   T
#  6    T   T   T

# (np2,np6) is table[2][1]
# hypotheses on x-axis
# conclusions on y-axis

import classes_linear as classes

class Table(object):

    def __init__(self, atom):
        self.hypotheses = [atom]
        self.conclusions = []
        self.table = []
        self.atom_bindings = []

    def add_hypothesis(self, atom):
        self.hypotheses.append(atom)

    def add_conclusion(self, atom):
        self.conclusions.append(atom)

    def create_table(self):
        n = len(self.hypotheses)
        self.table = [[True]*n for i in range(n)]

    # Linking two atoms both bound
    # to the same tensor leads to
    # acyclicity
    def prune_acyclicity(self):
        for x in range(0, len(self.hypotheses)):
            for y in range(0, len(self.conclusions)):
                h = self.hypotheses[x]
                c = self.conclusions[y]
                if isinstance(h.conclusion, classes.Tensor) and h.conclusion is c.hypothesis:
                    self.table[x][y] = False

    def prune_connectedness(self):
        for x in range(0, len(self.hypotheses)):
            for y in range(0, len(self.conclusions)):
                print "TODO"

    # A cotensor will only contract
    # if both of its non-main bindings
    # are bound to another tensor
```

```
    def prune_cotensor(self):
        for x in range(0, len(self.hypotheses)):
            for y in range(0, len(self.conclusions)):
                h = self.hypotheses[x]
                c = self.conclusions[y]
                cH = c.hypothesis
                hC = h.conclusion
                if h.is_lexical_item() and isinstance(cH, classes.Tensor):
                    if cH.is_cotensor() and cH.arrow != c.alpha:
                        self.table[x][y] = False
                elif c.is_lexical_item() and isinstance(hC, classes.Tensor):
                    if hC.is_cotensor() and hC.arrow != h.alpha:
                        self.table[x][y] = False

    def combine(self):
        self.atom_bindings = self.dfs(0,[],[])

    # Depth-first search, exhaustive
    def dfs(self, x, explored, combination):
        if x == len(self.hypotheses):
            return [combination]
        answers = []
        for y in range(len(self.conclusions)):
            if y not in explored and self.table[x][y]:
                combo = (self.hypotheses[x].alpha, self.conclusions[y].alpha)
                c = self.dfs(x+1, explored + [y], combination + [combo])
                if c != None:
                    answers += c
        return answers
```

Code/table.py

Appendix E

Graph

```python
# Working assumptions :
# 1 - All components are connected by mu/comu-links
# 2 - All components have a single command link attached (not true)

import classes_linear as classes
from helper_functions import *
import term


class Graph(object):

    def __init__(self, components, cotensors, mu_comu, command):
        self.components = components
        self.cotensors = cotensors
        self.mu_comu = mu_comu
        self.command = command
        self.component_nodes = [None for x in components]
        self.cotensor_nodes = [None for x in cotensors]
        self.mu_comu_edges = [None for x in mu_comu]
        self.command_edges = [None for x in command]
        for c in components:
            self.add_component_node(c, components.index(c))
        for co in cotensors:
            self.add_cotensor_node(co, cotensors.index(co))
        for m in mu_comu:
            self.add_mu_comu_edge(m, mu_comu.index(m))
        for comm in command:
            self.add_command_edge(comm, command.index(comm))

        for co in self.cotensor_nodes:
            co.get_attached()

    def add_component_node(self, c, i):
        component_node = Component(self, c, i)
        self.component_nodes[i] = component_node

    def add_cotensor_node(self, c, i):
        cotensor_node = Cotensor(self, c, i)
        self.cotensor_nodes[i] = cotensor_node

    def add_mu_comu_edge(self, m, i):
        mu_comu_edge = Mu_Comu(self, m, i)
        self.mu_comu_edges[i] = mu_comu_edge

    def add_command_edge(self, c, i):
        command_edge = Command(self, c, i)
        self.command_edges[i] = command_edge

    def get_starting_point(self, mu_vis):
```

```
        return [x for x in self.component_nodes if x.get_outgoing(mu_vis)]

    def match(self):
        return self.recursive_match([],{},[],[],[],[])

    def recursive_match(self, match, subs, comp_vis, cot_vis, comm_vis, mu_vis):

        if [x for x in self.mu_comu_edges if not x in mu_vis]:

            comp = self.get_starting_point(mu_vis)
            if not comp:
                comp = [x for x in self.component_nodes if not x in comp_vis]

            temp_match = []

            for c in comp:
                y = self.match_body(c, match, subs, comp_vis, cot_vis, comm_vis, mu_vis)
                if y:
                    temp_match.extend(y)
            return temp_match
        return [match]

    def match_body(self, comp, match, subs, comp_vis, cot_vis, comm_vis, mu_vis):

        c_match = match[:]
        compvis = comp_vis[:]
        cotvis = cot_vis[:]
        commvis = comm_vis[:]
        muvis = mu_vis[:]

        # Temporary hack, should not be allowed
        if not hasattr(comp, 'command'):
            return [match]

        comm = comp.command
        if comp in compvis:
            comm = subs[comp].command
            compvis.append(subs[comp])
        else:
            compvis.append(comp)

        if comm in commvis:
            return []

        c_match.append(comm.command)
        commvis.append(comm)

        for c in [x for x in self.cotensor_nodes if not x in cotvis]:
            if c.attachable(compvis + cotvis + commvis + muvis):
                c_match.append(c.cotensor)
                cotvis.append(c)

        m = []
        outgoing = False
        if comp.get_outgoing(muvis):
            m = comp.get_outgoing(muvis)
            outgoing = True
        else:
            leftover_mu = [x for x in self.mu_comu_edges if not x in muvis]
            for mu in leftover_mu:
                if mu.origin in compvis + cotvis:
                    m.append(mu)
                elif mu.destination in compvis + cotvis:
                    m.append(mu)

        if not m:
            return []

        temp_match = []
        for mu in m:
            x = c_match + [mu.mu_comu]
            mvis = muvis + [mu]
            s = {}
            for k,v in subs.items():
```

```
                s[k] = v
            if outgoing:
                s[comp] = mu.destination
            y = self.recursive_match(x, s, compvis, cotvis, commvis, mvis)
            if y:
                temp_match.extend(y)

        return temp_match

    def to_TeX(self, matching, cgraph):
        f = open('formula.tex', 'a')
        f.write("{\\scalefont{0.7}\n")
        f.write("\\begin{tikzpicture}\n")
        f.write("\\node [mybox] (box){\n")
        f.write("\\begin{minipage}{0.70\\textwidth}\n")
        f.write("\\begin{center}\n")

        non_empty_match = [x for x in matching if not x == []]

        if not non_empty_match:
            f.write('$' + operators_to_TeX(cgraph.main.hypothesis) + '$')

        for m in non_empty_match:

            term = self.linear_term(m)

            f.write("$")

            for x in term:
                f.write(term2tex(x))
                f.write(" ")
            f.write("$\n\n")
            f.write("\\vspace{5mm}\n")

        f.write("\end{center}\n")
        f.write("\end{minipage}\n\n};\n")
        f.write("\end{tikzpicture}}\n")
        f.close()

    def linear_term(self, m):
        term = []
        subs = []

        while m:
            # Command
            comlink = m.pop(0)
            left = comlink.top.get_term(False).term2list()
            right = comlink.bottom.get_term(True).term2list()
            harpoon = ['/|']
            if comlink.positive():
                harpoon = ['|`']
            # TODO: substitutions (method of Term object?)
                left = substitute_term(subs, left, term)

            else:
                right = substitute_term(subs, right, term)

            term = ['<'] + left + harpoon + right + ['>']

            # (Possible) Cotensor(s)
            while isinstance(m[0], classes.Tensor):
                cotensor = m.pop(0)
                term = cotensor.get_term().term2list() + ['.'] + term

            # Mu / Comu
            mulink = m.pop(0)
            mu = []
            source = None
            target = None
            if mulink.positive():
                mu = ["comu"]
                source = mulink.bottom.get_term(True)
                target = mulink.top.get_term(False)
            else:
```

```
                mu = ["mu"]
                source = mulink.top.get_term(False)
                target = mulink.bottom.get_term(True)

            term = mu + source.term2list() + ['.'] + term
            subs.extend(target.term2list())
        return term


class Node(object):

    def __init__(self):
        print "error"


class Component(Node):

    def __init__(self, g, component, index):
        self.index = index
        self.graph = g
        self.component = component
        self.outgoing_mu_comu = []

    def set_command(self, command):
        self.command = command

    def add_outgoing_mu_comu(self, m):
        self.outgoing_mu_comu.append(m)

    def get_outgoing(self, mu_vis):
        return [x for x in self.outgoing_mu_comu if not x in mu_vis]


class Cotensor(Node):

    def __init__(self, g, cotensor, index):
        self.index = index
        self.graph = g
        self.cotensor = cotensor
        self.attached = []

    def get_attached(self):
        [t1, t2] = self.cotensor.non_main_connections()
        i1 = t1
        i2 = t2
        for c in self.graph.components:
            if t1 in c:
                i1 = self.graph.component_nodes[self.graph.components.index(c)]
            if t2 in c:
                i2 = self.graph.component_nodes[self.graph.components.index(c)]

        attach = [i1, i2]

        for x,i in enumerate(attach):
            if isinstance(i, classes.Link):
                if i.is_command():
                    attach[x] = self.graph.command_edges[self.graph.command.index(i)]
                else:
                    attach[x] = self.graph.mu_comu_edges[self.graph.mu_comu.index(i)]

        self.attached = attach

    def attachable(self, visited):
        if not [x for x in self.attached if not x in visited]:
            return True
        return False


class Edge(object):

    def __init__(self):
        print "error"

    def set_origin_and_destination(self, l):
```

```
        origin = None
        destination = None
        t = l.top
        b = l.bottom
        if isinstance(t.hypothesis, classes.Tensor):
            for c in self.graph.components:
                if t.hypothesis in c:
                    t = self.graph.components.index(c)
                    break
            else:   # t is a cotensor
                t = t.hypothesis
        if isinstance(b.conclusion, classes.Tensor):
            for c_ in self.graph.components:
                if b.conclusion in c_:
                    b = self.graph.components.index(c_)
                    break
            else:   # b is a cotensor
                b = b.conclusion

        if l.positive():
            origin = b
            destination = t
        else:
            origin = t
            destination = b
        if l.is_command():
            temp = origin
            origin = destination
            destination = temp

        self.origin = origin
        self.destination = destination
        if isinstance(origin, classes.Tensor):
            self.origin = self.graph.cotensor_nodes[self.graph.cotensors.index(origin)]
        if isinstance(destination, classes.Tensor):
            self.destination = self.graph.cotensor_nodes[self.graph.cotensors.index(destination)]
        if isinstance(origin, int):  # component
            self.origin = self.graph.component_nodes[origin]
        if isinstance(destination, int):  # component
            self.destination = self.graph.component_nodes[destination]


class Mu_Comu(Edge):

    def __init__(self, g, mu_comu, index):
        self.index = index
        self.graph = g
        self.mu_comu = mu_comu
        self.set_origin_and_destination(mu_comu)

        # Working assumption 1
        if isinstance(self.origin, Component) and isinstance(self.destination, Component):
            self.origin.add_outgoing_mu_comu(self)

class Command(Edge):

    def __init__(self, g, command, index):
        self.index = index
        self.graph = g
        self.command = command
        self.set_origin_and_destination(command)

        # Working assumption 2
        self.graph.component_nodes[index].set_command(self)
```

Code/graph.py

Appendix F

Term

```python
# Proof terms as objects

next_alpha = 1


class Term(object):

    def __init__(self):
        print "error"


class Atomic_Term(Term):

    def __init__(self, atom=None):
        global next_alpha
        self.text = False
        if atom:
            self.atom = atom
            self.text = True
        else:
            self.atom = None

    def term2list(self):
        global next_alpha
        if self.text:
            return ['\\textrm{' + self.atom + '}']
        if not self.atom:
            self.atom = chr(96 + next_alpha)
            next_alpha += 1
        return [self.atom]


class Connective_Term(Term):

    def __init__(self, con):
        self.connective = con

    def term2list(self):
        return [self.connective]


class Complex_Term(Term):

    def __init__(self, left, middle, right):
        self.middle = middle
        self.left = left
        self.right = right

    def term2list(self):
```

```
        left = self.left.term2list()
        right = self.right.term2list()

        if isinstance(left, Complex_Term):
            left = ['('] + left + [')']

        if isinstance(right, Complex_Term):
            right = ['('] + right + [')']

        return left + self.middle.term2list() + right


class Cotensor_Term(Complex_Term):

    def __init__(self, left, right, bottom):
        self.left = left
        self.right = right
        self.bottom = bottom

    def term2list(self):

        t1 = self.left.term2list()
        t2 = self.right.term2list()
        bottom = self.bottom.term2list()

        return ['\\frac{'] + t1 + t2 + ['}{'] + bottom + ['}']
```

Code/term.py

Appendix **G**

Argparser

```
import argparse
import textwrap


class Parser(object):

    def __init__(self):
        self.p = argparse.ArgumentParser(
            formatter_class=argparse.RawDescriptionHelpFormatter,
            description = textwrap.dedent('''\
        Theorem prover for LG
        Formula language:
                A,B ::= p |                 atoms (use alphanum)
                A*B | B\A | A/B |           product
                A(*)B | A(/)B | A(\)B       coproduct
                =>                          inference

        To use LaTeX commands as atoms, use |.
        For example: |phi will be translated to \phi
        Example call: LGprover.py "np/n , n => np"'''),
            usage = 'LGprover.py [options] sequent')
        self.p.add_argument('sequent', metavar='F', type=str, nargs='+',
                            help='a formula in LG to unfold')
        self.p.add_argument('--lexicon', '-l', action = 'store',
                            help='filepath to lexicon')
        self.p.add_argument('--tex', '-t', action = 'store_true',
                        help = 'print result to LaTeX')
        self.p.add_argument('--abstract', '-a', action = 'store_true',
                        help = 'hide internal node decoration')
        self.p.add_argument('--main', '-m',
                        help = 'hide main formula as argument given')
        self.p.add_argument('--term', action = 'store_true',
                        help = 'show term(s) accordingly')
        self.p.add_argument('--rotate', '-r', action = 'store_true',
                        help = 'rotate structure 90 degrees counterclockwise')
        self.arguments = self.p.parse_args()

    def get_arguments(self):
        return self.arguments
```

Code/argparser.py

Appendix H

Sample lexicon

```
##################################################
#
#                Sample Lexicon
#
##################################################

# Polarity for atomic formulas
# Given in a different format
# Default is negative

np = +
n = +
s = -

de :: np/n
man :: n
slaapt :: np\s

test :: (a/b)*(c\d)

# Double entries don't raise errors but are not considered (yet)
man :: x

# LIRA Figure 5
from :: (s(/)s)(\)np
to :: s/(np\s)

# LIRA Figure 18
subj :: (np/n)*n
tv :: (np\s)/np
det :: np/n
noun :: n

# Time flies like an arrow
time :: np
flies :: np\s
like :: ((np\s)\(np\s))/np
an :: np/n
arrow :: n

# Embedded
mary :: np
thinks :: (np\s)/s
john :: np
likes :: (np\s)/np
nobody :: (s(/)s)(\)np
```

Code/lexicon.txt