On the complexity of the Lambek-Grishin calculus

Jeroen Bransen

Utrecht University, The Netherlands

July 12, 2010

Master's Thesis
Cognitive Artificial Intelligence
30 ECTS

Supervisors:
Michael Moortgat
Rosalie Iemhoff

Table of Contents

1 Introduction

The field of Artificial Intelligence is a broad research area in which there are two main goals. The first goal is to understand and formalize (human) intelligence, both on a neurological level as well as a conceptual level. There is a focus on cognitive functions like attention, memory, problem solving and language because it is not too hard to specify tasks that require one of these functions.

The second goal is to model this knowledge in computers, thereby simulating intelligent behaviour. This can be done on various scales, from the field of robotics to computer programs that can perform a small task. A good example is a speech recognition program, which can translate spoken text to written text.

In this thesis we will focus on the field of *Computational Linguistics*, and more specifically on the formalization of natural language. Natural languages play an important role in the field of Artificial Intelligence. Let us for example consider the following questions: How do children learn a language? Is there a universal grammar? Will computers ever be able to understand natural language? All of these questions require understanding about natural languages, and especially understanding about the structure behind natural languages.

Humans are very good in speaking, writing and understanding a language, while even the best computer systems are worse than a child. And if we think about the way children learn a language, the difference becomes even bigger: a child can learn any language by observing only correct sentences for a few years. Nobody is explaining any gramatical rules to children, nor are they correcting wrong sentences in the first few years of a childs life. Still, from this children learn to construct valid sentences that they have never heard before.

However, if we try to model this behaviour in computers we will fail to find any interesting language models. Any self-learning algorithm will require both positive and negative examples, so in this case it will also need wrong sentences to be able to distinguish between sentences and non-sentences. If we will use only correct sentences, the algorithm will just learn that any sentence is correct, which is obviously not the indended behaviour.

Therefore the idea is that humans already have some kind of built-in *Universal Grammar*. This grammar could define the structure of natural languages in general, and by hearing sentences (together with observing events in the real-world to find associations) children could fill in the grammar for a specific language like English.

The framework that we will use in this thesis is based on the idea of *Parsing as Deduction*, which is the idea that humans use a deductive system to parse natural language. The axioms and rules of inference of this deductive system then define how language can be used, so could be seen as a definition of the universal grammar. The learning of a language will then set certain parameters that represent the specific rules of this language. This deductive system could also be extended to the semantical level by means of a Curry-Howard interpretation.

In 1958, Lambek published his paper titled *The Mathematics of Sentence Structure*. In this paper he formulated the *Syntactic Calculus*, a calculus for

modelling the structure of natural languages based on the idea of parsing as deduction. The approach that Lambek took with his syntactic calculus is to define a logical framework for describing the structure of sentences. In this framework every word is assigned one or more types that define the function of a word in the sentence. The learning of a natural language like English would in this case be the process of assigning types to words, which then would provide us a method to distinguish between sentences and non-sentences. This could be the start for handling natural languages in a formalized way.

In 1961, Lambek formulated another version of the syntactic calculus: in (Lambek, 1958), types are assigned to *strings*, which are then combined by an *associative* operation; in (Lambek, 1961), types are assigned to *phrases* (bracketed strings), and the composition operation is non-associative. We refer to these two versions as **L** and **NL** respectively.

As for the theoretical generative power, Kandulski (1988) proved that **NL** defines exactly the context-free languages. Pentus (1993b) showed that this also holds for the associative sytem **L**. As for the complexity of the derivability problem, de Groote (1999) showed that for **NL** this belongs to `PTIME`; for **L**, Pentus (2003) proves that the problem is NP-complete and Savateev (2009) shows that NP-completeness also holds for the product-free fragment of **L**.

It is well known that some natural language phenomena require generative capacity beyond context-free. Several extensions of the syntactic calculus have been proposed to deal with such phenomena. In this thesis we look at the Lambek-Grishin calculus **LG** (Moortgat, 2007, 2009). **LG** is a *symmetric* extension of the nonassociative Lambek calculus **NL**. In addition to $\otimes, \backslash, /$ (product, left and right division), **LG** has dual operations $\oplus, \oslash, \oslash$ (coproduct, left and right difference). These two families are related by linear distributivity principles.

Melissen (2009) shows that all languages which are the intersection of a context-free language and the permutation closure of a context-free language are recognizable in **LG**. This places the lower bound for **LG** recognition beyond LTAG. LTAG is the class of languages generated by (Lexicalized) Tree-Adjoining Grammars (Joshi and Schabes, 1997), a grammar formalism somewhat similar to context-free grammars, but the elementary unit of rewriting is a tree rather than a symbol. The upper bound is still open.

In this thesis we will first formally define the system **LG**, and show an example of an application of the calculus. In the first part we will present results about the computational complexity of **LG** and the product-free fragment of **LG**. We will show that the derivability problem for both **LG** and the product-free fragment of **LG** is NP-complete, thereby solving two open problems.

The second part of the thesis will describe the implementation of a theorem prover for **LG** that has been written along with this thesis. We will give implementational details as well as details and examples about the usage of this theorem prover. We will conclude by describing some ideas for future work on this theorem prover.

2 Lambek-Grishin calculus

We define the formula language of **LG** as follows.

Let Var be a set of *primitive types*, we use lowercase letters to refer to an element of Var. Let *formulas* be constructed using primitive types and the binary connectives $\otimes$, $/$, $\backslash$, $\oplus$, $\oslash$ and $\otimes$ as follows:

$$A, B ::= p \mid A \otimes B \mid A/B \mid B\backslash A \mid A \oplus B \mid A \oslash B \mid B \otimes A$$

The sets of *input* and *output structures* are constructed using formulas and the binary structural connectives $\cdot \otimes \cdot$, $\cdot/\cdot$, $\cdot\backslash\cdot$, $\cdot \oplus \cdot$, $\cdot \oslash \cdot$ and $\cdot \otimes \cdot$ as follows:

$$\text{(input)} \qquad X, Y ::= A \mid X \cdot \otimes \cdot Y \mid X \cdot \oslash \cdot P \mid P \cdot \otimes \cdot X$$

$$\text{(output)} \qquad P, Q ::= A \mid P \cdot \oplus \cdot Q \mid P \cdot / \cdot X \mid X \cdot \backslash \cdot P$$

The *sequents* of the calculus are of the form $X \to P$, and as usual we write $\vdash_{LG} X \to P$ to indicate that the sequent $X \to P$ is derivable in **LG**. The axioms and inference rules are presented in Figure 1, we use the *display logic* from (Goré, 1998), but with different symbols for the *structural connectives*.

It has been proven by Moortgat (2007) that we have *Cut admissibility* for **LG**. This means that we can transform every derivation using the *Cut*-rule into a corresponding derivation that is *Cut-free*. Therefore we will assume that the Cut-rule is not needed anywhere in a derivation.

2.1 Example

We will now give a short example of a linguistic application that shows how **LG** can be used. Let us define the following lexicon of words:

$everybody ::= s/(np\backslash s)$
$teases ::= (np\backslash s)/np$
$someone ::= (s \oslash s) \otimes np$

Here we have chosen to use np as the primitive type for a *noun phrase*, and s as the primitive type for a *sentence*. A formula A/B (resp. $B\backslash A$) can be explained as follows: If this formula is followed (resp. preceded) by an expression that produces B, then it produces A. The word *teases* can therefore be explained as *a word that can form a sentence if it is followed by an expression that produces a noun phrase and preceded by an expression that produces a noun phrase*. This is exactly as one would expect from a transitive verb like *teases*.

For $\oslash$ and $\otimes$ it is harder to give a clear explanation in general, but the type for *someone* can be explained as: *within the context of a sencence, "someone" will produce a noun phrase and take over the scope of the sentence*. This use will become clear in the following example.

$$\frac{}{p \to p}\ Ax$$

$$\frac{X \to A \quad A \to P}{X \to P}\ Cut$$

$$\frac{\dfrac{Y \to X \cdot \backslash \cdot P}{X \cdot \otimes \cdot Y \to P}\ r}{X \to P \cdot / \cdot Y}\ r \qquad \frac{\dfrac{X \cdot \oslash \cdot Q \to P}{X \to P \cdot \oplus \cdot Q}\ dr}{P \cdot \oslash \cdot X \to Q}\ dr$$

(a) Display rules

$$\frac{X \cdot \otimes \cdot Y \to P \cdot \oplus \cdot Q}{X \cdot \oslash \cdot Q \to P \cdot / \cdot Y}\ d \oslash /$$

$$\frac{X \cdot \otimes \cdot Y \to P \cdot \oplus \cdot Q}{Y \cdot \oslash \cdot Q \to X \cdot \backslash \cdot P}\ d \oslash \backslash$$

$$\frac{X \cdot \otimes \cdot Y \to P \cdot \oplus \cdot Q}{P \cdot \oslash \cdot X \to Q \cdot / \cdot Y}\ d \oslash /$$

$$\frac{X \cdot \otimes \cdot Y \to P \cdot \oplus \cdot Q}{P \cdot \oslash \cdot Y \to X \cdot \backslash \cdot Q}\ d \oslash \backslash$$

(b) Distributivity rules (Grishin interaction principles)

$$\frac{A \cdot \otimes \cdot B \to P}{A \otimes B \to P}\ \otimes L \qquad \frac{X \to B \cdot \oplus \cdot A}{X \to B \oplus A}\ \oplus R$$

$$\frac{X \to A \cdot / \cdot B}{X \to A/B}\ /R \qquad \frac{B \cdot \oslash \cdot A \to P}{B \oslash A \to P}\ \oslash L$$

$$\frac{X \to B \cdot \backslash \cdot A}{X \to B \backslash A}\ \backslash R \qquad \frac{A \cdot \oslash \cdot B \to P}{A \oslash B \to P}\ \oslash L$$

$$\frac{X \to A \qquad Y \to B}{X \cdot \otimes \cdot Y \to A \otimes B}\ \otimes R \qquad \frac{B \to P \qquad A \to Q}{B \oplus A \to P \cdot \oplus \cdot Q}\ \oplus L$$

$$\frac{X \to A \qquad B \to P}{B/A \to P \cdot / \cdot X}\ /L \qquad \frac{X \to B \qquad A \to P}{P \cdot \oslash \cdot X \to A \oslash B}\ \oslash R$$

$$\frac{X \to A \qquad B \to P}{A \backslash B \to X \cdot \backslash \cdot P}\ \backslash L \qquad \frac{X \to B \qquad A \to P}{X \cdot \oslash \cdot P \to B \oslash A}\ \oslash R$$

(c) Logical rules

Fig. 1: The Lambek-Grishin calculus inference rules

The sentence *Everybody teases someone* is an ambiguous sentence, for which there exist two readings. If we use the given formulas for the words and a target type s, we can construct the following sequent:

$$(s/(np\backslash s)) \cdot \otimes \cdot (((np\backslash s)/np) \cdot \otimes \cdot ((s \oslash s) \oslash np)) \to s$$

For this sequent there are two different derivations, that correspond to the different readings of the sentence. The derivations are given in Figure 2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{s \to s}\quad \dfrac{\dfrac{\dfrac{}{np \to np}\quad \dfrac{}{np \to np}}{np\backslash s \to np\backslash s}\backslash R,\backslash L}{(np\backslash s)/np \to (np\backslash s)\cdot /\cdot np}r}{((np\backslash s)/np)\cdot\otimes\cdot np \to np\backslash s}/L}{s/(np\backslash s) \to s\cdot /\cdot (((np\backslash s)/np)\cdot\otimes\cdot np)}r}{(s/(np\backslash s))\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot np)\to s}\quad s\to s}{((s/(np\backslash s))\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot np))\cdot\oslash\cdot s\to s\oslash s}\oslash R}{(s/(np\backslash s))\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot np)\to(s\oslash s)\cdot\oplus\cdot s}dr}{(s\oslash s)\cdot\oslash\cdot((s/(np\backslash s))\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot np))\to s}dr}{\underbrace{(s/(np\backslash s))}_{\text{Everybody}}\cdot\otimes\cdot(\underbrace{((np\backslash s)/np)}_{\text{teases}}\cdot\otimes\cdot(\underbrace{(s\oslash s)\oslash np}_{\text{someone}}))\to s}\oslash L, Gr$$

(a) $\exists x.\forall y.((\text{teases } x)\ y)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{s\to s}\quad \dfrac{\dfrac{\dfrac{}{np\to np}}{np\backslash s\to np\cdot\backslash\cdot s}\backslash L\quad \dfrac{}{np\to np}}{(np\backslash s)/np\to(np\cdot\backslash\cdot s)\cdot/\cdot np}/L}{((np\backslash s)/np)\cdot\otimes\cdot np\to np\cdot\backslash\cdot s}r}{np\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot np)\to s}r}{(np\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot np))\cdot\oslash\cdot s\to s\oslash s}\oslash R}{np\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot np)\to(s\oslash s)\cdot\oplus\cdot s}dr}{(s\oslash s)\cdot\oslash\cdot(np\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot np))\to s}dr}{np\cdot\otimes\cdot(((np\backslash s)/np)\cdot\otimes\cdot((s\oslash s)\oslash np))\to s}\oslash L, Gr}{((np\backslash s)/np)\cdot\otimes\cdot((s\oslash s)\oslash np)\to np\backslash s}\backslash R, r\qquad s\to s}{s/(np\backslash s)\to s\cdot/\cdot(((np\backslash s)/np)\cdot\otimes\cdot((s\oslash s)\oslash np))}/L}{\underbrace{(s/(np\backslash s))}_{\text{Everybody}}\cdot\otimes\cdot(\underbrace{((np\backslash s)/np)}_{\text{teases}}\cdot\otimes\cdot(\underbrace{(s\oslash s)\oslash np}_{\text{someone}}))\to s}r$$

(b) $\forall y.\exists x.((\text{teases } x)\ y)$

Fig. 2: Two readings for *Everybody teases someone*

Part I

Theoretical results

3 Preliminaries

3.1 Derivation length

We will first show that for every derivable sequent there exists a Cut-free derivation that is polynomial in the length of the sequent. The length of a sequent φ, denoted as $|\varphi|$, is defined as the number of connectives used to construct this sequent. A subscript will be used to indicate that we count only certain connectives, for example $|\varphi|_{\otimes}$.

Lemma 1. *If $\vdash_{LG} \varphi$ there exists a derivation with exactly $|\varphi|$ logical rules.*

Proof. If $\vdash_{LG} \varphi$ then there exists a Cut-free derivation for φ. Because every logical rule removes 1 logical connective and there are no rules that introduce logical connectives, this derivation contains $|\varphi|$ logical rules. □

Lemma 2. *If $\vdash_{LG} \varphi$ there exists a derivation with at most $\frac{1}{4}|\varphi|^2$ Grishin interactions.*

Proof. Let us take a closer look at the Grishin interaction principles. First of all, it is not hard to see that the interactions are irreversible. Also note that the interactions happen between the families of input connectives $\{\otimes, /, \backslash\}$ and output connectives $\{\oplus, \oslash, \obslash\}$ and that the Grishin interaction principles are the only rules of inference that apply on both families. So, on any pair of 1 input and 1 output connective, at most 1 Grishin interaction principle can be applied.

If $\vdash_{LG} \varphi$ there exists a Cut-free derivation of φ. The maximum number of possible Grishin interactions in 1 Cut-free derivation is reached when a Grishin interaction is applied on every pair of 1 input and 1 output connective. Thus, the maximum number of Grishin interactions in 1 Cut-free derivation is $|\varphi|_{\{\otimes,/,\backslash\}} \cdot |\varphi|_{\{\oplus,\oslash,\obslash\}}$.

By definition, $|\varphi|_{\{\otimes,/,\backslash\}} + |\varphi|_{\{\oplus,\oslash,\obslash\}} = |\varphi|$, so the maximum value of $|\varphi|_{\{\otimes,/,\backslash\}} \cdot |\varphi|_{\{\oplus,\oslash,\obslash\}}$ is reached when $|\varphi|_{\{\otimes,/,\backslash\}} = |\varphi|_{\{\oplus,\oslash,\obslash\}} = \frac{|\varphi|}{2}$. Then the total number of Grishin interactions in 1 derivation is $\frac{|\varphi|}{2} \cdot \frac{|\varphi|}{2} = \frac{1}{4}|\varphi|^2$, so any Cut-free derivation of φ will contain at most $\frac{1}{4}|\varphi|^2$ Grishin interactions. □

Lemma 3. *If $\vdash_{LG} \varphi$ there exists a Cut-free derivation of length $O(|\varphi|^3)$.*

Proof. From Lemma 1 and Lemma 2 we know that there exists a derivation with at most $|\varphi|$ logical rules and $\frac{1}{4}|\varphi|^2$ Grishin interactions. Thus, the derivation consists of $|\varphi| + \frac{1}{4}|\varphi|^2$ rules, with between each pair of consecutive rules the display rules. It can be easily seen that at most $2|\varphi|$ display rules are needed to display any of the structural parts. So, at most $2|\varphi| \cdot (|\varphi| + \frac{1}{4}|\varphi|^2) = 2|\varphi|^2 + \frac{1}{2}|\varphi|^3$ derivation steps are needed in the shortest possible Cut-free derivation for this sequent, and this is in $O(|\varphi|^3)$. □

3.2 Additional notations

Let us first introduce some additional notations to make the proofs shorter and easier to read.

Let us call an input structure X which does not contain any structural connectives except for $\cdot \otimes \cdot$ a $\otimes$-*structure*. A $\otimes$-structure can be seen as a binary tree with $\cdot \otimes \cdot$ in the internal nodes and formulas in the leafs. Formally we define $\otimes$-structures U and V as:

$$U, V ::= A \mid U \cdot \otimes \cdot V$$

We define $X[]$ and $P[]$ as the input and output structures X and P with a hole in one of their leafs. Formally:

$$X[] ::= [] \mid X[] \cdot \otimes \cdot Y \mid Y \cdot \otimes \cdot X[] \mid X[] \cdot \oslash \cdot Q \mid Y \cdot \oslash \cdot P[] \mid Q \cdot \otimes\!\!\backslash \cdot X[] \mid P[] \cdot \otimes\!\!\backslash \cdot Y$$

$$P[] ::= [] \mid P[] \cdot \oplus \cdot Q \mid Q \cdot \oplus \cdot P[] \mid P[] \cdot / \cdot Y \mid Q \cdot / \cdot X[] \mid Y \cdot \backslash \cdot P[] \mid X[] \cdot \backslash \cdot Q$$

This notation is similar to the one of de Groote (1999) but with structures. If $X[]$ is a structure with a hole, we write $X[Y]$ for $X[]$ with its hole filled with structure Y. We will write $X^{\otimes}[]$ for a $\otimes$-structure with a hole.

Furthermore, we extend the definition of hole to formulas, and define $A[]$ as a *formula* A with a hole in it, in a similar manner as for structures. Hence, by $A[B]$ we mean the formula $A[]$ with its hole filled by formula B.

In order to distinguish between input and output polarity formulas, we write $A^{\bullet}$ for a formula with *input* polarity and A° for a formula with *output* polarity. Note that for structures this is already defined by using X and Y for input polarity and P and Q for output polarity. This can be extended to formulas in a similar way, and we will this notation only in cases where the polarity is not clear from the context.

3.3 Derived rules of inference

Now we will show and prove some derived rules of inference of **LG**.

Lemma 4. *If $\vdash_{LG} A \to B$ and we want to prove $X^{\otimes}[A] \to P$, we can* replace *A by B in $X^{\otimes}[]$. We have the inference rule below:*

$$\frac{A \to B \quad X^{\otimes}[B] \to P}{X^{\otimes}[A] \to P}\ \mathit{Repl}$$

Proof. We consider 3 cases:

1. If $X^{\otimes}[A] = A$, it is simply the cut-rule:

   $$\frac{A \to B \quad B \to P}{A \to P}\ \mathit{Cut}$$

2. If $X^{\otimes}[A] = Y^{\otimes}[A] \cdot \otimes \cdot V$, we can move V to the righthand-side and use induction to prove the sequent:

$$\dfrac{A \to B \quad \dfrac{\dfrac{Y^{\otimes}[B] \cdot \otimes \cdot V \to P}{Y^{\otimes}[B] \to P \cdot / \cdot V}\, r}{Y^{\otimes}[A] \to P \cdot / \cdot V}\, Repl}{Y^{\otimes}[A] \cdot \otimes \cdot V \to P}\, r$$

3. If $X^{\otimes}[A] = U \cdot \otimes \cdot Y^{\otimes}[A]$, we can move U to the righthand-side and use induction to prove the sequent:

$$\dfrac{A \to B \quad \dfrac{\dfrac{U \cdot \otimes \cdot Y^{\otimes}[B] \to P}{Y^{\otimes}[B] \to U \cdot \backslash \cdot P}\, r}{Y^{\otimes}[A] \to U \cdot \backslash \cdot P}\, Repl}{U \cdot \otimes \cdot Y^{\otimes}[A] \to P}\, r$$

□

Lemma 5. *If we want to prove $X^{\otimes}[A \oslash B] \to P$, then we can* move *the expression $\oslash B$ out of the $\otimes$-structure. We have the inference rule below:*

$$\dfrac{X^{\otimes}[A] \cdot \oslash \cdot B \to P}{X^{\otimes}[A \oslash B] \to P}\, Move$$

Proof. We consider 3 cases:

1. If $X^{\otimes}[A \oslash B] = A \oslash B$, then this is simply the $\oslash L$-rule:

$$\dfrac{A \cdot \oslash \cdot B \to Y}{A \oslash B \to Y}\, \oslash L$$

2. If $X^{\otimes}[A \oslash B] = Y^{\otimes}[A \oslash B] \cdot \otimes \cdot V$, we can move V to the righthand-side and use induction together with the Grishin interaction principles to prove the sequent:

$$\dfrac{\dfrac{\dfrac{\dfrac{(Y^{\otimes}[A] \cdot \otimes \cdot V) \cdot \oslash \cdot B \to P}{Y^{\otimes}[A] \cdot \otimes \cdot V \to P \cdot \oplus \cdot B}\, dr}{Y^{\otimes}[A] \cdot \oslash \cdot B \to P \cdot / \cdot V}\, d\oslash /}{Y^{\otimes}[A \oslash B] \to P \cdot / \cdot V}\, Move}{Y^{\otimes}[A \oslash B] \cdot \otimes \cdot V \to P}\, r$$

3. If $X^{\otimes}[A \oslash B] = U \cdot \otimes \cdot Y^{\otimes}[A \oslash B]$, we can move U to the righthand-side and use induction together with the Grishin interaction principles to prove the sequent:

$$\dfrac{\dfrac{\dfrac{\dfrac{(U \cdot \otimes \cdot Y^{\otimes}[A]) \cdot \oslash \cdot B \to P}{U \cdot \otimes \cdot Y^{\otimes}[A] \to P \cdot \oplus \cdot B}\, dr}{Y^{\otimes}[A] \cdot \oslash \cdot B \to U \cdot \backslash \cdot P}\, d\oslash \backslash}{Y^{\otimes}[A \oslash B] \to U \cdot \backslash \cdot P}\, Move}{U \cdot \otimes \cdot Y^{\otimes}[A \oslash B] \to P}\, r$$

□

Lemma 6. $\vdash_{LG} A \otimes B \rightarrow P$ *iff* $\vdash_{LG} A \cdot \otimes \cdot B \rightarrow P$

Proof. The *if*-part is trivial, this is simply the $\otimes L$ rule:

$$\frac{A \cdot \otimes \cdot B \rightarrow P}{A \otimes B \rightarrow P} \otimes L$$

The *only-if*-part can be derived as follows:

$$\frac{\dfrac{A \rightarrow A \quad B \rightarrow B}{A \cdot \otimes \cdot B \rightarrow A \otimes B} \otimes R \qquad A \otimes B \rightarrow P}{A \cdot \otimes \cdot B \rightarrow P} Cut$$

Note that because of the Cut elimination theorem, there exists a cut-free derivation for every possible choice of A, B and P.

□

3.4 Type similarity

The type simililarity relation $\sim$, introduced by Lambek (1958), is the reflexive transitive symmetric closure of the derivability relation. Formally we define this as:

Definition 1. $A \sim B$ iff *there exists a sequence* $C_1 \ldots C_n (1 \leq i \leq n)$ *such that* $C_1 = A$, $C_n = B$ *and* $C_i \rightarrow C_{i+1}$ *or* $C_{i+1} \rightarrow C_i$ *for all* $1 \leq i < n$.

It was proved by Lambek that $A \sim B$ *iff* one of the following equivalent statements holds (the so-called *diamond property*):

$$\exists C \text{ such that } A \rightarrow C \text{ and } B \rightarrow C \quad \text{(join)}$$

$$\exists D \text{ such that } D \rightarrow A \text{ and } D \rightarrow B \quad \text{(meet)}$$

Definition 2. *If* $A \sim B$ *and* C *is the* join *type of* A *and* B *so that* $A \rightarrow C$ *and* $B \rightarrow C$, *we define* $A \overset{C}{\sqcap} B = (A/((C/C)\backslash C)) \otimes ((C/C)\backslash B)$ *as the meet type of* A *and* B.

This is also the solution given by Lambek (1958) for the associative system **L**, but in fact this is the shortest solution for the non-associative system **NL** (Foret, 2003).

Lemma 7. *If* $A \sim B$ *with join-type* C *and* $\vdash_{LG} A \rightarrow P$ *or* $\vdash_{LG} B \rightarrow P$, *then we also have* $\vdash_{LG} A \overset{C}{\sqcap} B \rightarrow P$. *We can write this as a derived rule of inference:*

$$\frac{A \rightarrow P \quad or \quad B \rightarrow P}{A \overset{C}{\sqcap} B \rightarrow P} Meet$$

Proof.

1. If $A \to P$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{C \to C \quad C \to C}{C/C \to C \cdot / \cdot C}\,/L}{C/C \to C/C}\,/R \qquad B \to C}{(C/C)\backslash B \to (C/C) \cdot \backslash \cdot C}\,\backslash L}{(C/C)\backslash B \to (C/C)\backslash C}\,\backslash R \qquad A \to P}{A/((C/C)\backslash C) \to P \cdot / \cdot ((C/C)\backslash B)}\,/L}{(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P}\,r}{(A/((C/C)\backslash C)) \otimes ((C/C)\backslash B) \to P}\,\otimes L$$

2. If $B \to P$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{C \to C \quad C \to C}{C/C \to C \cdot / \cdot C}\,/L}{(C/C) \cdot \otimes \cdot C \to C}\,r}{C \to (C/C) \cdot \backslash \cdot C}\,r \qquad A \to C}{C \to (C/C)\backslash C}\,\backslash R}{A/((C/C)\backslash C) \to C \cdot / \cdot C}\,/L}{A/((C/C)\backslash C) \to C/C}\,/R \qquad B \to P}{(C/C)\backslash B \to (A/((C/C)\backslash C)) \cdot \backslash \cdot P}\,\backslash L}{(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P}\,r}{(A/((C/C)\backslash C)) \otimes ((C/C)\backslash B) \to P}\,\otimes L$$

□

The following lemma is the key lemma of this paper, and its use will become clear to the reader in the construction of Section 4.

Lemma 8. *If* $\vdash_{LG} A \overset{C}{\sqcap} B \to P$ *then* $\vdash_{LG} A \to P$ *or* $\vdash_{LG} B \to P$*, if it is not the case that:*

- $P = P'[A'[(A_1 \otimes A_2)^\circ]]$
- $\vdash_{LG} A/((C/C)\backslash C) \to A_1$
- $\vdash_{LG} (C/C)\backslash B \to A_2$

Proof. We have that $\vdash_{LG} (A/((C/C)\backslash C)) \otimes ((C/C)\backslash B) \to P$, so from Lemma 6 we know that $\vdash_{LG} (A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P$. Remark that this also means that there exists a cut-free derivation for this sequent. By induction on the length of the derivation we will show that *if* $\vdash_{LG} (A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P$, *then* $\vdash_{LG} A \to P$ or $\vdash_{LG} B \to P$, under the assumption that P is not of the form that is explicitly excluded in this lemma.

The induction base is the case where a logical rule is applied on the lefthand-side of the sequent. At a certain point in the derivation, possibly when P is an atom, one of the 3 following rules must be applied:

1. The $\otimes R$ rule, but then $P = A_1 \otimes A_2$ and in order to come to a derivation it must be the case that $\vdash_{LG} A/((C/C)\backslash C) \to A_1$ and $\vdash_{LG} (C/C)\backslash B \to A_2$. However, this is explicitly excluded in this lemma so this can never be the case.
2. The $/L$ rule, in this case first the r rule is applied so that we have $\vdash_{LG} A/((C/C)\backslash C) \to P \cdot / \cdot ((C/C)\backslash B)$. Now if the $/L$ rule is applied, we must have that $\vdash_{LG} A \to P$.
3. The $\backslash L$ rule, in this case first the r rule is applied so that we have $\vdash_{LG} (C/C)\backslash B \to (A/((C/C)\backslash C)) \cdot \backslash \cdot P$. Now if the $\backslash L$ rule is applied, we must have that $\vdash_{LG} B \to P$.

The induction step is the case where a logical rule is applied on the righthand-side of the sequent. Let $\delta = \{r, dr, d \oslash /, d \oslash \backslash, d \otimes /, d \otimes \backslash\}$ and let δ^* indicate a (possibly empty) sequence of structural residuation steps and Grishin interactions. For example for the $\oslash R$ rule there are 2 possibilities:

– The lefthand-side ends up in the first premisse of the $\oslash R$ rule:

$$\dfrac{\dfrac{\dfrac{(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P''[A']}{P'[(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B)] \to A'}\ \delta^* \quad B' \to Q}{P'[(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B)] \cdot \oslash \cdot Q \to A' \oslash B'}\ \oslash R}{(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P[A' \oslash B']}\ \delta^*$$

In order to be able to apply the $\oslash R$ rule, we need to have a formula of the form $A' \oslash B'$ on the righthand-side. In the first step all structural rules are applied to display this formula in the righthand-side, and we assume that in the lefthand-side the meet-type ends up in the first structural part (inside a structure with the remaining parts from P that we call P'). After the $\oslash R$ rule has been applied, we can again display our meet-type in the lefthand-side of the formula by moving all other structural parts from P' back to the righthand-side (P'').
In this case it must be that $\vdash_{LG} (A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P''[A']$, and by induction we know that in this case also $\vdash_{LG} A \to P''[A']$ or $\vdash_{LG} B \to P''[A']$. In the case that $\vdash_{LG} A \to P''[A']$, we can show that $\vdash_{LG} A \to P[A' \oslash B']$ as follows:

$$\dfrac{\dfrac{\dfrac{A \to P''[A']}{P'[A] \to A'}\ \delta^* \quad B' \to Q}{P'[A] \cdot \oslash \cdot Q \to A' \oslash B'}\ \oslash R}{A \to P[A' \oslash B']}\ \delta^*$$

The case for B is similar.

– The lefthand-side ends up in the second premisse of the $\oslash R$ rule:

$$\dfrac{\dfrac{Q \to A' \quad \dfrac{(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P''[B']}{B' \to P'[(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B)]}\ \delta^*}{Q \cdot \oslash \cdot P'[(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B)] \to A' \oslash B'}\ \oslash R}{(A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \to P[A' \oslash B']}\ \delta^*$$

This case is similar to the other case, except that the meet-type ends up in the other premisse. Note that, although in this case it is temporarily moved to the righthand-side, the meet-type will still be in an input polarity position and can therefore be displayed in the lefthand-side again.
In this case it must be that $\vdash_{LG} (A/((C/C)\backslash C)) \cdot \otimes \cdot ((C/C)\backslash B) \rightarrow P''[B']$, and by induction we know that in this case also $\vdash_{LG} A \rightarrow P''[B']$ or $\vdash_{LG} B \rightarrow P''[B']$. In the case that $\vdash_{LG} A \rightarrow P''[B']$, we can show that $\vdash_{LG} A \rightarrow P[A' \oslash B']$ as follows:

$$\dfrac{\dfrac{Q \rightarrow A' \quad \dfrac{A \rightarrow P''[B']}{B' \rightarrow P'[A]}\,\delta^*}{Q \cdot \oslash \cdot P'[A] \rightarrow A' \oslash B'}\,\oslash R}{A \rightarrow P[A' \oslash B']}\,\delta^*$$

The case for B is similar.

The cases for the other logical rules are similar. □

4 Reduction from SAT to LG

In this section we will show that we can reduce a Boolean formula in conjunctive normal form to a sequent of the *Lambek-Grishin calculus*, so that the corresponding **LG** sequent is provable *if and only if* the CNF formula is satisfiable. This has already been done for the associative system **L** by Pentus (2003) with a similar construction.

Let $\varphi = c_1 \wedge \ldots \wedge c_n$ be a Boolean formula in conjunctive normal form with clauses $c_1 \ldots c_n$ and variables $x_1 \ldots x_m$. For all $1 \leq j \leq m$ let $\neg_0 x_j$ stand for the literal $\neg x_j$ and $\neg_1 x_j$ stand for the literal x_j. Now $\langle t_1, \ldots, t_m \rangle \in \{0,1\}^m$ is a satisfying assignment for φ if and only if for every $1 \leq i \leq n$ there exists a $1 \leq j \leq m$ such that the literal $\neg_{t_j} x_j$ appears in clause c_i.

Let p_i (for $1 \leq i \leq n$) be distinct primitive types from Var. We now define the following families of types:

$$E^i_j(t) \leftrightharpoons \begin{cases} p_i \oslash (p_i \oslash p_i) & \text{if } \neg_t x_j \text{ appears in clause } c_i \\ p_i & \text{otherwise} \end{cases} \quad \begin{array}{l} \text{if } 1 \leq i \leq n,\ 1 \leq j \leq m \\ \text{and } t \in \{0,1\} \end{array}$$

$$E_j(t) \leftrightharpoons E^1_j(t) \otimes (E^2_j(t) \otimes (\ldots (E^{n-1}_j(t) \otimes E^n_j(t)))) \quad \text{if } 1 \leq j \leq m \text{ and } t \in \{0,1\}$$

$$H_j \leftrightharpoons p_1 \otimes (p_2 \otimes (\ldots (p_{n-1} \otimes p_n))) \quad \text{if } 1 \leq j \leq m$$

$$F_j \leftrightharpoons E_j(1) \overset{H_j}{\sqcap} E_j(0) \quad \text{if } 1 \leq j \leq m$$

$$G_0 \leftrightharpoons H_1 \otimes (H_2 \otimes (\ldots (H_{m-1} \otimes H_m)))$$

$$G_i \leftrightharpoons G_{i-1} \oslash (p_i \oslash p_i) \quad \text{if } 1 \leq i \leq n$$

Let $\bar{\varphi} = F_1 \otimes (F_2 \otimes (\ldots (F_{m-1} \otimes F_m))) \rightarrow G_n$ be the **LG** sequent corresponding to the Boolean formula φ. We now claim that the $\vDash \varphi$ *if and only if* $\vdash_{LG} \bar{\varphi}$.

4.1 Example

Let us take the Boolean formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$ as an example. We have the primitive types $\{p_1, p_2\}$ and the types as shown in Figure 3. The formula is satisfiable, thus $\vdash_{LG} F_1 \otimes F_2 \rightarrow G_2$. A sketch of the derivation is given in Figure 3, some parts are proved in lemma's later on.

4.2 Intuition

Let us give some intuitions for the different parts of the construction, and a brief idea of why this would work. The basic idea is that on the lefthand-side we create a type for each literal (F_j is the formula for literal j), which will in the end result in the base type H_j, so $F_1 \otimes (F_2 \otimes (\ldots (F_{m-1} \otimes F_m)))$ will result in G_0. However, on the righthand-side we have an occurence of the expression $\oslash(p_i \oslash p_i)$ for each clause i, so in order to come to a derivation, we need to apply the $\oslash R$ rule for every clause i.

Each literal on the lefthand-side will result in either $E_j(1)$ (x_j is *true*) or $E_j(0)$ (x_j is *false*). This choice is created using a *join type* H_j such that $\vdash_{LG} E_j(1) \rightarrow H_j$ and $\vdash_{LG} E_j(0) \rightarrow H_j$, which we use to construct the *meet type* F_j. It can be shown that in this case $\vdash_{LG} F_j \rightarrow E_j(1)$ and $\vdash_{LG} F_j \rightarrow E_j(0)$, i.e. in the original formula we can replace F_j by either $E_j(1)$ or $E_j(0)$, giving us a choice for the truthvalue of x_j.

Let us assume that we need $x_1 = \mathit{true}$ to satisfy the formula, so on the lefthand-side we need to replace F_j by $E_1(1)$. $E_1(1)$ will be the product of exactly n parts, one for each clause $(E_1^1(1) \ldots E_1^n(1))$. Here $E_1^i(1)$ is $p_i \oslash (p_i \oslash p_i)$ *iff* x_1 does appear in clause i, and p_i otherwise. The first thing that should be noticed that $\vdash_{LG} p_i \oslash (p_i \oslash p_i) \rightarrow p_i$, so we can rewrite all $p_i \oslash (p_i \oslash p_i)$ into p_i so that $\vdash_{LG} E_1(1) \rightarrow H_1$.

However, we can also use the type $p_i \oslash (p_i \oslash p_i)$ to facilitate the application of the $\oslash R$ rule on the occurrence of the expression $\oslash(p_i \oslash p_i)$ in the righthand-side. From Lemma 5 we know that $\vdash_{LG} X^{\otimes}[p_i \oslash (p_i \oslash p_i)] \rightarrow G_i$ if $\vdash_{LG} X^{\otimes}[p_i] \cdot \oslash \cdot (p_i \oslash p_i) \rightarrow G_i$, so if the expression $\oslash Y$ occurs somewhere in a $\otimes$-structure we can move it to the outside. Hence, from the occurrence of $p_i \oslash (p_i \oslash p_i)$ on the lefthand-side we can move $\oslash(p_i \oslash p_i)$ to the outside of the $\otimes$-structure and p_i will be left behind within the original structure (just like if we rewrote it to p_i). However, the sequent is now of the form $X^{\otimes}[p_i] \cdot \oslash \cdot (p_i \oslash p_i) \rightarrow G_{i-1} \oslash (p_i \oslash p_i)$, so after applying the $\oslash R$ rule we have $X^{\otimes}[p_i] \rightarrow G_{i-1}$.

Now if the original CNF formula is satisfiable, we can use the meet types on the lefthand-side to derive the correct value of $E_j(1)$ or $E_j(0)$ for all j. If this assignment indeed satisfies the formula, then for each i the formula $p_i \oslash (p_i \oslash p_i)$ will appear at least once. Hence, for all occurrences of the expression $\oslash(p_i \oslash p_i)$ on the righthand-side we can apply the $\oslash R$ rule, after which the rest of the $p_i \oslash (p_i \oslash p_i)$ can be rewritten to p_i in order to derive the base type.

If the formula is not satisfiable, then there will be no way to have the $p_i \oslash (p_i \oslash p_i)$ types on the lefthand-side for *all* i, so there will be at least one occurence of $\oslash(p_i \oslash p_i)$ on the righthand-side where we cannot apply the $\oslash R$ rule. Because

$$
\begin{array}{l}
E_1(0) = p_1 \otimes (p_2 \oslash (p_2 \oslash p_2)) \\
E_1(1) = (p_1 \oslash (p_1 \oslash p_1)) \otimes p_2 \\
E_2(0) = (p_1 \oslash (p_1 \oslash p_1)) \otimes (p_2 \oslash (p_2 \oslash p_2)) \\
E_2(1) = p_1 \otimes p_2 \\
H_1 = p_1 \otimes p_2 \\
H_2 = p_1 \otimes p_2 \\
F_1 = E_1(1) \overset{H_1}{\sqcap} E_1(0) \\
F_2 = E_2(1) \overset{H_2}{\sqcap} E_2(0) \\
G_2 = ((H_1 \otimes H_2) \oslash (p_1 \oslash p_1)) \oslash (p_2 \oslash p_2)
\end{array}
$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{p_1 \to p_1 \quad p_1 \to p_1}{p_1 \oslash p_1 \to p_1 \cdot \oslash \cdot p_1}\ \oslash L
}{p_1 \cdot \oslash \cdot (p_1 \oslash p_1) \to p_1}\ dr
}{p_1 \oslash (p_1 \oslash p_1) \to p_1}\ \oslash L
\qquad p_2 \to p_2
}{(p_1 \oslash (p_1 \oslash p_1)) \cdot \otimes \cdot p_2 \to p_1 \otimes p_2}\ \otimes R
}{(p_1 \oslash (p_1 \oslash p_1)) \otimes p_2 \to p_1 \otimes p_2}\ \otimes L
}{E_1(1) \to H_1}\ Def
\qquad
\dfrac{
\dfrac{p_1 \to p_1 \quad p_2 \to p_2}{p_1 \cdot \otimes \cdot p_2 \to p_1 \otimes p_2}\ \otimes R
}{p_1 \cdot \otimes \cdot p_2 \to H_2}\ Def
}{F_1 \to H_1}\ 11
}{F_1 \cdot \otimes \cdot (p_1 \cdot \otimes \cdot p_2) \to H_1 \otimes H_2}\ \otimes R
\qquad p_1 \oslash p_1 \to p_1 \oslash p_1
}{(F_1 \cdot \otimes \cdot (p_1 \cdot \otimes \cdot p_2)) \cdot \oslash \cdot (p_1 \oslash p_1) \to (H_1 \otimes H_2) \oslash (p_1 \oslash p_1)}\ \oslash R
}{(F_1 \cdot \otimes \cdot (p_1 \cdot \otimes \cdot p_2)) \cdot \oslash \cdot (p_1 \oslash p_1) \to G_1}\ Def
}{F_1 \cdot \otimes \cdot ((p_1 \oslash (p_1 \oslash p_1)) \cdot \otimes \cdot p_2) \to G_1}\ Move
\qquad p_2 \oslash p_2 \to p_2 \oslash p_2
}{(F_1 \cdot \otimes \cdot ((p_1 \oslash (p_1 \oslash p_1)) \cdot \otimes \cdot p_2)) \cdot \oslash \cdot (p_2 \oslash p_2) \to G_1 \oslash (p_2 \oslash p_2)}\ \oslash R
}{(F_1 \cdot \otimes \cdot ((p_1 \oslash (p_1 \oslash p_1)) \cdot \otimes \cdot p_2)) \cdot \oslash \cdot (p_2 \oslash p_2) \to G_2}\ Def
}{F_1 \cdot \otimes \cdot ((p_1 \oslash (p_1 \oslash p_1)) \cdot \otimes \cdot (p_2 \oslash (p_2 \oslash p_2))) \to G_2}\ Move
}{(p_1 \oslash (p_1 \oslash p_1)) \cdot \otimes \cdot (p_2 \oslash (p_2 \oslash p_2)) \to F_1 \cdot \backslash \cdot G_2}\ r
}{(p_1 \oslash (p_1 \oslash p_1)) \otimes (p_2 \oslash (p_2 \oslash p_2)) \to F_1 \cdot \backslash \cdot G_2}\ \otimes L
}{F_1 \cdot \otimes \cdot ((p_1 \oslash (p_1 \oslash p_1)) \otimes (p_2 \oslash (p_2 \oslash p_2))) \to G_2}\ r
}{F_1 \cdot \otimes \cdot E_2(0) \to G_2}\ Def
\qquad
\dfrac{}{F_2 \to E_2(0)}\ 11
}{F_1 \cdot \otimes \cdot F_2 \to G_2}\ Repl
}{F_1 \otimes F_2 \to G_2}\ \otimes L
$$

Fig. 3: Sketch proof for LG sequent corresponding to $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$

the $\oslash$ will be the main connective we cannot apply any other rule, and we will never come to a valid derivation.

Note that the meet type F_j provides an *explicit* switch, so we first have to replace it by *either* $E_j(1)$ *or* $E_j(0)$ before we can do anything else with it. This guarantees is that if $\vdash_{LG} \bar{\varphi}$, there also must be some assignment $\langle t_1, \ldots, t_m \rangle \in \{0,1\}^m$ such that $\vdash_{LG} E_1(t_1) \otimes (E_2(t_2) \otimes (\ldots (E_{m-1}(t_{m-1}) \otimes E_m(t_m)))) \to G_n$, which means that $\langle t_1, \ldots, t_m \rangle$ is a satisfying assigment for φ.

5 Proof

We will now prove the main claim that $\vDash \varphi$ *if and only if* $\vdash_{LG} \bar{\varphi}$. First we will prove that *if* $\vDash \varphi$, *then* $\vdash_{LG} \bar{\varphi}$.

5.1 If-part

Let us assume that $\vDash \varphi$, so there is an assignment $\langle t_1, \ldots, t_m \rangle \in \{0,1\}^m$ that satisfies φ.

Lemma 9. *If* $1 \leq i \leq n$, $1 \leq j \leq m$ *and* $t \in \{0,1\}$ *then* $\vdash_{LG} E^i_j(t) \to p_i$.

Proof. We consider two cases:

1. If $E^i_j(t) = p_i$ this is simply the axiom rule.
2. If $E^i_j(t) = p_i \oslash (p_i \osl p_i)$ we can prove it as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{p_i \to p_i \quad p_i \to p_i}{p_i \cdot \osl \cdot p_i \to p_i \osl p_i} \osl R}{p_i \to p_i \cdot \oplus \cdot (p_i \osl p_i)} dr}{p_i \cdot \oslash \cdot (p_i \osl p_i) \to p_i} dr}{p_i \oslash (p_i \osl p_i) \to p_i} \oslash L$$

□

Lemma 10. *If* $1 \leq j \leq m$ *and* $t \in \{0,1\}$, *then* $\vdash_{LG} E_j(t) \to H_j$.

Proof. By applying the $\otimes L$ rule $n-1$ times together with the r rules we can turn $E_j(t)$ into a $\otimes$-structure. From Lemma 9 we know that $\vdash_{LG} E^i_j(t) \to p_i$, so using Lemma 4 we can replace all $E^i_j(t)$ by p_i in $E_j(t)$ after which we can apply the $\otimes R$ rule $n-1$ times to prove the lemma. □

Lemma 11. *If* $1 \leq j \leq m$, *then* $\vdash_{LG} F_j \to E_j(t_j)$

Proof. From Lemma 10 we know that $\vdash_{LG} E_j(1) \to H_j$ and $\vdash_{LG} E_j(0) \to H_j$, so $E_j(1) \sim E_j(0)$ with join-type H_j. Now from Lemma 7 we know that $\vdash_{LG} E_j(1) \overset{H_j}{\sqcap} E_j(0) \to E_j(1)$ and $\vdash_{LG} E_j(1) \overset{H_j}{\sqcap} E_j(0) \to E_j(0)$. □

Lemma 12. *We can replace each F_j in $\bar{\varphi}$ by $E_j(t_j)$, so:*

$$\frac{E_1(t_1) \cdot \otimes \cdot (E_2(t_2) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot E_m(t_m)))) \to G_n}{F_1 \otimes (F_2 \otimes (\ldots (F_{m-1} \otimes F_m))) \to G_n}$$

Proof. This can be proven by applying the $\otimes L$ rule $m-1$ times (together with the r rules) to turn it into a $\otimes$-structure, and then apply Lemma 11 in combination with Lemma 4 m times. □

Lemma 13. *In $E_1(t_1) \cdot \otimes \cdot (E_2(t_2) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot E_m(t_m)))) \to G_n$, there is at least one occurrence of $p_i \oslash (p_i \oset p_i)$ in the lefthand-side for every $1 \leq i \leq n$.*

Proof. This sequence of $E_1(t_1), \ldots, E_m(t_m)$ represent the truthvalue of all variables, and because this is a satisfying assignment, for all i there is at least one index k such that $\neg_{t_k} x_k$ appears in clause i. By definition we have that $E^i_k(t_k) = p_i \oslash (p_i \oset p_i)$. □

Definition 3. *$Y^i_j \leftrightharpoons E_j(t_j)$ with every occurrence of $p_k \oslash (p_k \oset p_k)$ replaced by p_k for all $i < k \leq n$*

Lemma 14. $\vdash_{LG} Y^0_1 \cdot \otimes \cdot (Y^0_2 \cdot \otimes \cdot (\ldots (Y^0_{m-1} \cdot \otimes \cdot Y^0_m))) \to G_0$

Proof. Because $Y^0_j = H_j$ by definition for all $1 \leq j \leq m$ and $G_0 = H_1 \otimes (H_2 \otimes (\ldots (H_{m-1} \otimes H_m)))$, this can be proven by applying the $\otimes R$ rule $m-1$ times. □

Lemma 15. *If $\vdash_{LG} Y^{i-1}_1 \cdot \otimes \cdot (Y^{i-1}_2 \cdot \otimes \cdot (\ldots (Y^{i-1}_{m-1} \cdot \otimes \cdot Y^{i-1}_m))) \to G_{i-1}$, then $\vdash_{LG} Y^i_1 \cdot \otimes \cdot (Y^i_2 \cdot \otimes \cdot (\ldots (Y^i_{m-1} \cdot \otimes \cdot Y^i_m))) \to G_i$*

Proof. From Lemma 13 we know that $p_i \oslash (p_i \oset p_i)$ occurs in $Y^i_1 \cdot \otimes \cdot (Y^i_2 \cdot \otimes \cdot (\ldots (Y^i_{m-1} \cdot \otimes \cdot Y^i_m)))$ (because the Y^i_j parts are $E_j(t_j)$ but with $p_k \oslash (p_k \oset p_k)$ replaced by p_k only for $k > i$). Using Lemma 5 we can move the expression $\oslash(p_i \oset p_i)$ to the outside of the lefthand-side of the sequent, after which we can apply the $\oslash R$-rule. After this we can replace all other occurrences of $p_i \oslash (p_i \oset p_i)$ by p_i using Lemma 9 and Lemma 4. This process can be summarized as:

$$\dfrac{\dfrac{\dfrac{\dfrac{Y^{i-1}_1 \cdot \otimes \cdot (Y^{i-1}_2 \cdot \otimes \cdot (\ldots (Y^{i-1}_{m-1} \cdot \otimes \cdot Y^{i-1}_m))) \to G_{i-1} \quad p_i \oset p_i \to p_i \oset p_i}{(Y^{i-1}_1 \cdot \otimes \cdot (Y^{i-1}_2 \cdot \otimes \cdot (\ldots (Y^{i-1}_{m-1} \cdot \otimes \cdot Y^{i-1}_m)))) \cdot \oslash \cdot (p_i \oset p_i) \to G_{i-1} \oslash (p_i \oset p_i)}\ \oslash R}{Y^{i-1}_1 \cdot \otimes \cdot (Y^{i-1}_2 \cdot \otimes \cdot (\ldots (Y^{i-1}_{m-1} \cdot \otimes \cdot Y^{i-1}_m))) \cdot \oslash \cdot (p_i \oset p_i) \to G_i}\ Def}{Y^{i-1}_1 \cdot \otimes \cdot (Y^{i-1}_2 \cdot \otimes \cdot (\ldots (Y^{i-1}_{m-1} \cdot \otimes \cdot Y^{i-1}_m))) \to G_i}\ 13, 5}{Y^i_1 \cdot \otimes \cdot (Y^i_2 \cdot \otimes \cdot (\ldots (Y^i_{m-1} \cdot \otimes \cdot Y^i_m))) \to G_i}\ 9, 4$$

□

Lemma 16. $\vdash_{LG} Y^n_1 \cdot \otimes \cdot (Y^n_2 \cdot \otimes \cdot (\ldots (Y^n_{m-1} \cdot \otimes \cdot Y^n_m))) \to G_n$

Proof. We can prove this using induction with Lemma 14 as base and Lemma 15 as induction step. □

Lemma 17. *If* $\vDash \varphi$*, then* $\vdash_{LG} \bar{\varphi}$*,*

Proof. From Lemma 16 we know that $\vdash_{LG} Y_1^n \cdot\otimes\cdot (Y_2^n \cdot\otimes\cdot (\ldots (Y_{m-1}^n \cdot\otimes\cdot Y_m^n))) \rightarrow G_n$, and because by definition $Y_j^n = E_j(t_j)$, we also have that $\vdash_{LG} E_1(t_1) \cdot\otimes\cdot (E_2(t_2) \cdot\otimes\cdot (\ldots (E_{m-1}(t_{m-1}) \cdot\otimes\cdot E_m(t_m)))) \rightarrow G_n$. Finally combining this with Lemma 12 we have that $\vdash_{LG} \bar{\varphi} = F_1 \otimes (F_2 \otimes (\ldots (F_{m-1} \otimes F_m))) \rightarrow G_n$, using the assumption that $\vDash \varphi$. □

5.2 Only-if part

For the only-if part we will need to prove that *if* $\vdash_{LG} \bar{\varphi}$*, then* $\vDash \varphi$. Let us now assume that $\vdash_{LG} \bar{\varphi}$.

Lemma 18. *If* $\vdash_{LG} X \rightarrow P \oslash Y$*, then there exist a* Q *such that* Q *is part of* X *(possibly inside a formula in* X*) and* $\vdash_{LG} Y \rightarrow Q$*.*

Proof. The only rule that matches a $\oslash$ in the righthand-side is the $\oslash R$ rule. Because this rule needs a $\cdot \oslash \cdot$ connective in the lefthand-side, we know that if $\vdash_{LG} X \rightarrow P \oslash Y$ it must be the case that we can turn X into $X' \cdot \oslash \cdot Q$ such that $\vdash_{LG} X' \rightarrow P$ and $\vdash_{LG} Y \rightarrow Q$. □

Lemma 19. *If* $\vdash_{LG} E_1(t_1) \cdot\otimes\cdot (E_2(t_2) \cdot\otimes\cdot (\ldots (E_{m-1}(t_{m-1}) \cdot\otimes\cdot E_m(t_m))) \rightarrow G_n$*, then there is an occurrence* $p_i \oslash (p_i \oslash p_i)$ *on the lefthand-side at least once for all* $1 \leq i \leq n$*.*

Proof. G_n by definition contains an occurrence of the expression $\oslash(p_i \oslash p_i)$ for all $1 \leq i \leq n$. From Lemma 18 we know that somewhere in the lefthand-side we need an occurrence of a structure Q such that $\vdash_{LG} p_i \oslash p_i \rightarrow Q$. From the construction it is obvious that the only possible type for Q is in this case $p_i \oslash p_i$, and it came from the occurrence of $p_i \oslash (p_i \oslash p_i)$ on the lefthand-side. □

Lemma 20. *If* $\vdash_{LG} E_1(t_1) \cdot\otimes\cdot (E_2(t_2) \cdot\otimes\cdot (\ldots (E_{m-1}(t_{m-1}) \cdot\otimes\cdot E_m(t_m))) \rightarrow G_n$*, then* $\langle t_1, t_2, \ldots, t_{m-1}, t_m\rangle$ *is a satisfying assignment for the CNF formula.*

Proof. From Lemma 19 we know that there is a $p_i \oslash (p_i \oslash p_i)$ in the lefthand-side of the formula for all $1 \leq i \leq n$. From the definition we know that for each i there is an index j such that $E_j^i(t_j) = p_i \oslash (p_i \oslash p_i)$, and this means that $\neg_{t_j} x_j$ appears in clause i, so all clauses are satisfied. Hence, this choice of $t_1 \ldots t_m$ is a satisfying assignment. □

Lemma 21. *If* $1 \leq j \leq m$ *and* $\vdash_{LG} X^{\otimes}[F_j] \rightarrow G_n$*, then* $\vdash_{LG} X^{\otimes}[E_j(0)] \rightarrow G_n$ *or* $\vdash_{LG} X^{\otimes}[E_j(1)] \rightarrow G_n$*.*

Proof. We know that $X^{\otimes}[F_j]$ is a $\otimes$-structure, so we can apply the r rule several times to move all but the F_j-part to the righthand-side. We then have that $\vdash_{LG} F_j \rightarrow \ldots \cdot \backslash \cdot G_n \cdot / \cdot \ldots$. From Lemma 8 we know that now have that $\vdash_{LG} E_j(0) \rightarrow \ldots \cdot \backslash \cdot G_n \cdot / \cdot \ldots$ or $\vdash_{LG} E_j(1) \rightarrow \ldots \cdot \backslash \cdot G_n \cdot / \cdot \ldots$. Finally we can apply the r rule again to move all parts back to the lefthand-side, to show that $\vdash_{LG} X^{\otimes}[E_j(0)] \rightarrow G_n$ or $\vdash_{LG} X^{\otimes}[E_j(1)] \rightarrow G_n$.

Note that, in order for Lemma 8 to apply, we have to show that this sequent satisfies the constraints. G_n does contain $A_1 \otimes A_2$ with output polarity, however the only connectives in A_1 and A_2 are $\otimes$. Because no rules apply on $A/((C/C)\backslash C) \to A_1' \otimes A_1''$, we have that $\not\vdash_{LG} A/((C/C)\backslash C) \to A_1$. In $X^{\otimes}[]$, the only $\otimes$ connectives are within other F_k, however these have an input polarity and do not break the constraints either.

So, in all cases F_j provides an *explicit switch*, which means that the truthvalue of a variable can only be changed in all clauses simultanously. □

Lemma 22. *If* $\vdash_{LG} \bar{\varphi}$*, then* $\vDash \varphi$*.*

Proof. From Lemma 21 we know that all derivations will first need to replace each F_j by *either* $E_j(1)$ *or* $E_j(0)$. This means that if $\vdash_{LG} F_1 \otimes (F_2 \otimes (\ldots (F_{m-1} \otimes F_m))) \to G_n$, then also $\vdash_{LG} E_1(t_1) \cdot \otimes \cdot (E_2(t_2) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot E_m(t_m))) \to G_n$ for some $\langle t_1, t_2, \ldots, t_{m-1}, t_m \rangle \in \{0,1\}^m$. From Lemma 20 we know that this is a satisfying assignment for φ, so if we assume that $\vdash_{LG} \bar{\varphi}$, then $\vDash \varphi$. □

5.3 Conclusion

Theorem 1. ***LG*** *is NP-complete.*

Proof. From Lemma 3 we know that for every derivable sequent there exists a derivation that is of polynomial length, so the derivability problem for **LG** is in NP. From Lemma 17 and Lemma 22 we can conclude that we can reduce SAT to **LG**. Because SAT is a known NP-hard problem (Garey and Johnson, 1979), and our reduction is polynomial, we can conclude that derivability for **LG** is also NP-hard.

Combining these 2 facts we conclude that the derivability problem for **LG** is NP-complete. □

6 The product-free case

In the *product-free* fragment of **LG**, the $\otimes L$, $\otimes R$, $\oplus L$ and $\oplus R$ rules are omitted, so formulas cannot contain the $\otimes$ or $\oplus$ connectives. Note that the structural $\cdot \otimes \cdot$ and $\cdot \oplus \cdot$ can still occur in derivations, for example after application of a r or dr rule.

For the product-free fragment of **L** it has been proven by Savateev (2009) that the derivability problem is also NP-complete. This is a remarkable result that does not follow directly from the fact that the derivability problem for **L** itself is NP-complete. It is known that removing the product restricts the calculus in an essential way, for example the diamond property does not hold in its original form in the product-free fragment (Pentus, 1993a). As the diamond property was used in the original proof by Pentus (2003), it could have been the case that the reduction from SAT was not possible in the product-free fragment.

However, as Savateev proved this is not the case, and the product-free fragment pf **L** is also NP-complete.

In this section we will show that we can also reduce a Boolean formula in conjunctive normal form to a *product-free* sequent of the Lambek-Grishin calculus. Let again $\varphi = c_1 \wedge \ldots \wedge c_n$ be a Boolean formula in conjunctive normal form with clauses $c_1 \ldots c_n$ and variables $x_1 \ldots x_m$ and for all $1 \leq j \leq m$ let $\neg_0 x_j$ stand for the literal $\neg x_j$ and $\neg_1 x_j$ stand for the literal x_j.

Let p_i (for $1 \leq i \leq n$) be distinct primitive types from Var. We define the following families of types:

$$E^i_j(t) \leftrightharpoons \begin{cases} p_i/(p_i \backslash p_i) & \text{if } \neg_t x_j \text{ appears in clause } c_i \\ p_i & \text{otherwise} \end{cases} \qquad \text{if } 1 \leq i \leq n,\ 1 \leq j \leq m \text{ and } t \in \{0,1\}$$

$$E_j(t) \leftrightharpoons (((((s/s) \oslash E^1_j(t)) \oslash E^2_j(t)) \oslash \ldots) \oslash E^{n-1}_j(t)) \oslash E^n_j(t) \qquad \text{if } 1 \leq j \leq m \text{ and } t \in \{0,1\}$$

$$H \leftrightharpoons (((((s/s) \oslash p_1) \oslash p_2) \oslash \ldots) \oslash p_{n-1}) \oslash p_n$$

$$F^0_j \leftrightharpoons E_j(0)/((H/H)\backslash H) \qquad \text{if } 1 \leq j \leq m$$

$$F^1_j \leftrightharpoons (H/H)\backslash E_j(1) \qquad \text{if } 1 \leq j \leq m$$

$$G^0_m \leftrightharpoons s$$

$$G^i_0 \leftrightharpoons G^{i-1}_m \qquad \text{if } 1 \leq i \leq n$$

$$G^i_j \leftrightharpoons G^i_{j-1} \oslash p_i \qquad \text{if } 1 \leq i \leq n \text{ and } 1 \leq j < m$$

$$G^i_m \leftrightharpoons G^i_{m-1} \oslash (p_i/(p_i \backslash p_i)) \qquad \text{if } 1 \leq i \leq n$$

Let $\bar{\varphi} = (F^0_1 \cdot \otimes \cdot F^1_1) \cdot \otimes \cdot (\ldots ((F^0_{m-1} \cdot \otimes \cdot F^1_{m-1}) \cdot \otimes \cdot ((F^0_m \cdot \otimes \cdot F^1_m) \cdot \otimes \cdot s))) \rightarrow G^n_m$ be the product-free **LG** sequent corresponding to the Boolean formula φ. We now claim that the $\vDash \varphi$ *if and only if* $\vdash_{LG} \bar{\varphi}$.

6.1 Intuition

For each variable we use again a kind of meet-type to create a choice for its truthvalue. Remark that the diamond property does not hold in the product-free case in its original form, i.e. if we have $\vdash_{LG} A \rightarrow C$ and $\vdash_{LG} B \rightarrow C$, it is not always the case that there exists a product-free formula D such that $\vdash_{LG} D \rightarrow A$ and $\vdash_{LG} D \rightarrow B$. However, we can always construct the formulas $D_1 = A/((C/C)\backslash C)$ and $D_2 = (C/C)\backslash B$ such that $\vdash_{LG} D_1 \cdot \otimes \cdot D_2 \rightarrow A$ and $\vdash_{LG} D_1 \cdot \otimes \cdot D_2 \rightarrow B$. Note that this structure is the same as the structure in Lemma 8, so we know that this property holds.

In the construction we now have F^0_j and F^1_j as parts of the meet-type. We can replace these by either $E_j(1)$ or $E_j(0)$, so that we have $E_1(t_1) \cdot \otimes \cdot (E_2(t_2) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s)))) \rightarrow G^n_m$ for some $\langle t_1, t_2 \ldots t_{m-1}, t_m \rangle \in \{0,1\}^m$.

Let us call a formula of the form $((((A \oslash B_1) \oslash B_2) \oslash \ldots) \oslash B_{n-1}) \oslash B_n$ a $\oslash$-*stack* with n items, and let us call B_n the topmost item of this stack. A is considered to be the base of this $\oslash$-stack.

The sequent $E_1(t_1)\cdot\otimes\cdot(E_2(t_2)\cdot\otimes\cdot(\ldots(E_{m-1}(t_{m-1})\cdot\otimes\cdot(E_m(t_m)\cdot\otimes\cdot s))))\rightarrow G_m^n$ consists of m $\oslash$-stacks in the lefthand-side $(E_1(t_1)\ldots E_m(t_m))$, each containing one item for every clause. On the righthand-side there is also a $\oslash$-stack, with $n\cdot m$ items. Because of the logical $\oslash$ connective on the righthand-side, there has to be a matching item on the lefthand-side for every item from the righthand-side. By means of the Grishin interactions, every topmost item from a $\oslash$-stack in the lefthand-side can directly match with the item on the righthand-side, as the lefthand-side is an $\otimes$-structure.

Now the $\oslash$-stack on the righthand-side will be in order of decreasing clause number. For each clause it first contains the type $p_i/(p_i\backslash p_i)$, so in the lefthand-side there has to be at least one item $p_i/(p_i\backslash p_i)$ (meaning that there is a variable satisfying clause i). Then, there are $m-1$ types p_i to remove the rest of the p_i or $p_i/(p_i\backslash p_i)$ in the lefthand-side. Note that $\vdash_{LG} p_i\rightarrow p_i/(p_i\backslash p_i)$.

Finally, if all items from the stacks could be removed, for each clause there was a variable satisfying it, so we are left with $(s/s)\cdot\otimes\cdot((s/s)\cdot\otimes\cdot ...((s/s)\cdot\otimes\cdot s))\rightarrow s$ which is obviously derivable. If it was not satisfyable, there was at least 1 index i for which there was no $p_i/(p_i\backslash p_i)$ in the lefthand-side, so the sequent is not derivable.

6.2 If-part

We will now prove that *if* $\vDash\varphi$, *then* $\vdash_{LG}\bar{\varphi}$. We now assume that $\vDash\varphi$, so there is an assignment $\langle t_1,\ldots,t_m\rangle\in\{0,1\}^m$ that satisfies φ.

Lemma 23. *If* $1\leq i\leq n$, $1\leq j\leq m$ *and* $t\in\{0,1\}$ *then* $\vdash_{LG} p_i\rightarrow E_j^i(t)$.

Proof. We consider two cases:

1. If $E_j^i(t)=p_i$ this is simply the axiom rule.
2. If $E_j^i(t)=p_i/(p_i\backslash p_i)$ we can prove it as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{p_i\rightarrow p_i\quad p_i\rightarrow p_i}{p_i\backslash p_i\rightarrow p_i\cdot\backslash\cdot p_i}\,\backslash L}{p_i\cdot\otimes\cdot(p_i\backslash p_i)\rightarrow p_i}\,r}{p_i\rightarrow p_i\cdot/\cdot(p_i\backslash p_i)}\,r}{p_i\rightarrow p_i/(p_i\backslash p_i)}\,/R$$

□

Lemma 24. *If* $1\leq j\leq m$ *and* $t\in\{0,1\}$ *then* $\vdash_{LG} E_j(t)\rightarrow H$.

Proof. We can apply the $\oslash R$ rule n times together with Lemma 23 to prove this. □

Lemma 25. *If* $1\leq j\leq m$ *then* $\vdash_{LG} F_j^0\cdot\otimes\cdot F_j^1\rightarrow E_j(t_j)$.

Proof. $F_j^0\cdot\otimes\cdot F_j^1$ is the meet-type from Lemma 7 with a structural $\cdot\otimes\cdot$ in place of the original $\otimes$. From Lemma 24 we know that H_j is the join type of $E_j(0)$ and $E_j(1)$, so $\vdash_{LG} F_j^0\otimes F_j^1\rightarrow E_j(t_j)$. Finally from Lemma 6 we know that in this case also $\vdash_{LG} F_j^0\cdot\otimes\cdot F_j^1\rightarrow E_j(t_j)$. □

Lemma 26. *We can replace each* $F_j^0 \cdot \otimes \cdot F_j^1$ *in* $\bar{\varphi}$ *by* $E_j(t_j)$*, so:*

$$\frac{E_1(t_1) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s))) \to G_m^n}{(F_1^0 \cdot \otimes \cdot F_1^1) \cdot \otimes \cdot (\ldots ((F_{m-1}^0 \cdot \otimes \cdot F_{m-1}^1) \cdot \otimes \cdot ((F_m^0 \cdot \otimes \cdot F_m^1) \cdot \otimes \cdot s))) \to G_m^n}$$

Proof. This can be proven by applying the $\otimes L$ rule m times (together with the r rules) to turn it into a $\otimes$-structure, and then apply Lemma 25 in combination with Lemma 4 m times. □

Lemma 27. *In* $E_1(t_1) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s))) \to G_m^n$*, there is at least one occurrence of the expression* $\oslash(p_i/(p_i \backslash p_i))$ *in the lefthand-side for every* $1 \leq i \leq n$*.*

Proof. This sequence of $E_1(t_1), \ldots, E_m(t_m)$ represent the truthvalue of all variables, and because this is a satisfying assignment, for all i there is at least one index k such that $\neg_{t_k} x_k$ appears in clause i. By definition we have that $E_k^i(t_k) = p_i/(p_i \backslash p_i)$. □

Lemma 28. $\vdash_{LG} E_1(t_1) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s))) \to G_m^n$.

Proof. We prove this by induction on the length of G_m^n. By definition we have that $G_m^n = G_{m-1}^n \oslash (p_n/(p_n \backslash p_n))$, and from Lemma 27 we know that the expression $\oslash(p_n/(p_n \backslash p_n))$ occurs on the lefthand-side as outermost part of some $E_k(t_k)$, so by using Lemma 5 we can move this expression to the outside of the lefthand-side after which we can apply the $\oslash R$ rule.

Now on the righthand-side we have G_{m-1}^n, which consists of G_m^{n-1} surrounded by $m-1$ occurrences of the expression $\oslash p_n$. In the lefthand-side there are $m-1$ occurrences of $\oslash E_j^n(t_j)$, for every $1 \leq j \leq m$ $(j \neq k)$. Using the fact from Lemma 23 that $\vdash_{LG} E_j^n(t_j) \to p_n$, we can again use Lemma 5 and the $\oslash R$ rule to remove all these expressions from the left- and righthand-side.

The sequent that remains is of exactly the same form, but for $n-1$ instead of n clauses. The same reasoning applies on this sequent, so we can repeat this process n times. Then, the $\oslash R$ rule has been applied $n \cdot m$ times in total, and the sequent will be of the form $(s/s) \cdot \otimes \cdot ((s/s) \cdot \otimes \cdot \ldots ((s/s) \cdot \otimes \cdot s))) \to s$. This can easily be derived, so it must be the case that also $\vdash_{LG} E_1(t_1) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s))) \to G_m^n$. □

Lemma 29. *If* $\vDash \varphi$*, then* $\vdash_{LG} \bar{\varphi}$*,*

Proof. From Lemma 28 we know that $\vdash_{LG} E_1(t_1) \cdot \otimes \cdot (\ldots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s))) \to G_m^n$, and using Lemma 26 we know that in this case also $\vdash_{LG} (F_1^0 \cdot \otimes \cdot F_1^1) \cdot \otimes \cdot (\ldots ((F_{m-1}^0 \cdot \otimes \cdot F_{m-1}^1) \cdot \otimes \cdot ((F_m^0 \cdot \otimes \cdot F_m^1) \cdot \otimes \cdot s))) \to G_m^n$. □

6.3 Only-if part

For the only-if part we will need to prove that *if* $\vdash_{LG} \bar{\varphi}$*, then* $\vDash \varphi$. Let us now assume that $\vdash_{LG} \bar{\varphi}$.

Lemma 30. *If* $\vdash_{LG} E_1(t_1) \cdot \otimes \cdot (\dots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s))) \to G_m^n$, *then there is an occurrence of* $p_i/(p_i \backslash p_i)$ *on the lefthand-side at least once for all* $1 \leq i \leq n$.

Proof. G_m^n by definition contains an occurrence of the expression $\oslash(p_i/(p_i\backslash p_i))$ for all $1 \leq i \leq n$. From Lemma 18 we know that somewhere in the lefthand-side we need an occurrence of a structure Q such that $\vdash_{LG} p_i/(p_i\backslash p_i) \to Q$. From the construction it is obvious that the only possible type for Q is in this case $p_i/(p_i\backslash p_i)$. □

Lemma 31. *If* $\vdash_{LG} E_1(t_1) \cdot \otimes \cdot (\dots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s))) \to G_m^n$, *then* $\langle t_1, \dots, t_{m-1}, t_m \rangle$ *is a satisfying assignment for the CNF formula.*

Proof. From Lemma 30 we know that there is an occurrence of $p_i/(p_i\backslash p_i)$ in the lefthand-side of the formula for all $1 \leq i \leq n$. From the definition we know that for each i there is an index j such that $E_j^i(t_j) = p_i/(p_i\backslash p_i)$, and this means that $\neg_{t_j} x_j$ appears in clause i, so if for every $1 \leq i \leq n$ there is an occurrence of $p_i/(p_i\backslash p_i)$ in the lefthand-side then all clauses are satisfied. Hence, this choice of $t_1 \dots t_m$ is a satisfying assignment. □

Lemma 32. *If* $1 \leq j \leq m$ *and* $\vdash_{LG} X^{\otimes}[F_j^0 \cdot \otimes \cdot F_j^1] \to G_m^n$, *then* $\vdash_{LG} X^{\otimes}[E_j(0)] \to G_m^n$ *or* $\vdash_{LG} X^{\otimes}[E_j(1)] \to G_m^n$.

Proof. We know that $X^{\otimes}[F_j^0 \cdot \otimes \cdot F_j^1]$ is a $\otimes$-structure, so we can apply the r rule several times to move all but the structure $F_j^0 \cdot \otimes \cdot F_j^1$ to the righthand-side. We then have that $\vdash_{LG} F_j^0 \cdot \otimes \cdot F_j^1 \to \dots \cdot \backslash \cdot G_m^n \cdot / \cdot \dots$. As remarked in Lemma 25 this is exactly the meet-type from Lemma 8 with a structural $\cdot \otimes \cdot$, so we know that now have that $\vdash_{LG} E_j(0) \to \dots \cdot \backslash \cdot G_m^n \cdot / \cdot \dots$ or $\vdash_{LG} E_j(1) \to \dots \cdot \backslash \cdot G_m^n \cdot / \cdot \dots$. Finally we can apply the r rule again to move all parts back to the lefthand-side, to show that $\vdash_{LG} X^{\otimes}[E_j(0)] \to G_m^n$ or $\vdash_{LG} X^{\otimes}[E_j(1)] \to G_m^n$.

Note that, in order for Lemma 8 to apply we need to show that this sequent satisfies the constrains. However, as we are now using the product-free **LG** the $\otimes$ connective will never appear so the constraints are always satisfied. So, in all cases $F_j^0 \cdot \otimes \cdot F_j^1$ provides an *explicit switch*, which means that the truthvalue of a variable can only be changed in all clauses simultanously. □

Lemma 33. *If* $\vdash_{LG} \bar{\varphi}$, *then* $\vDash \varphi$.

Proof. From Lemma 32 we know that all derivations will first need to replace each $F_j^0 \cdot \otimes \cdot F_j^1$ by *either* $E_j(1)$ *or* $E_j(0)$. This means that if $\vdash_{LG} (F_1^0 \cdot \otimes \cdot F_1^1) \cdot \otimes \cdot (\dots ((F_{m-1}^0 \cdot \otimes \cdot F_{m-1}^1) \cdot \otimes \cdot ((F_m^0 \cdot \otimes \cdot F_m^1) \cdot \otimes \cdot s))) \to G_m^n$, then also $\vdash_{LG} E_1(t_1) \cdot \otimes \cdot (\dots (E_{m-1}(t_{m-1}) \cdot \otimes \cdot (E_m(t_m) \cdot \otimes \cdot s))) \to G_m^n$ for some $\langle t_1, \dots, t_{m-1}, t_m \rangle \in \{0,1\}^m$. From Lemma 31 we know that this is a satisfying assignment for φ, so if we assume that $\vdash_{LG} \bar{\varphi}$, then $\vDash \varphi$. □

6.4 Conclusion

Theorem 2. *The product-free fragment of **LG** is NP-complete.*

Proof. From Lemma 3 we know that for every derivable sequent there exists a derivation that is of polynomial length, so the derivability problem for (the product-free fragment of) **LG** is in NP. From Lemma 29 and Lemma 33 we can conclude that we can reduce SAT to the product-free fragment of **LG**. Because SAT is a known NP-hard problem (Garey and Johnson, 1979), and our reduction is polynomial, we can conclude that derivability for the product-free fragment **LG** is also NP-hard.

Combining these 2 facts we conclude that the derivability problem for the product-free fragment of **LG**, like the derivability problem for **LG**, is NP-complete. □

7 Conclusion and future work

We have showed that both **LG** and the product-free fragment of **LG** are NP-complete. Unfortunately this is an undesired property, as we would like to have an effient method for distinguishing between sentences and non-sentences. However, this result does not automatically mean that the calculus is useless, as it might be the case that certain restrictions on the derivations could guarantee polynomial time derivability while maintaining enough expressivity for natural languages.

In the next sections we will describe some ideas for future work on the complexity, as wel as the expressivity.

7.1 Bound on nesting depth

In the constructions for proving NP-completeness we have artificially constructed very complicated formulas. For example for a Boolean formula with n clauses, the righthand side of the sequent will consist of a formula with a literal that is nested in n $\oslash$-connectives.

In working with natural languages these deeply-nested formulas will not occur. Although the input sentences could be very long, the individual types of the words will be predefined, and therefore bounded to only a small nesting depth. So, it could be acceptable to place a bound on the maximum nesting depth. In this way the expressivity will remain, but the complexity of the derivability problem could be polynomial. However, this has not been looked into yet and is still an open problem.

7.2 Addition and multiplication

It is possible to model *counting* in **LG**. Let us define the following grammar:

Word	Types
a	$(s/s) \oslash (s \otimes a)$, $s \oslash (s \otimes a)$
c	$(s/s) \oslash (a \otimes s)$, $s \oslash (a \otimes s)$

This grammar will generate the permutation closure of the context-free language $\{a^n c^n | n \in \mathbb{N}\}$, so all strings with an equal number of a's an c's.

If we add the word b to this lexicon and assign it the same types as a, this will generate the permutation closure of the language $\{a^n b^m c^k | n, m \in \mathbb{N}, k = m+n\}$, so the language of strings where the number of a's plus the number of b's is equal to the number of c's. This can be seen as a model for addition.

Now the question is: is it also possible to define a grammar that recognizes strings where the number of a's *multiplied by* the number of b's equals the number of c's? Intuitively I would answer this question with a no, but I have not proven this yet and it therefore remains an open question.

7.3 The pigeonhole principle

The pigeonhole principle (Cook and Reckhow, 1979) can be defined as follows: if there are n items that are put in m pigeonholes and $n > m$, there must be at least one pigeonhole with more than one item in it. This might seem trivial, but it turns out to be an interesting problem.

It is easy to define the problem in propositional logic, but finding a polynomial size proof for this in proposional formula is much harder. The trivial proof is of exponential length, but it turns out that by means of counting a polynomial proof can be given.

Another interesting question arises: is it possible to encode the pigeonhole principle in **LG**? And is it possible to find a polynomial size derivation for it? The first question is still an open problem, and thus cannot be answered, but we could answer the second question: If it is possible to encode the pigeonhole principle in **LG**, then there will exist a polynomial size derivation for it. It can be easily seen that if it can be encoded then the corresponding sequent will be derivable, and from Lemma 3 we know that if this sequent is derivable, there exists a polynomial size derivation.

Part II

Theorem prover

8 Overview

Researchers that are working with the different kinds of categorial grammars, are often interested in having automated theorem provers for these grammars. For the *structure preserving* categorial grammars like **AB grammars**, **NL** and **bi-NL** (the system **LG** without Grishin interaction principles) theorem proving algorithms have been given by Capelletti (2007). The work by Capelletti also gives an overview of related calculi.

For **L**, **NL** and **NL**$\Diamond$ a theorem prover called *Grail* has been developed by Moot (1999). This theorem prover cannot only find derivations for a given sequent (based on a lexicon), but also has an interactive interface to allow the user to choose between different derivations. Since the paper from 1999 it has been updated multiple times, and the current latest version is *Grail 3*[1].

For **LG** there has been an implementation by Moot (2008) of a theorem prover called *Hyperion*[2]. This theorem prover is based on proofnets for **LG** embedded in Hyperedge Replacement Grammars. It is known that **LG** sequents can also be represented as proofnets (Moot, 2007), and Moot (2008) claimed that these proofnets are bound to certain restrictions in such way that they could be embedded in Hyperedge Replacement Grammars, for which there exist polynomial algorithms. Unfortunately it turned out that there exist sequents in **LG** that do not satisfy these restrictions, and the Hyperion theorem prover can therefore find derivations only for a fragment of **LG**.

9 Implementation

Together with this thesis a theorem prover for **LG** has been written. The theoretical runtime complexity of this theorem prover is in `EXPTIME`, which is no surprise after the result of Theorem 1. Although this means that the derivability problem is hard, it does not mean it is impossible to solve it, and as we will show this theorem prover can find derivations for smaller sequents (for example the ones used in examples for natural languages) within seconds.

The theorem prover, written in `C++`, has been set up as a generic theorem prover for categorial grammars in display logic format. A generic structure has been chosen for encoding the inference rules, in such a way that other rules can easily be added in later stages. The prover supports both unary and binary connectives, and it will return a representation of the derivation which can easily be translated to for example a LaTeX representation (already build-in) or a term of the $\bar{\lambda}\mu\tilde{\mu}$ calculus (Curien and Herbelin, 2000; Bernardi and Moortgat, 2010).

We will now discuss the implementational details of this theorem prover.

9.1 Representation

In this section we will discuss the structure of the internal representation of formulas, structures, sequents, inference rules and derivations. For memory ef-

[1] http://www.labri.fr/perso/moot/grail3.html
[2] http://www.labri.fr/perso/moot/hyperion/

ficiency all of these have been implemented as a `struct` in `C++`, so there is no overhead of extra information about the objects like there would be in an object oriented approach. The downside of this is that there cannot be any automatic garbage collection, so we will have to free al memory that is used ourselves.

Let us first declare the unary and binary connectives. These are implemented as an `enum` and take only 1 byte of memory. Currently we have defined the following unary and binary connectives:

```
enum UNARY_CONNECTIVE { ZERO, ONE };
enum BINARY_CONNECTIVE { OTIMES, BACKSLASH, SLASH,
                         OPLUS, OBACKSLASH, OSLASH};
```

The unary connectives refer to the *(dual) Galois negations*, the binary connectives refer to the six binary connectives used in **LG**.

For formulas and structures we distinguish between three different types:

- *Primitive*: The smallest possible formula (structure), in case of a formula this is a literal, in case of a structure this is a formula.
- *Unary*: A formula (structure) that is preceded or followed by an unary connective; a boolean value in the representation of the formula (structure) will indicate whether this is a prefix or postfix unary connective.
- *Binary*: Two formulas (structures) that are combined by a binary connective.

Now the declaration of a formula is as follows:

```
struct Formula {
   FORMULA_STRUCTURE_TYPE type;
   union {
      /* Primitive */
      struct {
         char *name;
      };
      /* Unary */
      struct {
         bool prefix;
         UNARY_CONNECTIVE unary_connective;
         Formula *inner;
      };
      /* Binary */
      struct {
         Formula *left;
         BINARY_CONNECTIVE binary_connective;
         Formula *right;
      };
   };
};
```

The `union` keyword is used in `C++` to declare a list of items, of which only 1 can be set at a time. In our case this means that only one of the three inner `structs`

can be set, which is obviously the supposed behaviour. The `type` has to be set in all cases to indicate which of the three types this formula represents.

The representation for structures is very similar to those of formulas and the declaration will therefore not be given. The type of a primitive is in this case `Formula *`, and the types of `inner`, `left` and `right` are `Structure *`.

For sequents we will also consider three different types: *Formula left*, *formula right*, or *structural*. Remark that in the representation of **LG** given in Chapter 2 we only considered sequents of the *structural* type, and we considered a formula A and a structure X containing formula A as a primitive to be the same. However, to allow for distinction between active and passive formulas we will consider these as not equal, and we will add rules of inference to switch between those two.

Now the representation of a sequent is as follows:

```
struct Sequent {
   SEQUENT_TYPE type;
   union {
      Formula *leftF;
      Structure *leftS;
   };
   union {
      Formula *rightF;
      Structure *rightS;
   };
};
```

In this representation we can theoretically set both the left and the right side to a formula, however in our implementation this will never be the case.

Now that we can represent sequents we will explain how the inference rules are represented. An inference rule has a *name*, a *LaTeX name*, a *conclusion* and a (possibly empty) list of *premisses*. It is implemented as follows:

```
struct Rule {
   char *name;
   char *latexName;
   Sequent *conclusion;
   int numPremisses;
   Sequent **premisses;
};
```

Now for each inference rule there must be an instance of this structure. In order to represent the conclusion and premisses in such a way that there can be wildcard matches, a fourth type is added to the formula and structure representations. With this type, *match*, only a single character is stored and it is a representation for a wildcard match, used for unification. If the same character is used both in the premisse and in the conclusion, those formulas or structures will be unified.

Let us for example look at the following rule of inference:

$$\frac{A \cdot \oslash \cdot B \to P}{A \oslash B \to P} \oslash L$$

This will be represented as:

```
new Rule("(/) L", "\\oslash L",
   new Sequent(
      new Formula(new Formula('A'), OSLASH, new Formula('B')),
      new Structure('P')
   ),
   new Sequent(
      new Structure(new Structure(new Formula('A')), OSLASH,
         new Structure(new Formula('B'))),
      new Structure('P')
   ));
```

Careful readers might have noticed that we used certain constructors to instantiate the `structs`. In the explanation of the representation these have been omitted for clarity, but for convenient use of the representations these have been declared in the `structs`. The constructors are not only used for shorter and better-readable code, but also make sure that we only use the representations in the intended way, by means of one of the constructors.

The last representation is those of derivations. It is implemented as follows:

```
struct Derivation {
   Rule *rule;
   Sequent *conclusion;
   int numPremisses;
   Derivation **premisses;
};
```

9.2 Search procedure

In this section the search procedure for finding derivations for the given sequent will be explained. The global search strategy is a *top-down* strategy that starts from the target sequent (the top) and tries to apply the inference rules in order to end in axioms (the bottom). Note that in our representation of the derivations these are turned upside-down, as we write the target sequent at the bottom and the axioms at the top.

The base of the search procedure is *iterative deepening*. In iterative deepening the search space, organized as a DAG (Directed Acyclic Graph), is searched step by step with increasing depth until a derivation has been found. The depth is measured as the maximum number of derivation steps between one of the axioms and the target sequent. So, the procedure starts by trying depth 1, which will only succeed if the target sequent is an axiom. If it doesn't succeed it will

continue with depth 2, where it will try to apply any rule on the target sequent and continue on the premisses with depth 1. This procedure continues until a derivation has been found.

With this procedure, the depth of a derivation is measured as the maximum depth from the conclusion to one of the axioms. Another way of measuring depth could be to count all steps in the derivation, which is more interesting from a theoretical point of view. However, from an implementational point of view this is more complex as it requires another loop inside the iterative deepening process to divide the total number of steps between the premisses.

The procedure can be summarized in pseudocode as follows:

```
find_derivation(sequent):
   derivation = null;
   depth = 1;
   do {
      derivation = find_derivation_step(sequent, depth);
      depth++;
   } while(derivation == null);
   return derivation;

find_derivation_step(sequent, depth):
   if(depth == 0) {
      return null;
   }
   foreach(rule in rules) {
      if(rule.match(sequent)) {
         premisses = rule.premisses(sequent);
         pderivations = [];
         foreach(premisse in premisses) {
            derivation = find_derivation_step(premisse, depth-1);
            if(derivation != null) {
               pderivations.add(derivation);
            }
         }
         if(pderivations.length == premisses.length) {
            return new derivation(rule, sequent, pderivations);
         }
      }
   }
   return null;
```

Again careful readers might have noticed that in this case the searchspace is not organized as a DAG but as a tree, where multiple nodes in the tree could represent the same sequent. In this way it will happend that the same sequent will appear in multiple positions in the tree, and will therefore be handled multiple times. For (huge) efficiency improvements we will have to organize our search

space as a DAG (for **NL**, in (de Groote, 1999), the change from a tree to a DAG even moves it from `EXPTIME` to `PTIME`). In this way, for each depth any possible sequent is handled at most once.

This is accomplished by keeping track of all sequents that have been tried so far, along with their (possible) derivations and meta-information about the depth at which this sequent was tried. This information is stored in a *hashmap*, with the sequent as key and the following `struct` as value:

```
struct HashmapItem {
   int depth;
   Derivarion *derivation;
};
```

In the `find_derivation_step` function we can now check whether the given sequent was already tried. If a derivation is already present we can return this derivation, and if the depth is higher than the current depth we can safely return `null`, as we will not find a derivation of the given depth, otherwise it would have been found when this sequent was tried before. The `find_derivation_step` function is changed as follows:

```
find_derivation_step(sequent, depth):
   if(hashmap[sequent] != null) {
      if(hashmap[sequent].derivation != null) {
         return hashmap[sequent].derivation;
      } else if(hashmap[sequent].depth > depth) {
         return null;
      } else {
         hashmap[sequent].depth = depth;
      }
   } else {
      hashmap[sequent] = new HashmapItem(depth, null);
   }
   if(depth == 0) {
      ...
         if(pderivations.length == premisses.length) {
            hashmap[sequent].derivation =
               new derivation(rule, sequent, pderivations);
            return hashmap[sequent].derivation;
         }
      }
   }
   return null;
```

Apart from the increase in speed that this optimization provides us, there is another interesting property that comes with this change. For non-derivable sequents we would like to know that these are non-derivable, so the theorem prover should stop at a certain point and indicate that they are non-derivable. If we use

the approach without the hashmap then we will also search through redundant derivations, i.e. derivations where there is a sequent $X \rightarrow P$ at a certain point in the derivation, and then after a few (residuation or dual residuation) steps we have $X \rightarrow P$ again.

With the introduction of the hashmap these redundant derivations will not appear as we will not search further if we arrive at a sequent that has already been found earlier in the derivation. Therefore, in the redundant-free search with the hashmap, we will, for non-derivable sequents, at a certain point find the maximum depth. An upperbound for this depth has already been given in Lemma 3, but if we keep a boolean variable indicating whether we triggered the `if(depth == 0)` statement for a certain start depth, we can stop when this statement is not triggered anymore and the whole searchspace has been searched. This way we have a finite, although still exponential time, method for deciding whether the given sequent is derivable with the given inference rules.

9.3 Output

After the search procedure has been completed and a `Derivation` representation has been built by the search procedure, we would like to represent this proof object in a useful way. One of the possibilities is translating to a LaTeX representation, after which it is parsed and showed as a PDF. With the `proof.sty` package for LaTeX this is a trivial step and we will not go into this further.

Other mappings, that are not included in the current implementation, could be CPS translations or lexical semantics.

10 Further improvements

The implementation as discussed in the previous section is a usable theorem prover for categorial grammars like **NL** and **LG**. However, in order to make it faster and better some optimizations have been done which will be discussed next.

10.1 Polarity check

From the rules of inference it can be seen that the polarity of a formula or structure is never changed. Hence, for every $p \in Var$ in the sequent we can directly calculate its polarity and count all input and output occurrences. Because an axiom has exactly one input and one output literal, we know that for every derivable sequent for all $p \in Var : count(p^\bullet) = count(p^\circ)$. Thus, if this count is not equal for any literal the sequent can never be derivable and we can therefore return `null` immediately.

This polarity check will be done at the start of the derivation to make sure that the input sequent has the correct polarity. The check will also be done after every derivation step with more than 1 premisse, as the literals might not be divided over the premisses in a balanced way, and therefore it can happen that the polarity is not correct in some of the premisses.

10.2 Structure sharing

After application of an inference rule during the search process, new formulas, structures and sequents are constructed in memory. Although by means of the hashmap the same sequent will not be handled again in the search procedure, in memory there will be multiple instances of the same formulas, structures and sequents. To make use of the memory in a more efficient way, we make use of *structure sharing*.

In our implementation this means that we keep another hashmap with both the key and value being the formula (or structure or sequent). Whenever a new formula (structure, sequent) is created, the hashmap is checked to see whether this formula (structure, sequent) already exists. If this is the case, the object that is already present in the hashmap is taken, and the newly created one is removed from memory again. If it was not present the new instance is added to the hashmap.

10.3 Lexicon

The idea behind the theorem prover is to decide whether a sequent is derivable or not, but in practice we would also like to use it in applications of **LG**. For use in language models, the theorem prover has been built in such a way that a user can create a lexicon with a single type for each word. An example lexicon:

```
alice :: np.
bob :: np.
walks :: np \ s.
sees :: (np \ s) / np.
everybody :: s / (np \ s).
teases :: (np \ s) / np.
someone :: (s (/) s) (\) np.
```

If this lexicon file is called `lexicon.txt`, the theorem prover can be called as follows:

```
./LGprover lexicon.txt "alice teases someone" s
```

We can replace every word in the input sentence by the type from the lexicon, but because **LG** is non-associative we have to add brackets to this sentence first. The brackets will associate to the right by default, so in this case this means that this is parsed as `"(alice (teases someone))"`. Now the sequent that belongs to this input is:

$$\underbrace{np}_{alice} \cdot \otimes \cdot (\underbrace{((np \backslash s)/np)}_{teases} \cdot \otimes \cdot \underbrace{((s \oslash s) \oslash np)}_{someone}) \rightarrow s$$

In order to accomplish this is a full parser for formulas has been written. This parser is used both for the lexicon and for the given sentence and target type, to construct the target sequent that is given to the theorem proving function.

The parsing starts with a tokenizer that takes a string as input and returns a list of strings (tokens) as output. The tokenizer works as follows:

```
specialTokens = {"::", "(*)", "(/)", "(\)", "(+)", ".",
                 "/", "1", "0", "\", "(", ")"};

tokenize(string):
   tokens = []
   while(string.length > 0) {
      // remove whitespace from the beginning
      string = trim(string);

      // Try to match one of the special tokens
      found = false;
      foreach(specialToken in specialTokens) {
         if(startsWith(string, specialToken)) {
            tokens.add(specialToken);
            string = string.substring(specialToken.length);
            found = true;
            break;
         }
      }
      if(found) {
         continue;
      }

      // Try to match the longest literal string
      i = 0;
      while(isLiteralChar(string[i])) {
         i++;
      }
      tokens.add(string.substring(0, i));
      string = string.substring(i);
   }
   return tokens;
```

The tokenizer will now return a list of tokens (provided that the input was formatted in the correct way), and from this we can easily create a formula by recursively going over the list of tokens. Every time an opening bracket is found in the list the function is called recursively to parse the part in the brackets, and if a formula is followed by a binary connective the function is also called recursively to parse the righthand side. From this we automatically have implicit right associativity if brackets are not added to the input.

10.4 Derivational and spurious ambiguity

Until now the objective of the theorem prover has been: given a sequent, decide whether it is derivable or not in **LG**. However, for linguistic applications we would like to have not only this question answered, but also the question: how

many (essentially) different derivations do there exist for the given sequent. Therefore, we would like to adapt the theorem prover in such a way that it returns not only the shortest derivation, but also all essentially different derivations.

Let us first define what we mean by essentially different. We distinguish between 2 types of ambiguity:

- *Spurious ambiguity*: This is the type of ambiguity where there are multiple derivations for the same sequent that are different in the rules that are applied, but are the same from a *proof-term* point of view, like the $\bar{\lambda}\mu\tilde{\mu}$ calculus.
  For example if there is a derivation Γ, one can easily create another derivation which is almost the same as Γ, but add two r steps somewhere in the middle of the derivation. Because the first r step can be reversed in the seconds step, this is a valid derivation, but from the proof-term point of view it is exactly the same.
- *Derivational ambiguity*: In derivational ambiguity there exist multiple derivations for the given sequent, that are essentially different also from the proof-term point of view. This can for example be the case for ambiguous sentences like *"everybody teases someone"*, where different derivations can really represent different readings of the input sentence.

Now the question remains: in the derivation, how can we distinguish between spurious and derivational ambiguity? We can answer this question as follows:

Two derivations are considered essentially different if the order of application of the logical rules is different, where the logical rules are $\{\otimes R, /L, \backslash L, \oplus L, \oslash R, \oslash R\}$. For the $\bar{\lambda}\mu\tilde{\mu}$ calculus this will return redundant derivations, but it will always find all possible $\bar{\lambda}\mu\tilde{\mu}$ terms.[3] Hence, for the $\bar{\lambda}\mu\tilde{\mu}$ calculus this will give some *false positives*, but it will never have *false negatives*. We believe that this is also the case for most other interpretations of **LG**.

In the implementation of the theorem prover this means that a lot needs to be changed. First of all the derivation procedure should always search the whole search space in order to find all derivations. Furthermore, the `find_derivation` and `find_derivation_step` functions need to return a list of derivations. The `Derivation` representation should contain a list that indicates in which order the logical rules where applied on the sequents. This is done by indexing all logical connectives before the search procedure starts, and then at application of a logical rule store on which logical connective this rule applied. In this way, all essentially different derivations can be identified and returned.

11 Conclusion and future work

This theorem prover is currently at the level where it can be very useful, especially for smaller sequents like the ones in the linguistic applications. It is flexible

[3] At first it seemed that axiom linking could be used for this, but a counter-example is the sequent $(a \oslash (np \oslash a)) \cdot \otimes \cdot (((np\backslash s)/np) \cdot \otimes \cdot (b \oslash (np \oslash b))) \rightarrow s$ which has at least two $\bar{\lambda}\mu\tilde{\mu}$ terms with the same axiom linking.

and can easily be adapted to work with other derivation rules like the *Grishin class I interactions*. One could easily translate the `Derivation` representation to other representations like a Prolog or Haskell representation to work with other code.

Still, this is far from the most efficient implementation of a theorem prover for **LG**. There has been a lot of research after theorem provers in general, and there are a lot of improvements that could be done. We will discuss some of them below.

11.1 Top-down vs bottom-up search

The top-down approach has been chosen for this theorem prover, and of course another approach would be to do it bottom-up. It is not know whether this will be faster or slower, but it is also possible to combine the 2 approaches (Fuchs, 1998). This is a totally different approach that did not fit in the timeframe of this thesis, but it is an interesting approach to the general problem of theorem proving.

11.2 Rule compilation

The current representation of the inference rules provides a flexible framework in which new rules can easily be added. However the price that has to be paid for this flexibility is efficiency, as this implementation of the inference rules creates extra work during the search process. A possible solution for this could be to do some kind of *preprocessing* on the inference rules, so that the parsing of the flexible implementation has to be done only once.

In the current approach, for every sequent in the search process we try to apply all rules. There are however 3 types of sequents (formula left, formula right and structural) and any rule matches on only one of these 3 types. A simple improvement would be to divide the rules in 3 groups, one for each of the sequent types. Then whenever all rules that apply on a certain sequent have to be found, only the appropriate list of inference rules has to be considered.

11.3 Focussed proof search

A whole different approach for searching the search space is the approach of *focussed proof search* (Andreoli, 2002). The idea of focussed proof search is that multiple steps are done in parallel if possible, thereby restricting the search space and removing some spurious ambiguity. For example if we have the sequent

$(A \otimes B) \cdot \otimes \cdot (C \otimes D) \rightarrow E$, we can have the following two derivations:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \cdot \otimes \cdot B) \cdot \otimes \cdot (C \cdot \otimes \cdot D) \rightarrow E}{C \cdot \otimes \cdot D \rightarrow (A \cdot \otimes \cdot B) \cdot \backslash \cdot E}\, r}{C \otimes D \rightarrow (A \cdot \otimes \cdot B) \cdot \backslash \cdot E}\, \otimes L}{(A \cdot \otimes \cdot B) \cdot \otimes \cdot (C \otimes D) \rightarrow E}\, r}{A \cdot \otimes \cdot B \rightarrow E \cdot / \cdot (C \otimes D)}\, r}{A \otimes B \rightarrow E \cdot / \cdot (C \otimes D)}\, \otimes L}{(A \otimes B) \cdot \otimes \cdot (C \otimes D) \rightarrow E}\, r$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \cdot \otimes \cdot B) \cdot \otimes \cdot (C \cdot \otimes \cdot D) \rightarrow E}{A \cdot \otimes \cdot B \rightarrow E \cdot / \cdot (C \cdot \otimes \cdot D)}\, r}{A \otimes B \rightarrow E \cdot / \cdot (C \cdot \otimes \cdot D)}\, \otimes L}{(A \otimes B) \cdot \otimes \cdot (C \cdot \otimes \cdot D) \rightarrow E}\, r}{C \cdot \otimes \cdot D \rightarrow (A \otimes B) \cdot \backslash \cdot E}\, r}{C \otimes D \rightarrow (A \otimes B) \cdot \backslash \cdot E}\, \otimes L}{(A \otimes B) \cdot \otimes \cdot (C \otimes D) \rightarrow E}\, r$$

In focussed proof search the two $\otimes L$ rule are done in parallel, and for this part of the derivation there will only be 1 possible derivation.

11.4 Lexical ambiguity

In the current implementation every word in the lexicon can have exactly 1 type. However, in natural language words often can have multiple functions in a sentence. Therefore it would be useful for a theorem prover to support lexical ambiguity, but unfortunately this will add another layer of complexity. If the input sentence consists of n words, and every word has exactly 2 types assigned to it, there are 2^n possible sequents that can be constructed for this sentence, and in this implementation all these sequents have to be considered.

11.5 Parsing vs Recognition

While the current implementation can do *recognition*, it does not yet do *parsing*. In recognition the input is a phrase (a bracketed string) which can easily be translated to a sequent of **LG**, but in parsing the input is a sentence, a string without any bracketing. For working with natural language the theorem prover obviously needs to do the parsing, but unfortunately the best known solution for this (for **LG**) is to try all possible bracketings, which is again an exponential time procedure.

Bibliography

Andreoli, J. M. (2002). Focussing proof-net construction as a middleware paradigm. In *Automated Deduction–CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 165–187. Springer Berlin / Heidelberg.

Bernardi, R. and Moortgat, M. (2010). Continuation semantics for the Lambek-Grishin calculus. *Information and Computation*, 208(5):397–416. Special Issue: 14th Workshop on Logic, Language, Information and Computation (WoLLIC 2007).

Capelletti, M. (2007). *Parsing with Structure-Preserving Categorial Grammars*. PhD thesis, Utrecht Institute of Linguistics OTS, Utrecht University.

Cook, S. A. and Reckhow, R. A. (1979). The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):36–50.

Curien, P.-L. and Herbelin, H. (2000). The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM.

de Groote, P. (1999). The Non-associative Lambek Calculus with Product in Polynomial Time. In *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1617 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Foret, A. (2003). On the computation of joins for non associative Lambek categorial grammars. In *Proceedings of the 17th International Workshop on Unification Valencia, Spain, June 8-9, (UNIF'03)*.

Fuchs, D. (1998). Cooperation between top-down and bottom-up theorem provers by subgoal clause transfer. In *Artificial Intelligence and Symbolic Computation*, volume 1476 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Goré, R. (1998). Substructural logics on display. *Logic Jnl IGPL*, 6(3):451–504.

Joshi, A. and Schabes, Y. (1997). Tree-adjoining grammars.

Kandulski, M. (1988). The non-associative Lambek calculus. *Categorial Grammar, Linguistic and Literary Studies in Eastern Europe (LLSEE)*, 25:141–151.

Lambek, J. (1958). The Mathematics of Sentence Structure. *American Mathematical Monthly*, 65:154–170.

Lambek, J. (1961). On the calculus of syntactic types. *Structure of Language and Its Mathematical Aspects*, pages 166–178.

Melissen, M. (2009). The generative capacity of the Lambek-Grishin calculus: A new lower bound. In de Groote, P., editor, *Proceedings 14th conference on Formal Grammar*, volume 5591 of *Lecture Notes in Computer Science*. New York: Springer.

Moortgat, M. (2007). Symmetries in Natural Language Syntax and Semantics: The Lambek-Grishin Calculus. In *Logic, Language, Information and Computation*, volume 4576 of *Lecture Notes in Computer Science*, pages 264–284. Springer Berlin / Heidelberg.

Moortgat, M. (2009). Symmetric categorial grammar. *Journal of Philosophical Logic*, 38(6):681–710.

Moot, R. (1999). Grail: an interactive parser for categorial grammars. In *Proceedings of VEXTAL99, University C Foscari*, pages 255–261.

Moot, R. (2007). Proof nets for display logic. *CoRR*, abs/0711.2444.

Moot, R. (2008). Lambek grammars, tree adjoining grammars and hyperedge replacement grammars. In Gardent, C. and Sarkar, A., editors, *Proceedings of TAG+9, The ninth international workshop on tree adjoining grammars and related formalisms*, pages 65–72.

Pentus, M. (1993a). The conjoinability relation in Lambek calculus and linear logic. ILLC Prepublication Series ML–93–03, Institute for Logic, Language and Computation, University of Amsterdam.

Pentus, M. (1993b). Lambek grammars are context free. In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 429–433, Los Alamitos, California. IEEE Computer Society Press.

Pentus, M. (2003). Lambek calculus is NP-complete. CUNY Ph.D. Program in Computer Science Technical Report TR–2003005, CUNY Graduate Center, New York.

Savateev, Y. (2009). Product-Free Lambek Calculus Is NP-Complete. In *LFCS '09: Proceedings of the 2009 International Symposium on Logical Foundations of Computer Science*, pages 380–394, Berlin, Heidelberg. Springer-Verlag.