HAL open science

# Type-Logical and Hyperedge Replacement Grammars

Richard Moot

## To cite this version:

Richard Moot. Type-Logical and Hyperedge Replacement Grammars. [Research Report] LaBRI; INRIA. 2008, pp.70. inria-00413340

**HAL Id: inria-00413340**

**https://inria.hal.science/inria-00413340v1**

Submitted on 3 Sep 2009

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Type-Logical and Hyperedge Replacement Grammars

Draft

Richard Moot
LaBRI (CNRS), INRIA Bordeaux SW, University of Bordeaux

September 19, 2008

# Contents


**1 Introduction** **1**

**2 Hyperedge Replacement Grammars** **3**
2.1 Hypergraphs . . . . . 3
2.2 Hyperedge Replacement . . . . . 4
2.3 Hyperedge Replacement Grammars . . . . . 5
2.4 Basic Results for Hyperedge Replacement Grammars . . . . . 7

**3 Proof Nets for Type-logical Grammars** **9**
3.1 Proof Structures . . . . . 9
3.2 Abstract Proof Structures . . . . . 14
3.3 Contractions and Structural Rules . . . . . 16
3.4 The Lambek-Grishin Calculus . . . . . 21
3.5 Proof Nets . . . . . 23
3.6 Type-logical Grammars . . . . . 26
3.7 Analysis of the Structural Rules . . . . . 27

**4 Type-Logical Grammars as HR Grammars** **33**
4.1 A Hyperedge Replacement Grammar for Abstract Proof Nets . 33
4.2 A Hyperedge Replacement Grammar for Proof Nets . . . . . 35
4.3 Soundness and Completeness . . . . . 35
4.4 A Hyperedge Replacement Grammar for Type-Logical Grammars 39

**5 Conclusions** **43**

**A Complete HR Grammar for NL$\diamond$** **49**
A.1 Start: Initial Axiom . . . . . 49
A.2 Tensor: Binary . . . . . 49
A.3 Tensor: Unary . . . . . 50
A.4 Par: L$\bullet$ . . . . . 50

A.5 Par: R/ . . . . . 51
A.6 Par: R\ . . . . . 52
A.7 Par: L$\diamond$ . . . . . 53
A.8 Par: R$\square$ . . . . . 53
A.9 End: Cut/Flow/Axiom . . . . . 54

**B Adding Structural Rules: HRG for NL$\diamond_{\mathcal{R}}$ 55**
B.1 Multimodality . . . . . 55
B.2 Mixed Associativity and Mixed Commutativity . . . . . 55
B.3 Mixed Associativity and Mixed Commutativity With Unary Control . . . . . 56
B.4 The K1, K2 interaction rules . . . . . 58

**C HR Rules for Lambek-Grishin 61**
C.1 Start: Initial Axiom . . . . . 61
C.2 Tensor: Binary . . . . . 61
C.3 Par: L$\bullet$ . . . . . 62
C.4 Par: R$\odot$ . . . . . 63
C.5 Par: R\ . . . . . 63
C.6 Par: L$\oslash$ . . . . . 64
C.7 Par: R/ . . . . . 64
C.8 Par: L$\oslash$ . . . . . 65
C.9 End: Cut/Flow/Axiom . . . . . 65
C.10 The Class IV Interactions . . . . . 66

# CHAPTER 1

# INTRODUCTION

Type-logical grammars (Moortgat 1997, Morrill 1994) are a family of logical calculi — with, as the name suggests, a strong relation to the theory of types — for natural language analysis and natural language semantics. Type-logical grammars were introduced by Lambek (1958), who extends the work of Ajdukiewicz (1935) and Bar-Hillel (1964) — which have only elimination rules — by adding the corresponding introduction rules and giving cut-elimination and decidability proofs. The addition of the introduction rules, besides being desirable from a logical point of view has descriptive advantages as well, permitting a treatment of *wh* extraction (at least in peripheral cases) and quantifier scope ambiguities (again with some limitations).

Given that it has long been suspected that Lambek grammars generate only context-free grammars — though this was proved only in (Pentus 1997) — different extensions to the Lambek calculus have been proposed in the literature, notably to deal with linguistic phenomena for which no satisfactory Lambek calculus treatment exists, such as medial extraction or the crossing dependencies of Dutch subordinate clauses. These additions are the use of *modes* to distinguish different modes of composition each with their own family of connectives. The addition of *structural rules* to allow more flexibe compisition of categories and the addition of *unary control operators*. The resulting logic takes the non-associative Lambek calculus **NL** as a base, adds modes and unary control operators and a set of structural rules $\mathcal{R}$, a small set of structural rules fixed for a grammar. The resulting logic is called $\mathbf{NL}\Diamond_{\mathcal{R}}$, or, when $\mathcal{R} = \emptyset$, just $\mathbf{NL}\Diamond$.

Relatively little is known about the exact class of languages generated by $\mathbf{NL}\Diamond_{\mathcal{R}}$. Moot (2002) shows that when we disallow structural rules to increase the total number of unary connectives then $\mathbf{NL}\Diamond_{\mathcal{R}}$ generates exactly the context-sensitive languages, which implies that the decision problem for $\mathbf{NL}\Diamond_{\mathcal{R}}$ is PSPACE complete. It is also known that $\mathbf{NL}\Diamond$ generates exactly the context-free languages and polynomial algorithms are known in this case (de Groote

1999, Capelletti 2007). While these results establish interesting upper and lower bounds, we would like to find fragments of $\mathbf{NL}\Diamond_{\mathcal{R}}$ which generate a bit more than just the context-free languages, though less than the full class of context-sensitive languages while maintaining a polynomial time parsable formalism. The class of mildly context-sensitive grammar formalisms appears to be a good compromise between parsing complexity and language classes generated.

In order to indentify mildly context-sensitive and polynomially parsable fragments of $\mathbf{NL}\Diamond_{\mathcal{R}}$, we will related two classes of (hyper-)graph languages: *hyperedge replacement grammars* and *proof nets* a graph-like representation of proofs in $\mathbf{NL}\Diamond_{\mathcal{R}}$.

Hyperedge replacement grammars (introduced in Chapter 2) are a type of context-free graph grammar where we replace a hyperedge by a hypergraph (Engelfriet 1997). Hyperedge replacement grammars have been well-studied in the context of theoretical computer science and many results about the classes of string languages, tree languages and graph languages generated are known (Engelfriet 1997, Drewes, Habel & Kreowski 1997, both give good overviews of many of the results for hyperedge replacement grammars).

Proof nets (introduced in Chapter 3) are a graph-like presentation which has several advantages over the other calculi. They are redundancy-free, in the sense that different proof nets correspond to different lambda-term semantics, much like natural deduction for the $(\rightarrow, \wedge)$ fragment of intuitionistic logic. In addition, they have been shown to share some proporties with human sentence processing (Morrill 1998, Johnson 1998).

In Chapter 4, I will relate hyperedge replacement grammars to type-logical proof nets. In particular, I will give hyperedge replacement grammars for several different, commonly used but restricted, Lambek calculi, $\mathbf{NL}\Diamond$, $\mathbf{NL}\Diamond_{\mathcal{R}}$ (for several instances of $\mathcal{R}$) and LG and show that for each of these systems there is a strongly equivalent hyperedge replacement grammar.

This has several important consequences: first of all it gives a characterization of the tree languages generated by type-logical grammars, which is exactly the class of tree languages generated by tree adjoining grammars. Secondly, it gives us new polynomial parsing algorithms for several restricted fragments of type-logical grammars and opens the door for the treatment other fragments.

# CHAPTER 2

# HYPEREDGE REPLACEMENT GRAMMARS

## 2.1 Hypergraphs

A *hypergraph* generalises the notion of graph by allowing the edges, called *hyperedges*, to connect not just two but any number of nodes. Hypergraphs will de the data structure used both for type-logical proof nets and for hyperedge replacment grammars. There are slight differences between authors on the definition of hypergraphs. In the current paper, I will follow Engelfriet (1997), but use $\sigma$ instead of $\Sigma$ for the alphabet of selectors, reserving $\Sigma$ for the alphabet of non-terminal word labels later.

**Definition 2.1** *Let $\Gamma$ be an alphabet of edge labels and let $\sigma$ be an alphabet of selectors. A* hypergraph *over $\Gamma$ and $\sigma$ is a tuple $\langle V, E, lab, nod, ext\rangle$, where*

*$V$ is the finite set of* vertices,

*$E$ is the finite set of* hyperedges *disjoint with $V$,*

*lab is the* labeling function, *from $E$ to $\Gamma$, assigning an edge label to each hyperedge,*

*nod is the* incidence function *that associates with each edge $e \in E$ a partial function $nod(e) : \sigma \to V$, that is, it selects a vertex for every selector $\sigma$ of the edge.*

*ext is the* external function, *a partial function from $\sigma$ to $V$, that is, for every selector $\sigma$ of the hypergraph it select a vertex.*

**Definition 2.2** *The* type *of a hypergraph $H$ is the domain of the external function, $type(H) = dom(ext)$. The type of an edge $e$ is the domain of the incidence fuction $type(e) = dom(nod(e))$.*

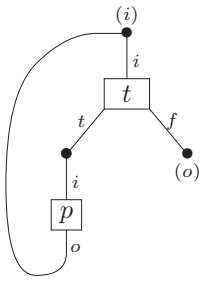



Figure 2.1: Hypergraph of a flowchart

To simplify the results and discussion which follow, we will assume that edge labels are ($\sigma$-)typed, that is to say that every edge $e$ with label $lab(e)$ will always appear with the same set of selectors.

The external nodes are special nodes in the graph which are identified by the selectors. Their goal is to define the embedding mechanism of hyperedge replacement grammars, where we will replace a hyperedge of a certain type by a hypergraph of the same type.

**Example 2.3** *As an example, inspired by Habel & Kreowski (1987), Figure 2.1 gives a hypergraph representing the flowchart of a very simple program. The selector alphabet $\sigma$ has member $i$ (input), $o$ (output), $t$ (true) and $f$ (false). The complete hypergraph has one input, the external vertex labeled $(i)$ and one output, the external vertex labeled $(o)$. The hyperedge $t$ is of type $\{i, t, f\}$. It represents a boolean test with a* false *branch exiting this piece of code immediately and a* true *branch executing a basic program $p$ (of type $\{i, o\}$) then returning to the boolean test. As should be clear from the intended meaning of this hypergraph, it is an abstract representation of a* while *loop.*

## 2.2 Hyperedge Replacement

The operation of hyperedge replacement replaces a hyperedge by a hypergraph $H$ of the same type. Basically, we delete the hyperedge then add a (disjoint copy of) $H$ and finally we identify each external node of $H$ with the node adjacent to the same selector of the deleted hyperedge. Formally, this is defined as follows.

**Definition 2.4** *Let $H$ and $K$ be two disjoint hypergraphs with the same set of edge labels $\Gamma$ and the same set of selectors $\sigma$. Let $e$ be an edge of $H$ such that $type(e) = type(K)$. The* hyperedge replacement *of $e$ by $G$, $H[e := G] = \langle V, E, lab, nod, ext\rangle$ is defined as follows.*

$$V = V_H \cup V_K$$

$$E = (E_H - e) \cup E_k$$

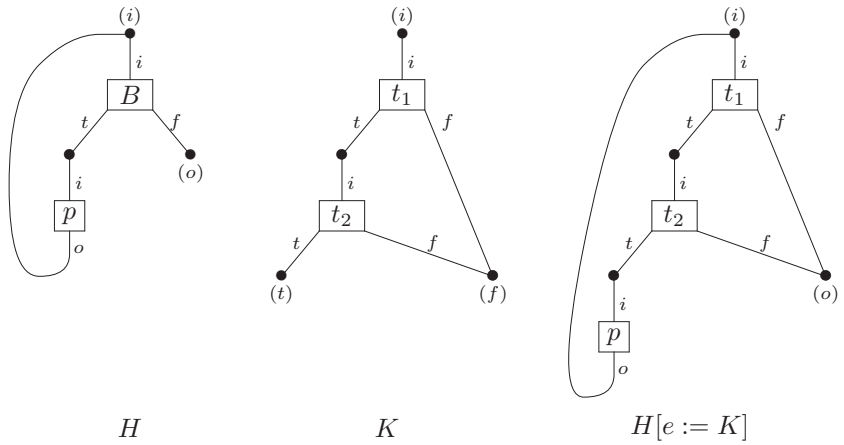



Figure 2.2: Hyperedge replacement

*lab* $=$ *lab*$_H \cup$ *lab*$_K$ *restricted to the members of* $E$.

*nod* $=$ *nod*$_H \cup$ *nod*$_K$ *restricted to the members of* $E$.

*ext* $=$ *ext*$_H$

*For all* $s \in$ *type*$(e)$, *nod*$_H(e, s) =$ *ext*$_K(s)$.

**Example 2.5** *Figure 2.2 shows how the hyperedge labeled* $b$ *of graph* $H$ *is replaced by hypergraph* $K$.

It is well-known that hyperedge replacement is both confluent and associative (Courcelle 1987, Lautemann 1990). Confluent in this context means that whenever we have a hypergraph $H$ with two distinct hyperedges $e_1$ and $e_2$ then $H[e_1 := K_1][e_2 := K_2] = H[e_2 := K_2][e_1 := K_1]$. That is to say we can change the order of the replacement of two distinct hyperedges without changing the resulting hypergraph.

In this context, associative means that when we have a hyperedge $e_1$ of $H$ and a hyperedge $e_2$ of $K_1$, then $H[e_1 := K_1][e_2 := K_2] = H[e_1 := K_1[e_2 := K_2]]$. That is to say we can perform a substitution ...

## 2.3 Hyperedge Replacement Grammars

Hyperedge replacement grammars were introduced by Bauderon & Courcelle (1987) and Habel & Kreowski (1987) and have been applied to ...

Engelfriet (1997) gives an overview of hyperedge and node replacement grammars and several of the resuls obtained for them.

**Definition 2.6** *A* hyperedge replacement grammar *(or HR grammar) is a tuple* $G = \langle N, T, \sigma, P, S \rangle$ *such that.*

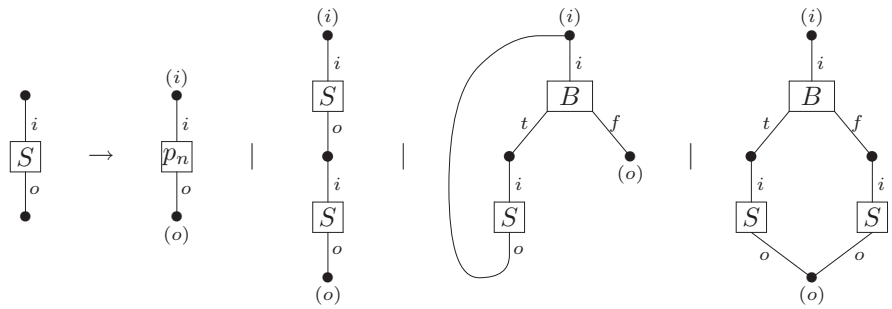



Figure 2.3: Hyperedge replacement grammar for flowcharts — statements

*$N$ is the alphabet of nonterminal edge labels.*

*$T$ is the disjoint alphabet of terminal edge labels.*

*$\sigma$ is the alphabet of selectors.*

*$P$ is the finite set of productions.*

*$S \in N$ is the start nonterminal symbol.*

**Example 2.7** *Example 2.3 showed a hypergraph representation of a flowchart. Figures 2.3 and 2.4 show a hyperedge replacement grammar, very close to the one proposed by Habel & Kreowski (1987), generating such hypergraphs. The grammar has two nonterminal symbols: $S$ (for* statement*) of type $\{i, o\}$ and $B$ (for* boolean*) of type $\{i, t, f\}$.*

*As shown in Figure 2.3, a statement is either an elementary program statement $p_n$, a combination of two statements, a* while *loop, which executes a statement until a boolean test returns false or an* if ... then ... else *statement, which executes one of two program statements depending on whether a boolean returns true or false.*

*Boolean tests for our toy programming language are shown in Figure 2.4. Again, a boolean test can be an elementary test $t_n$. But if can also be a combination of two boolean tests: either by* and *leaving by the $(f)$ node as soon as one of the two tests fails or by* or *leaving by the $(t)$ node as soon as one of the two tests succeeds.*

The notation of derivation and of languages generated generalize easily from those of context-free string languages.

**Definition 2.8** *Let $G$ be a hyperedge replacement grammar and $H$ a hypergraph. We say that $G$* derives *$H$ iff there is a sequence*

$$S \to \ldots \to H$$

*such that $S$ is the start symbol of the grammar and every step in the derivation corresponds to a hyperedge replacement according to a production in $P$*

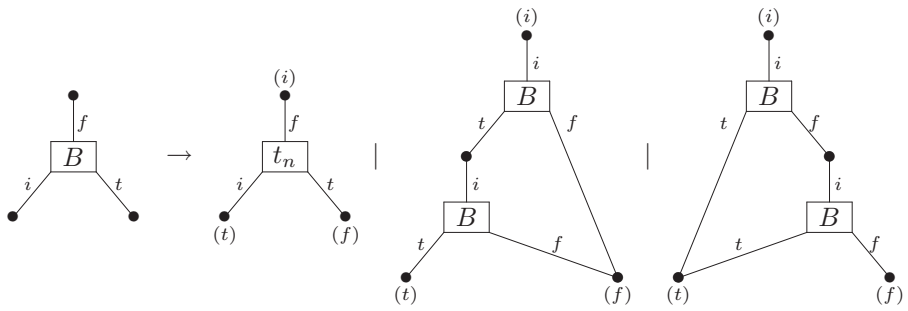



Figure 2.4: Hyperedge replacement grammar for flowcharts — booleans

**Definition 2.9** *Let $G$ be a hyperedge replacement grammar. The* language generate by *$G$ is the set of hypergraphs without hyperedges labeled by nonterminal edge labels derivable from $S$.*

## 2.4 Basic Results for Hyperedge Replacement Grammars

As attested by the aforementioned overviews (Engelfriet 1997, Drewes et al. 1997), hyperedge replacement grammars have are an area of active research with many results. In this section, I will provide an admittedly ecclectic selection of results which will prove useful later.

**Definition 2.10** *The* rank *of a terminal or nonterminal symbol is the number of its tentacles.*

*The* rank *of a hyperedge replacement grammar is the maximum rank of a nonterminal symbol in the grammar.*

Given that each additional tentacle permits us to have access to an additional vertex in the hypergraph, it should come as no surprise that increasing the rank of a hyperedge replacement grammar increases the language generated by the grammar.

It will often be convenient to look at hyperedge replacement grammars where we require all tentacles to reach distinct vertices and where we require all external vertices to be distinct. This last condition corresponds to not allowing a nonterminal to rewrite to the empty string in context-free string grammars.

**Definition 2.11** *Let $g$ be a hyperedge replacement grammar and $H$ the right hand side of a rule in $g$.*

- *$g$ is* loop-free *iff for every nonterminal $e$ of every right hand side $H$ of a rules in $g$ and for all $s_1, s_2 \in type(e)$ whenever $nod_H(e, s_1) = nod_H(e, s_2)$ then $s_1 = s_2$.*

- *$g$ is* identification-free *iff for every right hand side $H$ of a rule in $g$ and for all $s_1, s_2 \in type(H)$ whenever $ext_H(s_1) = ext_H(s_2)$ then $s_1 = s_2$.*

**Lemma 2.12 ((Engelfriet & Heyker 1992))** *For every hyperedge replacement grammar $g$ there is a hyperedge replacement grammar $g'$ which is loop-free and identification-free and which generates the language $L(g') = \{H \in L(g) | H \text{is identification-free}.$*

I will not consider loops in the current paper and therefore make the quiet assumption that all tentacles with different labels will reach different nodes. However, it is sometimes convenient to allow rules in our grammar to identify nodes. Even though by Lemma 2.12 they can be removed from the hyperedge replacement grammars I propose — and for parsing the hyperedge replacement grammars in this article it is generally necessary that they are — in the interest of keeping the grammars as compact as possible, I will use hyperedge replacement rules which identify nodes.

# CHAPTER 3

# PROOF NETS FOR TYPE-LOGICAL GRAMMARS

Proof nets are a way of representing proofs in linear logic introduced by Girard (1987). Proof nets naturally factor out the 'bureaucratic' rule permutations which are possible in the sequent calculus and natural deduction formulations of linear logic.

Proof nets are usually described in three steps: first, we start with a more general set of structures which includes all provable statements but also some non-provable statements. These structures are *proof structures*. For the second step, we 'forget' about a lot of the information of the proof structures. For example, for the Danos-Regnier switching criterion (Danos & Regnier 1989) we forget about the actual formulas in the proof structure and obtain just a set of graphs, the correction graphs. Similarly, we obtain *abstract proof structures* by forgetting about the internal formulas of a proof structure and by forgetting most of the names of the logical rules used. Finally, we state a *correctness condition* such that any proof structure of which the underlying abstract structure satisfies this condition is a proof net. For the Danos-Regnier criterion we verify that all correction graphs are acyclic and connected. For the type-logical proof nets, we use a contraction criterion in the style of Danos (1990).

For the description of proof nets in the current chapter, I follow Moot & Puite (2002) but define the proof structures in terms of hypergraphs to make the equivalence proof with hyperedge replacement grammars simpler.

## 3.1 Proof Structures

**Definition 3.1** *A* proof structure $\mathcal{P}$ *is a tuple* $\langle H, V, A, frm, pre, con\rangle$ *such that* $H$ *is a hypergraphwith selectors* $\sigma = \{s1, s2, t\}$[1] *and edge labels.*

[1]The selector labels represent 'source 1', 'source 2' and 'target' respectively: in the notation of Habel & Kreowski (1987) the source selector labels would be arrows *entering* the hyperedge and the

| | |
|---|---|
| $h, c$ | *of type* $\{s1\}$ |
| $a^+, a^-$ | *of type* $\{s1\}$ *for all* $a \in A$ |
| $L\Diamond, R\Diamond, L\Box, R\Box$ | *of type* $\{s1, t\}$ |
| $L/, R/, L\bullet, R\bullet, L\backslash, R\backslash$ | *of type* $\{s1, s2, t\}$ |

*and the empty set of external nodes.*

*$V$ is the set of vertices in the hypergraph.*

*$A$ is the set of atomic formulas. As shown above, each atomic formula $a \in A$ induces a positive hyperedge $a^+$ as well as a negative hyperedge $a^-$.*

*frm is a function from vertices to formulas such that for all $v \in V$, frm$(v)$ is a member of the set of formulas $F$, such that the neighborhood of every hyperedge is of one of the forms shown in Figure 3.1.*

*con and pre are functions from edge labels $\Gamma$ to lists of selectors, such that for an edge $e$ with label lab$(e) = l$ , pre$(l) \in \sigma^*$. That is, the function pre returns is a list of selectors indicating the list of* premisses *of the edges with label $l$: if selector $s$ is a member of pre$(l)$, then for an edge $e$ with this label, the vertex nod$(e, s)$ is a premiss of the edge and if selector $s_1$ precedes selector $s_2$ in pre$(l)$ then vertex nod$(e, s_1)$ occurs to the left of vertex nod$(e, s_2)$. The function con$(l)$ is a list of selectors indicating the list of* conclusions *of the edge.*

*For every edge label $l$ of type $t$, pre$(l)$ and con$(l)$ partition the set $t$ into two disjoint subsets: every member of $t$ occurs exactly once in either pre$(l)$ or con$(l)$.*

*Every vertex is incident to exactly two hyperedges: once as a premiss and once as a conclusion.*

*Table 3.1 shows the value of the premisses and the conclusions for the different link types. Remark the following:*

- *In Figure 3.1 the premisses are drawn from left to right above the edge whereas the conclusions are drawn from left to right below it. Note, however, than only the labels are important and that sometimes it will be convenient when drawing a graph to abandon this convention.*

- *The left and right link of a connective are inverse to eachother with respect to premisses and conclusions, that is to say for any connective $c$ we have pre$(Lc) =$ con$(Rc)$ and con$(Lc) =$ pre$(Rc)$. These are the symmetries of our logical calculus.*

*All vertices which are incident to an $h$ edge are called the* hypotheses *of the proof structure. All vertices which are incident to a $c$ edge are called the* conclusions *of the proof structure.*

There are few graphical and notational differences with the proof structures of Moot & Puite. I will discuss each of them in turn.

1. I use special $h$ and $c$ hyperedges to indicate the hypotheses and conclusions of the proof structure. This makes our proof structures more regular in that every formula is now the premiss exactly one link and the conclusion of exactly one link.

target selector and arrow *leaving* it, which is the way the links are presented in (Moot & Puite 2002)

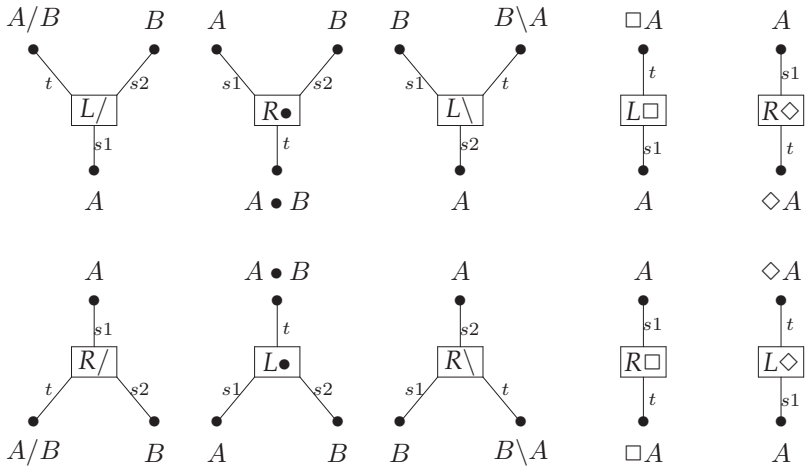



Figure 3.1: Hyperedges for **NL**◇ proof structures

| | | | |
|---|---|---|---|
| $pre(h)$ | $=[]$ | $con(h)$ | $=[s1]$ |
| $pre(c)$ | $=[s1]$ | $con(c)$ | $=[]$ |
| $pre(L\Diamond)$ | $=[t]$ | $con(L\Diamond)$ | $=[s1]$ |
| $pre(R\Diamond)$ | $=[s1]$ | $con(R\Diamond)$ | $=[t]$ |
| $pre(L\Box)$ | $=[t]$ | $con(L\Box)$ | $=[s1]$ |
| $pre(R\Box)$ | $=[s1]$ | $con(R\Box)$ | $=[t]$ |
| $pre(L/)$ | $=[t, s2]$ | $con(L/)$ | $=[s1]$ |
| $pre(R/)$ | $=[s1]$ | $con(R/)$ | $=[t, s2]$ |
| $pre(L\bullet)$ | $=[s1, s2]$ | $con(L\bullet)$ | $=[t]$ |
| $pre(R\bullet)$ | $=[t]$ | $con(R\bullet)$ | $=[s1, s2]$ |
| $pre(L\backslash)$ | $=[s1, t]$ | $con(L\backslash)$ | $=[s2]$ |
| $pre(R\backslash)$ | $=[s2]$ | $con(R\backslash)$ | $=[s1, t]$ |

Table 3.1: The functions *pre* and *con* for proof structures

2. I use explicit hyperedges corresponding to the positive and negative atomic formulas, the axiom/cut rule discussed below allows us to eliminate a postive and negative atomic hyperedge by identifying the vertices which are incident to the two links.

3. The list of premisses and list of conclusions of links are replaced by the functions *pre* and *con*.

4. The main formula of a link is the vertex which is reached by the $t$ selector, the active formulas are the vertices reached by the $s1$ and $s2$ selectors.

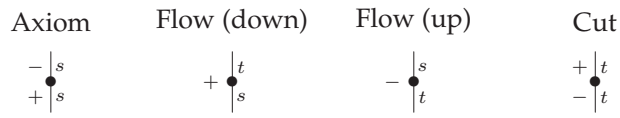



Figure 3.2: Cut, flow and axiom vertices

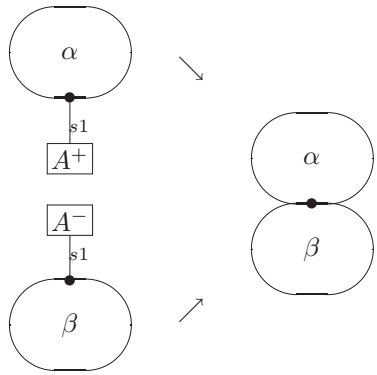



Figure 3.3: The axiom/cut rule

5. The difference between par and tensor links is not indicated explicitly but can be trivially recovered from the edge labels: $L/$, $R\bullet$, $L\backslash$, $L\Box$ and $R\Diamond$ (the entire top row of Figure 3.1) are tensor links and $R/$, $L\bullet$, $R\backslash$, $R\Box$ and $L\Diamond$ (the entire bottom row of the figure) are par links.

**Definition 3.2** *If a vertex is incident to two selectors* $\in \{s1, s2\}$*, we will call it an* axiomatic vertex*, if it is incident to one* $s$ *and one* $t$ *selector we will call it a* flow vertex *and if it is incident to two* $t$ *selectors we will call it a* cut vertex*.*

Proof nets for intuitionistic logic often use a notion of *polarity* to distinguish negative (antecedent) formulas from positive (succedent) formulas. It is possible to assign polarities to the vertices in proof structures here as well, but we need to keep in mind that axiom and cut formulas perform the role of *both* a positive and a negative formula. The way to see this is that an axiom is a negative formula with respect to the hyperedge of which it is a conclusion and a positive formula with respect to the hyperedge of which it is a premiss.

Figure 3.2 shows the different types of vertices. In all cases, the vertex is a conclusion of the hyperedge with the tentacle connecting it from above and a premiss of the hyperedge with the tentacle connecting it at the bottom.

The hyperedges corresponding to positive and negative atomic formulas are eliminated by means of the *axiom* or *cut* rule, shown in Figure 3.3. This rule allows us to connect two arbitrary formulas of opposite polarity; they need not be disjoint as suggested by the figure.

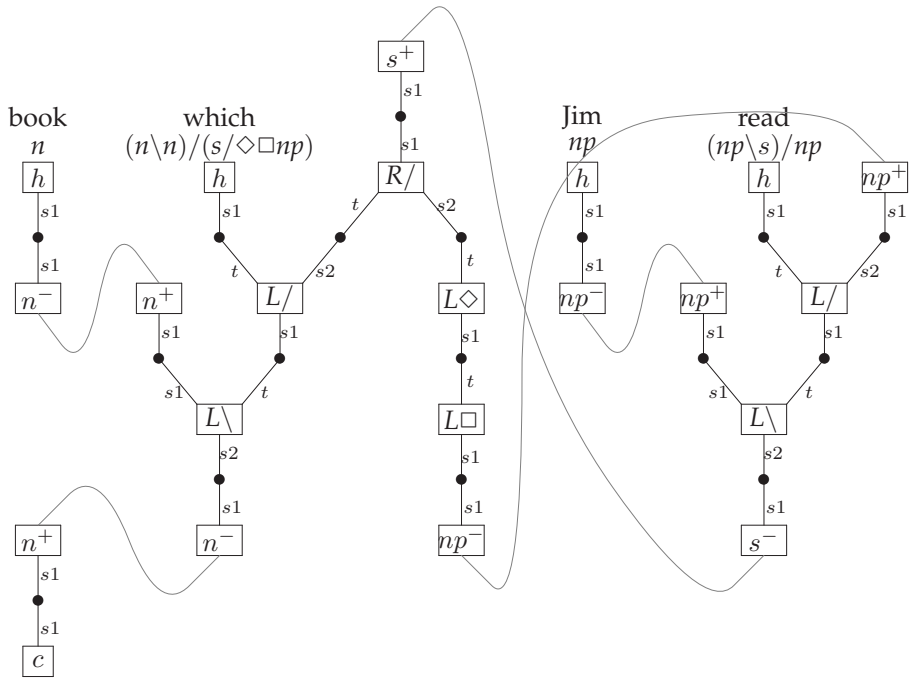



Figure 3.4: Example lexical graphs and a possible set of axiom connections

**Definition 3.3** *Let $F$ be an* ***NL***$\diamond$ *formula. Its* negative unfolding *is the proof structure we obtain by letting every subformula occurrence $f$ of $F$ correspond to a vertex $v \in V$ such that $frm(v) = f$, each local neighborhood corresponds to one of the forms shown in Figure 3.1, the vertex with $frm(v) = F$ is incident to an $h$ hyperedge and all atomic vertices are incident to either an $a^+$ or $a^-$ hyperedge in such a way that:*

- *all vertices adjacent to an $a^+$ or $a^-$ hyperedge are axiomatic vertices,*
- *all other vertices except are flow vertices.*

*The* positive unfolding *of an* ***NL***$\diamond$ *formula $F$ is defined analogously, but with the vertex such that $frm(v) = F$ incident to a $c$ hyperedge.*

**Example 3.4** *As an example Figure 3.4 gives the negative unfoldings of the formulas $n$, $(n\backslash n)/(s/\diamond\Box np)$, $np$ and $(np\backslash s)/np$ and the positive unfolding of $n$.*

*The grey lines indicate a possibility for the axiom identifications, the result of which is the proof structure shown in Figure 3.5.*

In the example above, all identifications are of atomic formulas and they result in axiomatic vertices in the resulting proof structure. Connecting an axiomatic vertex with a flow vertex will simply result in a new flow vertex, whereas connective two flow vertices will result in a cut vertex. Given that the axiom hyperedges are of type $\{s1\}$, these are the only possibilities.

Though the cut vertices don't present any problems for the proof net calculus, they can be eliminated without problem giving a shorter conversion sequence to the final tensor tree (Moot & Puite 2002). From the point of view of proof search, we therefore want to look at cut-free proof nets. Indeed, we can see that cut elimination is immediate for the cases where the axiom/cut rule produces an axiomatic or flow vertex, it corresponds simply to a composition of proofs similar to what we have in natural deduction.

Figure 3.5 shows an example of a proof structure corresponding to the sequent

$$n, (n\backslash n)/(s/\Diamond\Box np), np, (np\backslash s)/np \vdash n$$

on the left of the figure. The five axiomatic vertices in the proof structure have been circled. The only positive vertex, a premiss of the $[R/]$ link by the $t$ selector and a conclusion of the $[L/]$ link by the $s2$ selector has a grey color. All other vertices are negative flow vertices. Remark that I have rotated the $[R/]$ link, displaying the $s2$-selected vertex above the node label instead of below it, in spite of it being a conclusion of the link. Similarly, the $s2$-selected vertex of the topmost $[L/]$ link is portray below the node label, even though it is a premiss of its link.

## 3.2 Abstract Proof Structures

An abstract proof structure erases some of the distinctions made by proof structures. For example, the different tensor links with the same number of tentacles are no longer distinguished and the formula function is no longer defined for all vertices but only for the hypotheses and conclusions of the structure.

We keep the different par links, but because the tensor links are now indistinguishable, we lose nearly all information about axiom, flow and cut vertices.

**Definition 3.5** *An* abstract proof structure *is a tuple* $\langle H, V, frm, pre, con\rangle$ *such that* $H$ *is a hypergraph with selectors* $\{1, 2, 3\}$ *and edge labels*

| | |
|---|---|
| $h, c$ | *of type* $\{1\}$ |
| $\langle\rangle$ | *of type* $\{1, 2\}$ |
| $\circ$ | *of type* $\{1, 2, 3\}$ |
| $L\Diamond, R\Box$ | *of type* $\{1, 2\}$ |
| $R/, L\bullet, R\backslash$ | *of type* $\{1, 2, 3\}$ |

*such that every vertex is incident to exactly two hyperedges, once as a premiss and once as a conclusion. pre and con are the premiss and conclusion functions as before, but with the definitions for the edge labels of abstract proof structures as shown in Table 3.2.*

*The formula function frm is a partial function from* edges *to formulas.* $frm(e)$ *is defined iff* $e$ *is an* $h$ *or* $c$ *edge.*

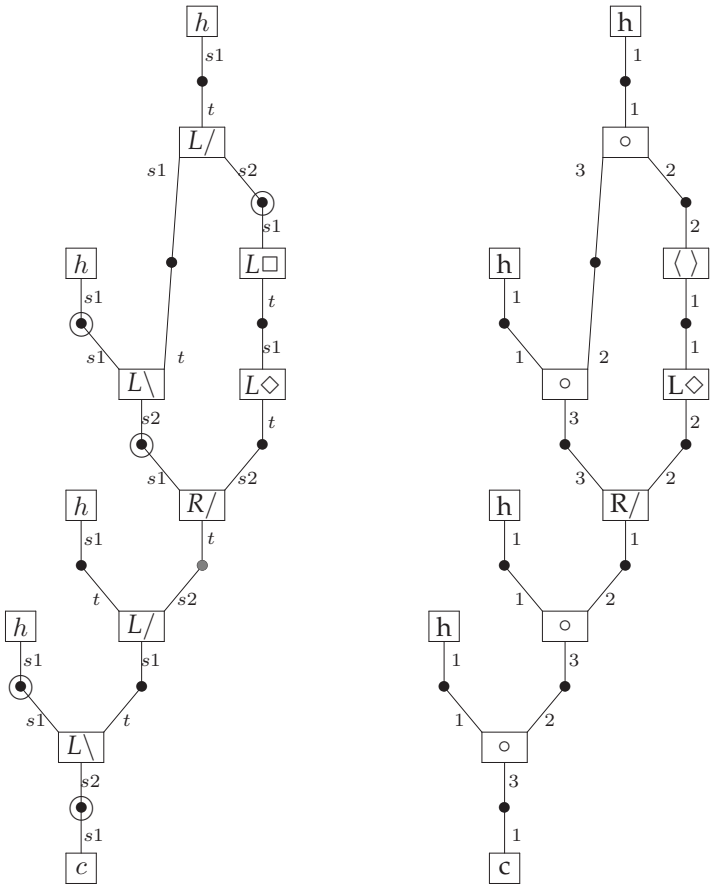



Figure 3.5: An example proof structure and the corresponding abstract proof structure

**Definition 3.6** *Let $\mathcal{P}$ be a proof structure. The abstract proof structure corresponding to $\mathcal{P}$, $\mathcal{A} = aps(\mathcal{P})$ is defined as follows.*

- $V_{\mathcal{A}} = V_{\mathcal{P}}$.
- $E_{\mathcal{A}} = tr(E_{\mathcal{P}})$, *where tr is the edge translation function shown in Figures 3.6 and 3.7.*
- $frm_{\mathcal{A}} = frm_{\mathcal{P}}(nod(e, s1))$ *for all* $e \in e_{\mathcal{P}}$ *such that* $lab(e) = h$ *or* $lab(e) = c$ *and undefined in all other cases.*

On the right of Figure 3.5 we see the conversion of the example proof structure into its abstract proof structure.

From the definition of the translation function it is clear that we can still recover *some* of the active and main vertices of an abstract proof structure: for

| | | | |
|---|---|---|---|
| $pre(\mathrm{h})$ | $=[]$ | $con(\mathrm{h})$ | $=[1]$ |
| $pre(\mathrm{c})$ | $=[1]$ | $con(\mathrm{c})$ | $=[]$ |
| $pre(\langle\ \rangle)$ | $=[1]$ | $con(\langle\ \rangle)$ | $=[2]$ |
| $pre(\circ)$ | $=[1,2]$ | $con(\circ)$ | $=[3]$ |
| $pre(\mathrm{L}\diamond)$ | $=[1]$ | $con(\mathrm{L}\diamond)$ | $=[2]$ |
| $pre(\mathrm{R}\Box)$ | $=[1]$ | $con(\mathrm{R}\Box)$ | $=[2]$ |
| $pre(\mathrm{R}/)$ | $=[3]$ | $con(\mathrm{R}/)$ | $=[1,2]$ |
| $pre(\mathrm{L}\bullet)$ | $=[3]$ | $con(\mathrm{L}\bullet)$ | $=[1,2]$ |
| $pre(\mathrm{R}\backslash)$ | $=[3]$ | $con(\mathrm{R}\backslash)$ | $=[1,2]$ |

Table 3.2: The functions *pre* and *con* for abstract proof structures

a R/ link, the main formula is the one incident to 1, for a R\ link it is incident to 2 and for a L• link it is incident to 3. Similary, the main formula of a R□ link is incident to 1 and the main formula of a L◇ link is incident to 2.

**Lemma 3.7** *If $\mathcal{P}$ is a proof structure then $\mathcal{A} = aps(\mathcal{P})$ is an abstract proof structure.*

**Proof** This is fairly trivial. It amounts to verifying is all cases that the incidence function reaches the same vertices with the renamed selector and that the hypotheses and conclusions of a hyperedge are the same list of vertices in each case. Since the translation is a simple renaming, this is trivial. The formula function *frm* has the required property of assigning a formula to all and only the h and c edges by construction. □

**Definition 3.8** *A* tensor tree *$\mathcal{T}$ is an abstract proof structure containing only edges with labels h, c, $\langle\rangle$ and $\circ$.*

## 3.3 Contractions and Structural Rules

For abstract proof structures, we define the contraction operation as follows. The configurations we contract are shown in Figure 3.8: in all cases a tensor link and a par link are connected to eachother by all tentacles of the two links which have the same selector except one and the difference between the par links is the tentacle chosen to be 'external', that is, pointing to the top and bottom vertices in the Figure.

**Definition 3.9** *If a hypergraph $H$ contains a subgraph of one of the forms shown in Figure 3.8, a* contraction in one step *is $H \to H'$ is obtained by deleting the two links as well as the vertices which are incident to both links, then identifing the two exterior vertices.*

*We will say $H \to^* H'$ or $H$* contracts to *$H'$ by taking the $\to^*$ to be the reflexitive, transitive closure of $\to$.*

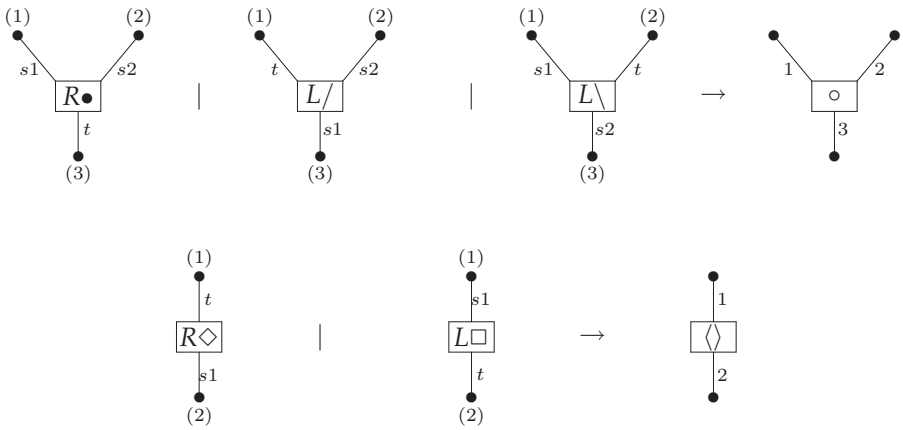



Figure 3.6: Converting a PS to an APS: tensor cases

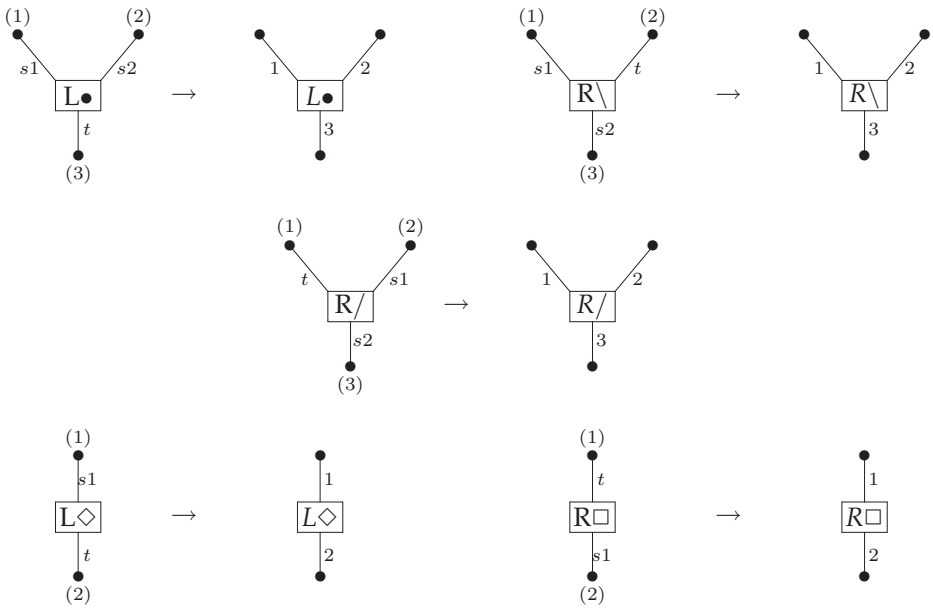



Figure 3.7: Converting a PS to an APS: par cases

**Lemma 3.10** *If $H \rightarrow H'$ then $H$ is an* ***NL***$\diamond$ *abstract proof structure iff $H'$ is an abstract proof structure.*

**Proof** Take $H = \langle E, v, \mathit{frm}, \mathit{pre}, \mathit{con} \rangle$ to be an abstract proof structure and $e_1$ and $e_2$ to be two hyperedges such that their neighbourhood is as shown in

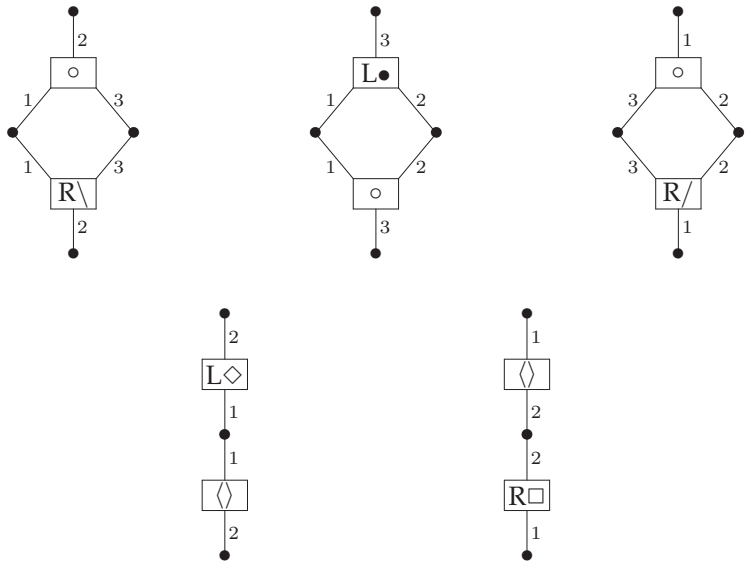



Figure 3.8: Contractions for **NL**◇ abstract proof structures

Figure 3.8, with $v_1$ and $v_2$ being the two internal vertices and $v_h$ the vertex which is a hypothesis of its link — and therefore the conclusion of another link — and $v_c$ the vertex which is a conclusion of its link — and therefore the hypothesis of another link. To obtain $H$ we delete hyperedges $e_1$ and $e_2$ and vertices $v_1$ and $v_2$. Furthermore we identify the two nodes $v_c$ and $v_h$, making it a vertex which is a conclusion of one link and a premiss of one link. Given that no hyperedges labeled $h$ or $c$ can disappear *frm* remains the same, whereas *pre* and *con* are the natural restriction of the same functions of $H$ to the hyperedges of $H'$. □

In addition to the contractions, which are a fixed component of all instantiations of **NL**$\diamond_{\mathcal{R}}$, a logic can specify a (finite and typically quite small) set of structural rules $\mathcal{R}$. These structural rules rewrite a tensor tree into another tensor tree, with the requirement that both tensor trees have the same set of distinct leaves, though we are allowed to change the order of the leaves. I will present some structural rules which have been used very frequently in the literature on multimodal categorial grammars (Moortgat & Oehrle 1993, Moortgat & Oehrle 1994, Moortgat 1997, Moortgat 1999, Vermaat 2005).

One of the first sets of structural rules used to extend the descriptive power of the non-associative Lambek calculus are the mixed associativity and mixed commutativity postulates. Figure 3.9 shows these postulates in their extraction formulation (the arrows pointing towards the central graph) as well as in their infixation formulation (the arrows pointing away from the central graph). In all cases, there is an interaction between a mode 0 and a mode 1 where the same

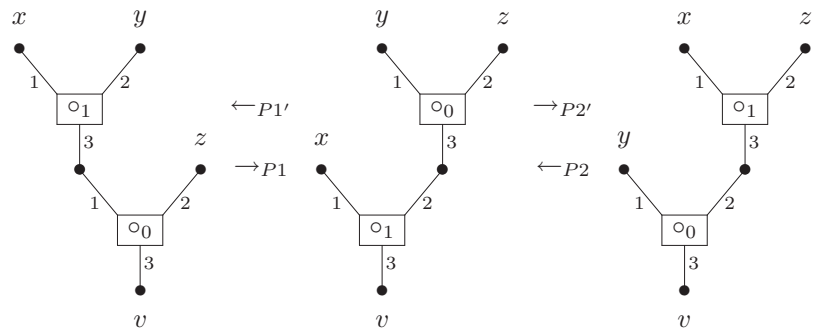



Figure 3.9: Mixed associativity and mixed commutativity — left branch

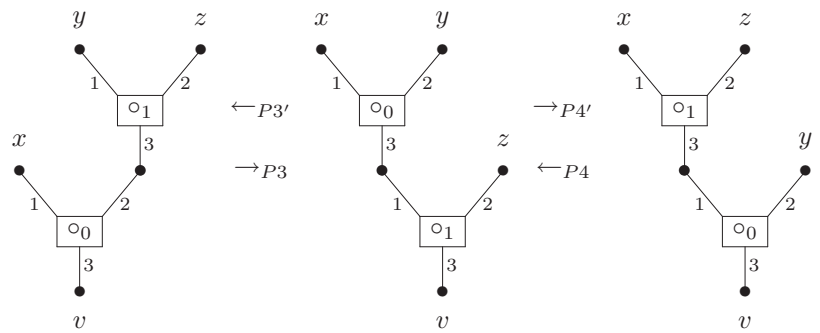



Figure 3.10: Mixed associativity and mixed commutativity — right branch

vertex — labeled $x$ in the figure — is the leftmost daughter of the hyperedge labeled $\circ_1$ on both sides of the rule.

Figure 3.10 shows the symmetric set of postulates. This time the vertex label by $z$ is the *right* daughter of the hyperedge labeled $\circ_1$ on both sides of the rule. Again there is the possibility of extraction (the arrows pointing towards the central graph) and infixation (the arrows pointing away from the central graph).

Another possibility for expressing controlled associativity and commutativty in a multimodal calculus is by using unary control. This time we have just a single binary mode interacting with a single unary mode. The unary mode marks the element which licenses the restructuring operations. In Figure 3.11 this is the element $x$ (compare this figure with the version of these structural rules without unary control in Figure 3.9). There is again the possibility to specify these structural rules in the form of infixation and in the form of extraction rules.

Figure 3.12 displays a version of the right branch extraction and infixation rules of Figure 3.10 with unary control. In both cases the $z$ node moves from one right branch to another.

**Definition 3.11** *Let $H$ be a hypergraph which is an abstract proof structure for some logic* ***NL***$\diamond_{\mathcal{R}}$ *with a set of structural rules $\mathcal{R}$, a* conversion in one step *is $H \rightarrow_{\mathcal{R}} H'$*

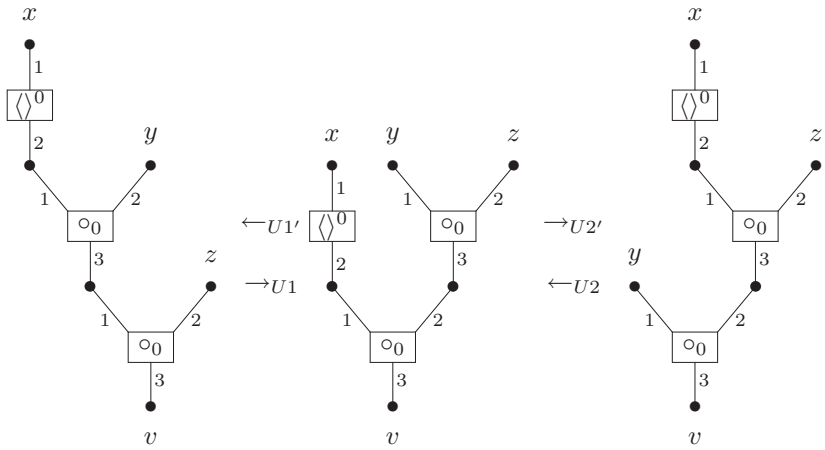



Figure 3.11: Mixed associativity and mixed commutativity — left branch with unary control

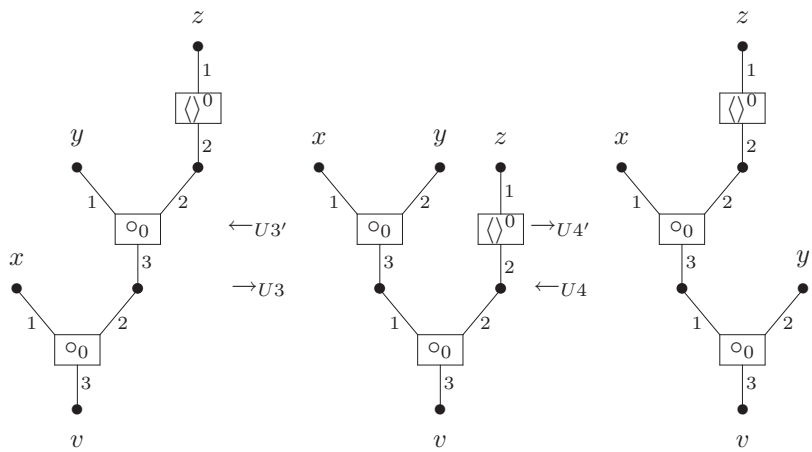



Figure 3.12: Mixed associativity and mixed commutativity — right branch with unary control

*is obtained by either:*

- *performing a contraction,*
- *given a rule $r \in \mathcal{R}$ which is of the form $\mathcal{T} \to \mathcal{T}'$ such that $\mathcal{T}$ is a subgraph of $H$ and $H'$ is $H$ with the subgraph $\mathcal{T}$ of $H$ replaced by $\mathcal{T}'$.*

*We will say $H \to^*_{\mathcal{R}} H'$ or $H$* converts to *$H'$ by taking $\to^*_{\mathcal{R}}$ to be the reflexitive, transitive closure of $\to_{\mathcal{R}}$.*

**Lemma 3.12** *If $H \to_{\mathcal{R}} H'$ then $H$ is an $\mathbf{NL}\Diamond_{\mathcal{R}}$ abstract proof structure iff $H'$ is an abstract proof structure.*

**Proof** This is a trivial extension of the previous lemma. For every structural rule, the internal nodes are a premiss and a conclusion of a link, as required. For the external nodes: the leaves which are premisses of the links of the structural rule on both the left hand side and the right hand side of the rule and therefore all the conclusion of some other link. Given structural rules preserve the leaves, the situation doesn't change from left hand side to the right hand side of a rule of vice verse. For the conclusion, we can make the same argument. It is a conclusion of a link in the structural rule and therefore a hypothesis of another link in the abstract proof structure before a structural conversion iff it is the hypothesis of this same link after the structural conversion. □

## 3.4 The Lambek-Grishin Calculus

The Lambek-Grishin calculus LG was introduced by Grishin (1983). It adds symmetric connectives to the Lambek calculus: reversing the left hand and right hand side of the turnstile, or, in our proof net case, adding up-down symmetric connectives, which turn each premiss into a conclusion and each conclusion into a premiss. In addition, Grishin explores the possible interaction principles between the connectives.

Bernardi & Moortgat (2007) explore the possiblities of using the Lambek-Grishin calculus, giving an analysis of quantifier scope using Grishin's interaction principles and an interpretation of the scope possiblities in terms of continuation semantics.

Figure 3.13 shows the contractions for proof nets in the Lambek-Grishin calculus. The contractions for Lambek connectives [R\], [L•] and [R/] are the same as before, but note how the Grishin connectives [L⊘], [L⊙] and [R⊘] are just the up-down symmetric versions of the Lambek connectives, given that ';' is the inverse of '∘'.

Grishin has proposed a number of interaction principles for LG divided into four classes. Class I and class IV will interest us here since they concern the interaction between a Lambek connector '∘' and an inverse Grishin connector ';'.

Figures 3.14 and 3.15 show all structural rules in Grishin classes I and IV. The inward pointing arrows, with the primed rule names, correspond to the structural rules of class I, whereas the outward pointing arrows correspond to the structural rules of class IV.

Let me take a moment to point out the similarity between Figure 3.14 and Figure 3.9 and between Figure 3.15 and Figure 3.10. Apart from the fact that $v$ is a conclusion of the rule in Figure 3.14 and that $x$ is a hypothesis of the rule in Figure 3.9 the two figures display the same configuration up to a relabeling of the selectors and hyperedges. The easiest way to see this is to rotate the ';' hyperedge to the right in such a way that $v$ is displayed at the top of the hyperedge.

Similarly, in Figure 3.15, when we abstract away of the fact that one vertex

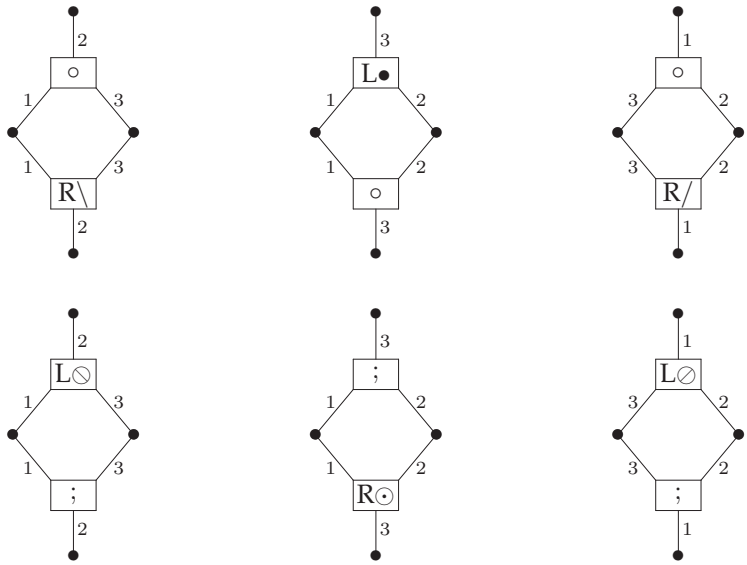



Figure 3.13: Contractions for LG abstract proof structures

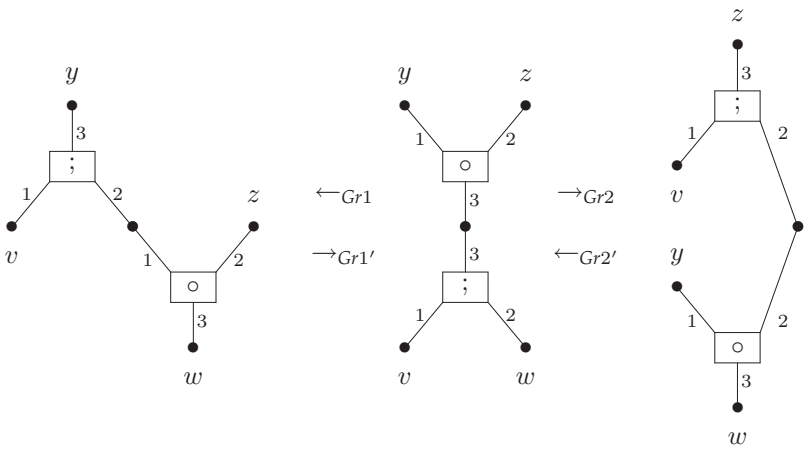



Figure 3.14: Mixed associativity and mixed commutativity — Lambek-Grishin, left branch

is a hypothesis and the other a conclusion, $w$ plays the role of $z$ in Figure 3.10 and a rotation of the ';' hyperedge — to the left this time — again gives an isomorphism of the graphs up to a relabeling of the hyperedges and selectors labels.

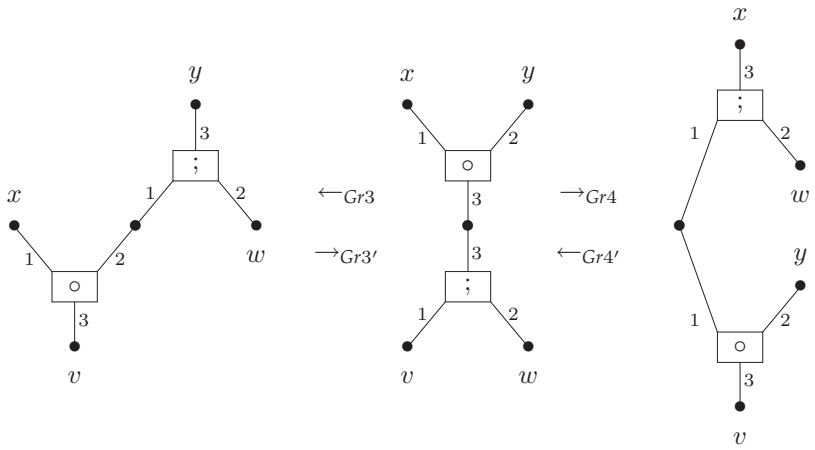



Figure 3.15: Mixed associativity and mixed commutativity — Lambek-Grishin, right branch

## 3.5 Proof Nets

We are now in a position to define proof nets. For **NL**$\diamond$ a proof net is simply a proof structure where we can convert its abstract proof structure to a tensor tree, that is to say we eliminate all par rules (labeled [R\], [L•], [R/], [R□] and [L$\diamond$]) and the resulting structure is a tree with the leaves labeled by $h$ edges and the root by a $c$ edge.

**Theorem 3.13 (Moot & Puite)** *An proof structure $S$ for **NL**$\diamond$ is a* proof net *if and only if its underlying abstract proof structure contracts to a tensor tree.*

The result extends naturally when we add any set $\mathcal{R}$ of structural rules to **NL**$\diamond$ to obtain **NL**$\diamond_{\mathcal{R}}$. A structural rule in this context is a rewrite of one tensor tree with $n$ distinct hypotheses as its leaves into another tensor tree with the same $n$ distinct leaves, though not necessarily in the same order.

**Theorem 3.14 (Moot & Puite)** *An proof structure $S$ for **NL**$\diamond_{\mathcal{R}}$ is a* proof net *if and only if its underlying abstract proof structure converts to a tensor tree using the contractions and the structural rules in $\mathcal{R}$.*

Finally, moving to LG not a lot changes. The only change is that interaction principles proposed by Grishin are defined on structures having two conclusions in addition to two hypotheses.

**Theorem 3.15 (Moot)** *An proof structure $S$ for LG is a* proof net *if and only if its underlying abstract proof structure converts to a tensor tree using the contractions and any subset of the Grishin interaction principles.*

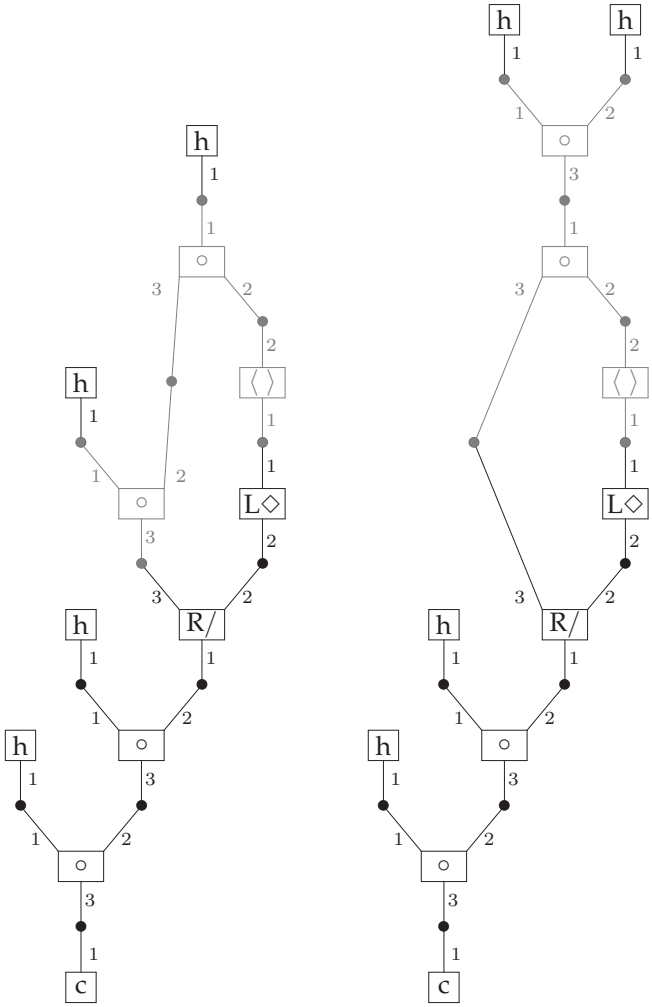



Figure 3.16: Applying a structural conversion to the abstract proof structure of Figure 3.5.

**Example 3.16** *Everything is now in place to show that the proof structure shown in Figure 3.5 is a proof net, given that we have the structural rules for right branch extraction with unary control shown in Figure 3.12.*

*Figure 3.16 shows the abstract proof structure of Figure 3.5 on the left. There are two possibilities here: either we perform the* [L◇] *contraction — in which case we have a dead end where neither structural rules nor contractions apply — or we perform the single structural rule which is possible in this configuration, moving the unary branch out as shown on the right of Figure 3.16.*

*To convince yourself that the part of Figure 3.16 marked in gray is just Figure 3.12,*

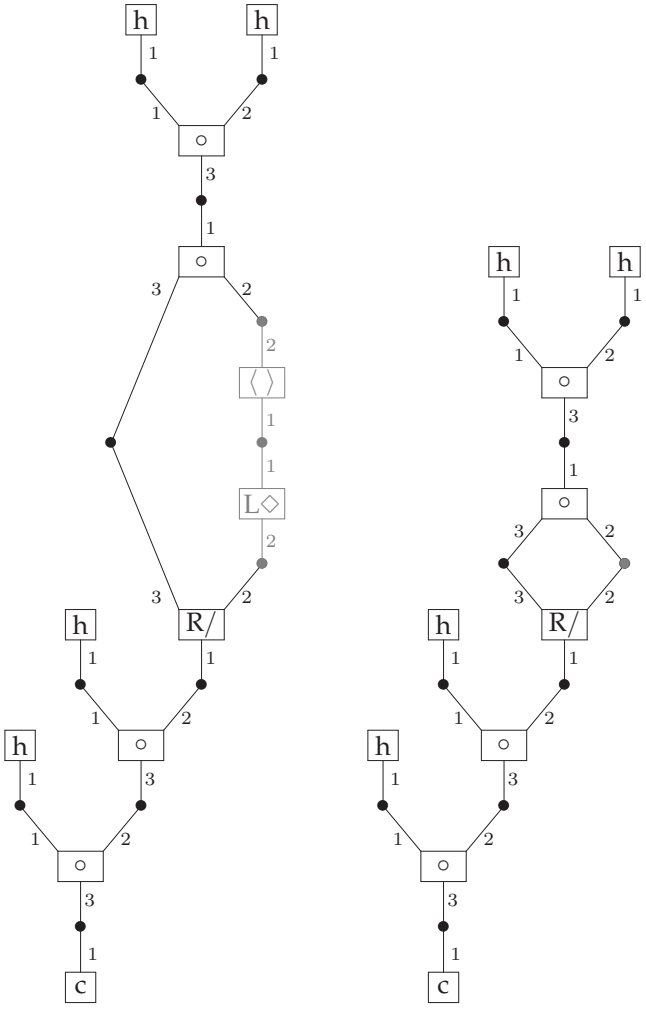



Figure 3.17: Applying an $L\diamond$ contraction to the abstract proof structure of Figure 3.16.

*remark that only the selector labels and the nodes connected to the rest of the graph are important. Figure 3.16 has turned the selector 2 of the top '∘' node has downward and has inversed the '⟨⟩' node, but the gray subgraph on the left is still isomorphic to the graph on the left of Figure 3.12 wheres the gray subgraph on the right is isomorphic to the graph in the middle of Figure 3.12.*

*Figure 3.17 shows repeats the result of the structural conversion at the left. Now the are no other possibilities for applying structural rules since only the extraction structural rules are available. In terms of the contractions, the redex for the* $[L\diamond]$ *contraction is marked in gray on the left of the figure and it contracts to the single*

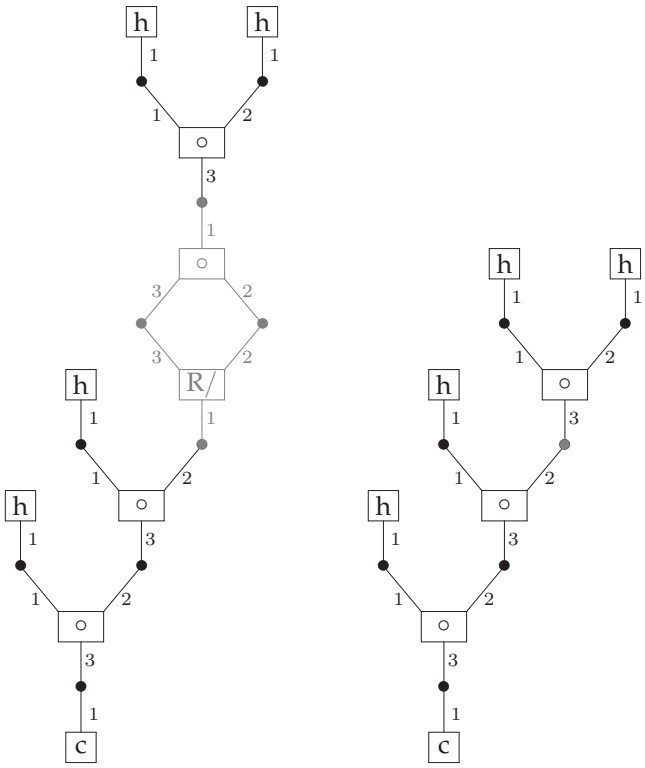



Figure 3.18: Applying an $R/$ contraction to the abstract proof structure of Figure 3.17.

*vertex shown in gray on the right of the figure.*

*Finally, we have the abstract proof structure shown in Figure 3.18 on the right. As shown in gray, we are in the right configuration to apply the final* $[R/]$ *contraction, giving us the tensor tree shown on the right.*

## 3.6 Type-logical Grammars

We now have everything in place to define type-logical grammars and the tree and string languages generated by them. In essence, as shown by the following definition, it corresponds to the simple addition of a lexicon to either an instance of $\mathbf{NL}\Diamond_{\mathcal{R}}$ or an instance of LG.

**Definition 3.17** *A* type-logical grammar *$G$ is a tuple $\langle \Sigma, A, C, lex, \mathcal{R}, S\rangle$ such that.*

$\Sigma$ *is the alphabet of non-terminal symbols,*

$A$ *is a finite set of atomic formulas,*

*$C$ is a set of logical connectives each with an arity,*

*lex is a function from $\Sigma$ to $\wp(F)$, that is from non-terminals to sets of formulas. We will assume without loss of generality that lex never assigns the empty set to a member of $\Sigma$.*

*$\mathcal{R}$ is a finite set of structural rules.*

*$S \in F$ is a special formula, call the start formula or the goal formula.*

The definition above is deliberately vague about the connectives. Moot (2007) lists the possibilities, but for the moment we will only consider the following two instantiations.

- the multimodal categorial grammar vocabulary, consisting of binary connectives $\{/_i, \bullet_i, \backslash_i\}$ for members $i$ of a finite set of binary modes $I$ together with unary connectives $\{\Diamond_j, \Box_j\}$ for members $j$ of a finite set of unary modes $J$.
- the Lambek-Grishin vocabulary consisting of the Lambek connectives $\{/, \bullet, \backslash\}$ and the Grishin connectives $\{\oslash, \odot, \obslash\}$.

For the structural rules $\mathcal{R}$, I will only consider subsets of the rules discussed in Section 3.3 for the multimodal vocabulary and I will only consider subsets of the structural rules dicussed in Section 3.4 for the Lambek-Grishin calculus.

**Example 3.18** *Given a single unary and a single binary mode, the set of structural rules $U3$ and $U4$ for their combination, the lexical assignments shown in Figure 3.4 and goal formula $n$ we have a (minimal) type-logical grammar $G_{wh}$ for* wh *extraction in English.*

**Definition 3.19** *Given a type-logical grammar $G$ and a sequence of words $w_1, \ldots, w_n$, such that $w_i \in \Sigma$ and $n > 0$, we say that $G \rightarrow w_1, \ldots, w_n$ iff $A_1, \ldots, A_n \vdash S$ where for each $i$, $A_i \in lex(w_i)$.*

**Definition 3.20** *Let $G$ be a type-logical grammar. The* string language *generated by $G$ is the set of sequences of words such that $G \rightarrow w_1, \ldots, w_n$. The* tree language *generated by $G$ is the set of tensor trees $T$ such that for each $t \in T$, $G \vdash t$.*

## 3.7 Analysis of the Structural Rules

The structural rules allow us to rewrite a tensor tree into a tensor tree with the same external nodes. Even with the condition posed in (Moot 2002) — that the number of unary tensor links on the right hand side of a tensor rules is equal to or less than the number of unary tensr links on the left hand side of a rule — this gives us extactly the context-sensitive languages and a PSPACE decision problem.

As we have seen in Section 2.3, hyperedge replacement grammars are *context free* graph grammars (Engelfriet 1997): while the right hand side of a rule can be any complex graph of the right type, the left hand side is always a single nonterminal hyperedge. The structural rules, on the other hand, have (typically non-trivial) trees on the left hand side of the rule. In this section we will investigate some very frequently used structural rules and ways to characterize the proof nets generated by using them in a context-free way.

Now, let's analyse the combinatorial possibilities of the combinations of the structural rules of Sections 3.3 and 3.4. Some combinations are know to have undesirable side-effects, such as collapse to an associative, commutative system but we will eliminate these combinations after discussing the total combinatorics. We will end up with a number of restrictions on the final grammar permitting us to describe the generated proof nets as a hyperedge replacement grammar while factoring out the structural rules.

The basic assumption we make in this analysis is that the final tensor trees in the grammar have a certain form: in particular, for the multimodal structural rules of Figures 3.9 and 3.10 we assume that mode 1 is grammar-internal and that no hyperedges labeled $\circ_1$ occur in the final tensor tree. All ternary hyperedges in the final tensor tree are labeled $\circ_0$ using the external mode 0 (we can extend this by adding other external modes without problem, requiring only that such additional modes do not interact with either mode 0 or mode 1).

We deploy a similar restriction to the structural rules with unary control, requiring that all unary branches are eliminated from the final tree and to the structural rules of the Lambek-Grishin calculus, requiring that all Grishin connectives ; are eleminated.

**Definition 3.21** *Given a structural rule $r$ with connectives $c_1, \ldots, c_n$ on the left hand side and connectives $d_1, \ldots, d_n$ on the right hand side a* trace *is a bijection associating each element $c_i$ with an element of $d_j$.*

**Definition 3.22** *Let $\mathcal{P}$ be a proof structure, $\mathcal{A}$ its corresponding abstract proof structure and $\rho$ be a conversion sequence coverting $\mathcal{A}$. For a hyperedge $e$ of $\mathcal{P}$ the* trace *of $e$ in $\rho$ is defined as the prefix of $\rho$ such that:*

- *...*

**Definition 3.23** *Let $\mathcal{P}$ be a proof net. It is* well-bracketed *iff there are no two contractions $c_1$ and $c_2$ in the conversion sequence of $\mathcal{A}$ such that such that on any of the paths between the two traces $t_{1a}$ and $t_{1b}$ of $c_1$ there is just one of the traces of $c_2$.*

Intuitively, the combination of well-bracketedness and traces guarantee that every path in a proof net can be described by a context free grammar, in much the same way as this is possible for several mildly context-sensitive grammar formalisms (Vijay-Shanker, Weir & Joshi 1987).

**Lemma 3.24** *Any **NL**$\diamond$ proof net is well-bracketed.*

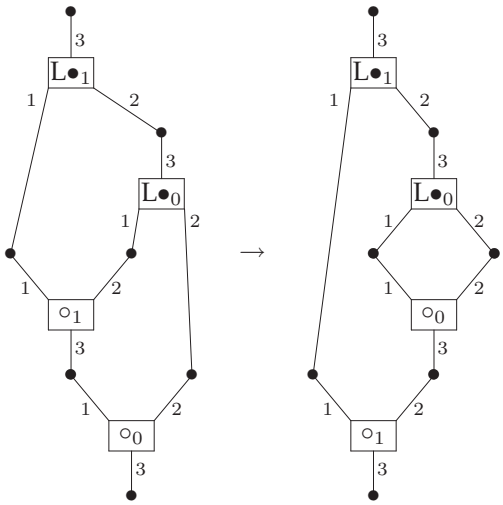



Figure 3.19: A proof net violating the well-bracketing conditions

**Proof** Looking backwards at a conversion sequence ending in a tensor tree, we see that every contraction corresponds to expanding a single vertex into two adjacent hyperedges with the empty path between their traces. All we need to verify is that whenever we apply such an expension the (output) result abstract proof structure is well-bracketed whenever the input was.

Suppose we have such a well-bracketed input and we apply an expansion. This operation has the effect that all paths passing through the vertex of the input which is expanded will be extended in the output graph, but extended in such a way all paths necessarily pass through both traces of the new hyperedges. □

Unfortunately, Lemma 3.24 doesn't extend in general to logics with structural rules. Figure 3.19 shows a non-wellbracketed proof net on the left, which we can contract after application of a mixed commutativity rule as shown on the right of the figure.

The configuration shown in Figure 3.19 can easily be adapted to the other binary par links and to the other sets of mixed associativity and mixed commutativity structural rules: those with unary control and the Lambek-Grishin interactions.

In order to provide conditions under which well-bracketing of the contractions can be guaranteed, I will introduce the notion of an inert mode.

**Definition 3.25** *Given a grammar $G$ with a set of structural rules $\mathcal{R}$, a mode $m$ is* inert *with respect to this grammar and these structural rules if we can appy all its contraction before applying any structural rules.*

In other words, a mode is inert

**Definition 3.26** *Given a grammar $G$ with a set of modes $M$ a grammar can select a subset $E$ of $M$ the member of which are called the* external *modes. All other modes are called* internal.

The distinction between external and internal modes is implicit in the work of many authors. A typical use of grammar-internal modes is feature-checking, where a unary branch of a certain mode — for example $n$ for nominative case — is introduced.

**Proposition 3.27** *Let $\mathcal{P}$ be a proof net. The internal modes of $\mathcal{P}$ occur only in pairs of par and tensor links.*

**Proof** This is a direct consequence of the fact the internal modes are not allowed to occur on the final tensor tree and that contractions eliminate a tensor and a par link at the same time. □

As already suggested by the description I gave of the structural rules in

**Lemma 3.28** *An* ***NL***$\diamond_{\mathcal{R}}$ *or LG grammar $G$ is well-bracketed if*

- *all its external modes are inert,*
- *the structural rules are a subset of one of the following sets*
  - $\{P1, P2, P3, P4\}$
  - $\{P1', P2'\}$
  - $\{P3', P4'\}$
  - $\{Gr1, Gr2, Gr3, Gr4\}$
  - $\{Gr1', Gr2'\}$
  - $\{Gr3', Gr4'\}$

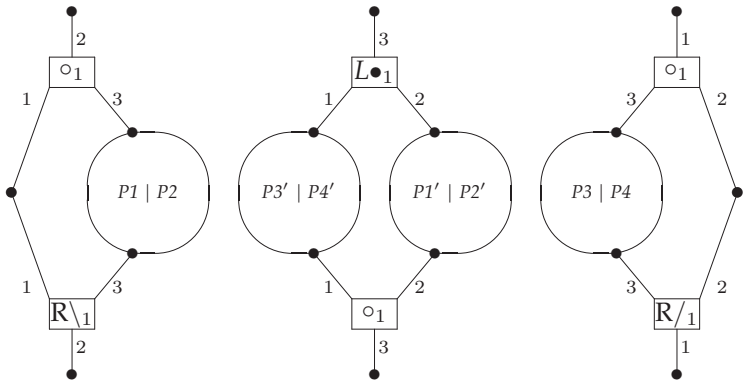



Figure 3.20: Generalised contractions for $\mathbf{NL}\Diamond_{\mathcal{R}}$ abstract proof structures

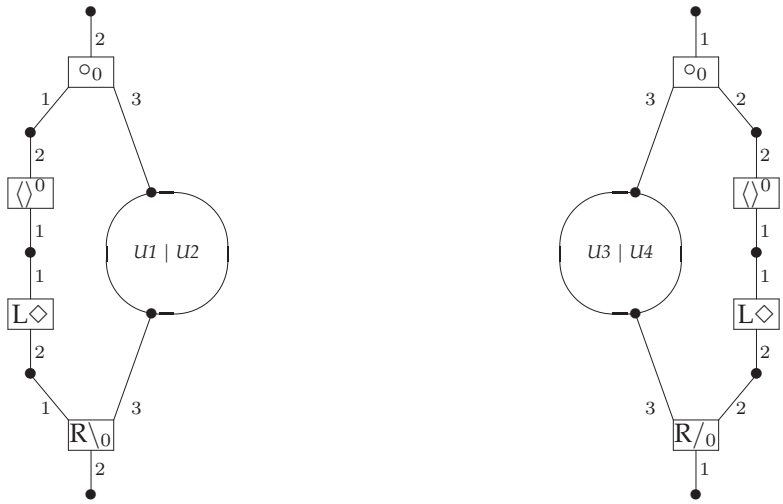



Figure 3.21: Generalised contractions for $\mathbf{NL}\Diamond_{\mathcal{R}}$ abstract proof structures with unary control

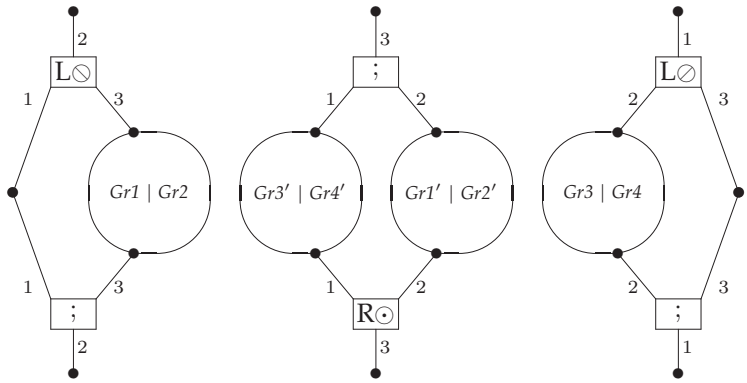



Figure 3.22: Generalised contractions for LG abstract proof structures

# Chapter 4

# Type-Logical Grammars as HR Grammars

Given that the data structures used for proof structures and abstract proof structures are both hypergraphs, it makes sense to try to provide hyperedge replacement grammars for both. I will do this in steps, adding more details at each step until the at final result we have a hyperedge replacement grammar for type-logical grammars.

As a first step, Section 4.1 presents a hyperedge replacement grammar for abstract proof nets, that is, abstract proof structures which contract to a tensor tree. The second step adds the rules fof converting an abstract proof structure to a proof structures, which are exactly the inverse rules of Section 3.2. Then we add additional information keeping track of the axiomatic, flow and cut formulas in the proof net, which will allow us to tell where we have to split up the proof net. Finally, special hyperedge replacement rules for different sets of structural rules will be discussed.

The correspondence is very strong: except for the structural rules, which are 'compiled away' we will generate the same sets of hypergraphs using the hyperedge replacement grammars as we doing generating the proof nets.

## 4.1 A Hyperedge Replacement Grammar for Abstract Proof Nets

The hyperedge replacement grammar *HR*(*APN*) is defined as follows.

$$HR(APN) = (\{S, T, V\}, \{h, c, \circ, \langle\rangle, /, \bullet, \backslash, \diamond, \square\}, \{s1, s2, t, 1, 2, 3\}, P, S)$$

The set of productions $P$ is shown in Figures 4.1, 4.2 and 4.3.

Figure 4.1 shows the start configuration. It rewrites the $S$ nonterminal of type $\emptyset$ to a tree $T$ of type $\{1, 2\}$ with the $1$ tentacle connected to a hypothesis

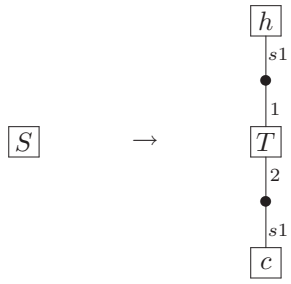



Figure 4.1: Abstract proof nets: start rule

vertex (ie. a vertex which is connected to the single $s1$ tentacle of an $h$ hyperedge) and the 2 tentacle connected to the conclusion vertex (ie. a vertex which is connected to the single $s1$ tentacle of an $c$ hyperedge).

The intuition behind the $T$ rules ($T$ for 'tensor tree') shown in Figure 4.2 is that we construct a tensor tree with binary und unary branches (we obtain the tree by interpreting 1 as indicating the left daugter, 2 as indicating the right daughter and 3 as indicating the parent node for binary branches and by interpreting 1 as indicating the single daughter and 2 the parent node for unary branches.

The tree is constructed top-down, that is to say the nonterminal hyperedge connected to the root (conclusion) external node — the external node (2) — is an abstract proof structure contracting to a single vertex, indicated by the $V$ nonterminal. This way, the $T$ rules construct a tree of tensor links where every vertex in the tensor tree is represented by a single $V$ nonterminal.

The $V$ rules are shown in Figure 4.3: again we descend the graph top-down. In order for an abstract proof net to contract to a single vertex, we need to be able to find a 1-1 correspondance between tensor links and par links. Figure 4.3 simply states that an abstract proof structure contracts to a single vertex whenever the two active vertices of a tensor and a par link are connected after contracting any intervening structure, as indicated by the $V$ hyperedges between the active vertices in all grammar rules. Finally, we can have a graph contracting to a single vertex which is followed by another sequence to be contracted (remember that the contraction of the first hyperedge doesn't necessarily occur with the last hyperedge).

The final case contracts the $V$ hyperedge to a single vertex, identifying the two distinct nodes it connected before.

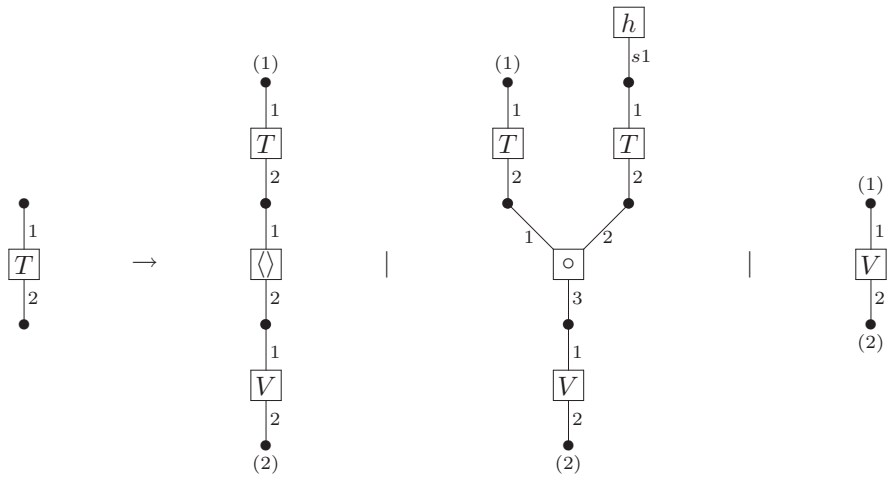



Figure 4.2: Abstract proof nets: tensor rules

## 4.2 A Hyperedge Replacement Grammar for Proof Nets

The hyperedge replacement grammar $HR(PN)$ is defined as follows.

$$HR(PN) = (\{S, T, V, \circ, \langle\rangle, /, \bullet, \backslash, \diamond, \square\},$$
$$\{h, c, L/, R/, L\bullet, R\bullet, L\backslash, R\backslash, L\diamond, R\diamond, L\square, R\square\}, \{s1, s2, t\}, P, S)$$

where the set of productions $P$ contains — in addition to all the productions of $HR(APN)$ — the following additional productions.

Remember that proof nets are relatively close to abstract proof nets. We have to distinguish the different possible instantiations of the '$\circ$' and the '$\langle\rangle$' tensor links in the abstract proof structures — as shown in Figure 4.4 — but for the par links this is just a simple matter of relabeling the hyperedges and tentacles.

## 4.3 Soundness and Completeness

With everything in place it is time to prove the main theorem, that the proof nets generated by the hyperedge replacement grammar of the previous section

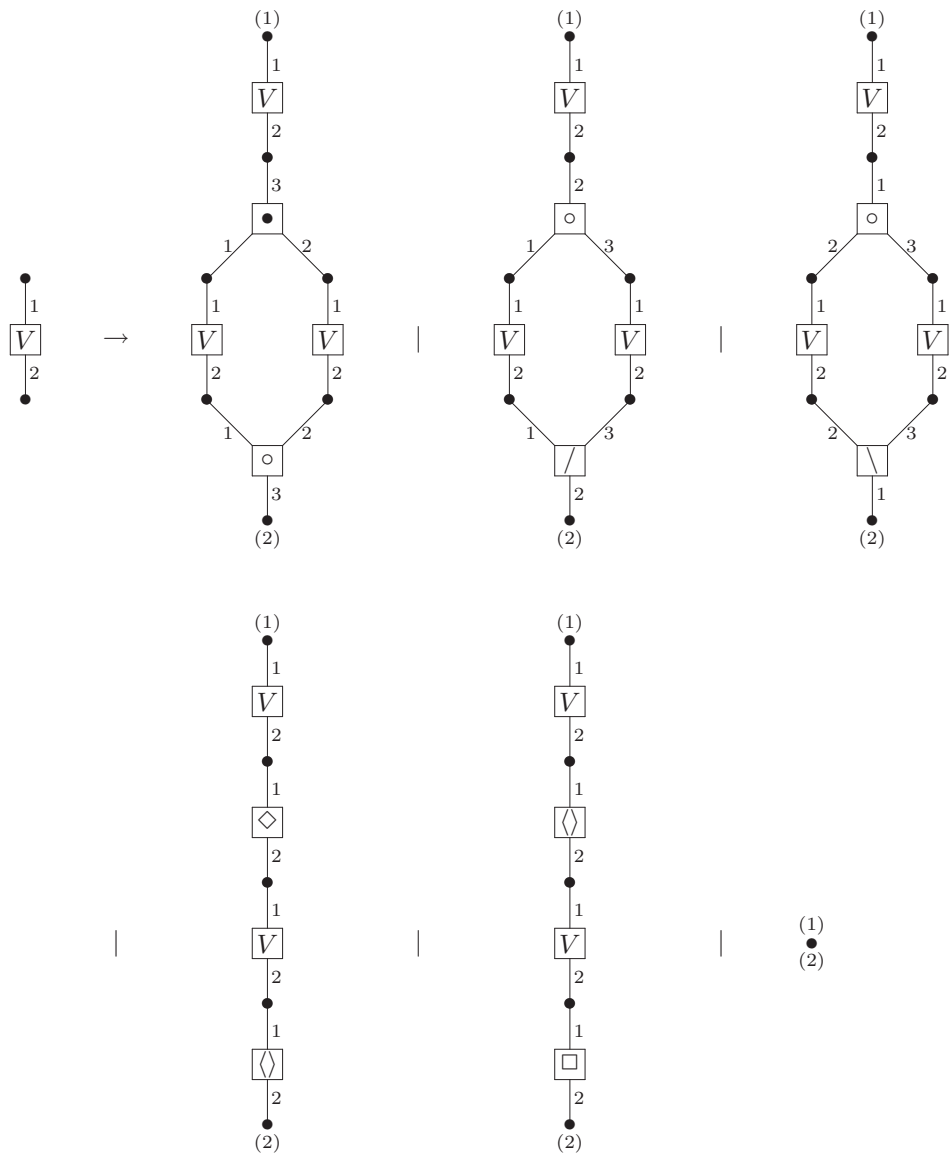


Figure 4.3: Abstract proof nest: par rules

and the proof nets of Section 3 are the same. To facilitate this proof, I will first prove two normal form lemma for the hyperedge replacement grammars.

**Definition 4.1** *A HR(APN) derivation is in* normal form *iff it is of the following*

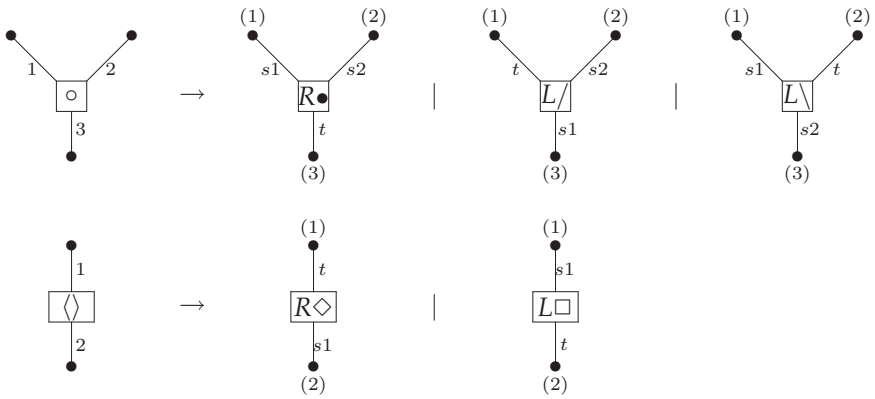



Figure 4.4: Converting tensor links from abstract proof nets to proof nets

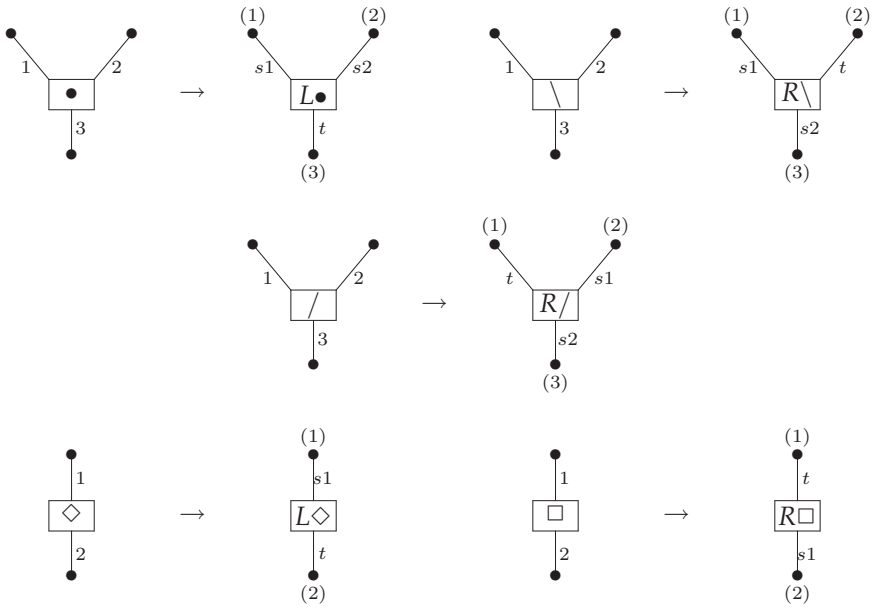



Figure 4.5: Converting par links from abstract proof nets to proof nets

*form*

$$t_0 \to \ldots \to t_i \to v_0 \to \ldots \to v_j \to v'_0 \to \ldots \to v'_k$$

*where each $t_n$ rewrites a $T$ nonterminal, each $v_n$ expands a $V$ nonterminal, i.e. applies on of the $V$ rules except the last one which contracts the $V$ edge, and finally each $v'_n$ corresponds to a contraction of a $V$ edge.*

**Lemma 4.2** *For every HR(APN) derivation $d$ there is a derivation $d'$ ending in the same graph which is in normal form.*

**Definition 4.3** *A HR(PN) derivation is in* normal form *iff it is of the following form*

$$a_0 \rightarrow \ldots \rightarrow a_i \rightarrow p_0 \rightarrow \ldots \rightarrow p_j$$

*where each the sequence $a_0, \ldots a_i$ is an HR(APN) derivation in normal form and the sequence $p_0 \rightarrow \ldots \rightarrow p_j$ contains only HR(PN) rules not in HR(APN).*

**Lemma 4.4** *For every HR(PN) derivation $d$ there is a derivation $d'$ ending in the same graph which is in normal form.*

**Theorem 4.5** *An proof structure $S$ for* **NL**$\diamond$ *is a* proof net *if an only if it is generated by HR(PN).*

***PN*** $\rightarrow$ ***HR*(*PN*)** Given that $S$ is a proof net, we know that there is a conversion sequence $\rho$ which converts the abstract proof structure $A$ of $S$ to a tensor tree $T$. We proceed by induction on the length $l$ of $\rho$. The induction hypothesis is that for every abstract proof structure in the conversion sequence we have a $V$ edge for every vertex in the abstract proof structure which can expand into a pair of links. Starting from the tensor tree at the end of the conversion sequence we move to the beginning of the conversion sequence generating the initial abstract proof net then finally we convert it to the proof net.

In case $l = 0$, we know that $A$ is a tensor tree. We apply the $T$ rules to obtain the same tensor tree, but where every vertex in this tensor tree is now a $V$ edge.

In case $l > 0$, we know by the induction hypothesis that the abstract proof structure has a $V$ edge for each of its vertices. The contraction expands a vertex into two edges and we apply the corresponding HR rule to add the two links to the abstract proof structure.

After all contractions have been performed, we have obtained an abstract proof net with some extra $V$ edges. We replace these edges with single nodes to obtain an *HR*(*APN*) derivation producing a hypergraph which is isomorphic to the original APN. Finally, using the links of the proof net we extend this *HR*(*APN*) derivation to a *HR*(*PN*) derivation which is isomorphic to the original proof net.

***HR*(*PN*)** $\rightarrow$ ***PN*** Suppose that $d$ is a *HR*(*PN*) derivation in normal form. Let $S$ be the proof structure at the end of the conversion sequence and let $A$ be the abstract proof structure at the end of the subsequence $d'$ of $d$ which is a *HR*(*APN*) derivation. In order to show that $S$ is a proof net we have to prove that $A$ contracts to a tensor tree, which we do by induction on the number of expansions $e$ in $d'$.

Suppose $d'$ contains no expansions, then $d'$ ends in a tensor tree we have a proof net.

If $d'$ does contain expansions, we look at the last one. Given that there are no further expansions, all $V$ nodes in the rule are converted to vertices to abtain

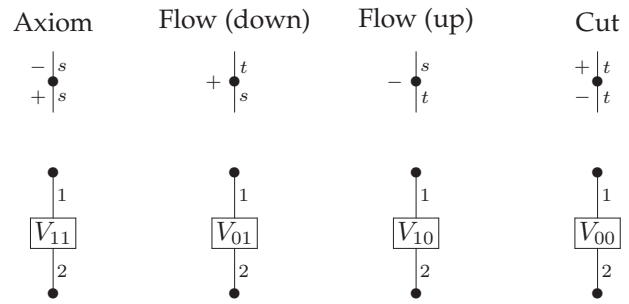



Figure 4.6: Cut, flow and axiom vertices and the corresponding hyperedges

the abstract proof structure and therefore the abstract proof structure is of the correct form to apply the contraction. □

## 4.4 A Hyperedge Replacement Grammar for Type-Logical Grammars

The main theorem states that proof nets generated by hyperedge replacement and proof nets conforming to the contraction criterion coincide for **NL**$\diamond$. However, we haven't yet taken the axiom connections and the structural rules into account.

In order to add the axiom connections to the hyperedge replacement grammar of the previous sections, we are confronted with the problem that in order to distinguish between cut, flow and axiom vertices we need to know the labels of the two tentacles adjacent to the vertex and the hyperedge replacement rules can distinguish only between *hyperedge labels*, we cannot rewrite a vertex based on its incident labels. The solution is very simple: since the $V$ hyperedges already correspond to the vertices in the hypergraph of a proof structure, we simply need to distinguish the four different cases (cut, flow up, flow down and axiom) in the hyperedge label. So the hyperedge label $V$ is replaced by a hyperedge label $V_{ij}$ where $i, j \in \{0, 1\}$. $i$ is 0 in case the vertex is the *hypothesis* of a link by means of its $t$ tentacle and 1 in case it is its hypothesis by means of an $s$ tentacle. Similarly, $j$ is 0 in case it is the *conclusion* of a link by means of its $t$ tentacle and 1 in case it is the conclusion by means of an $s$ tentacle. Figure 4.6 repeats Figure 3.2 while adding the different corresponding $V$ hyperedges.

The single $T$ hyperedge labels is likewise replaced by four instances $T_{11}$ (axiom), $T_{01}$ (flow down), $T_{10}$ (flow up) and $T_{00}$ (cut).

However, given that these vertex types depend crucially on the instantiations of the $\circ$, $\langle\rangle$ and ; labels, we need to compile out the different possibilities. This multiplication results in a hyperedge replacement grammar with a bigger number of rules, though it reduces a hyperedge replacement grammar of or-

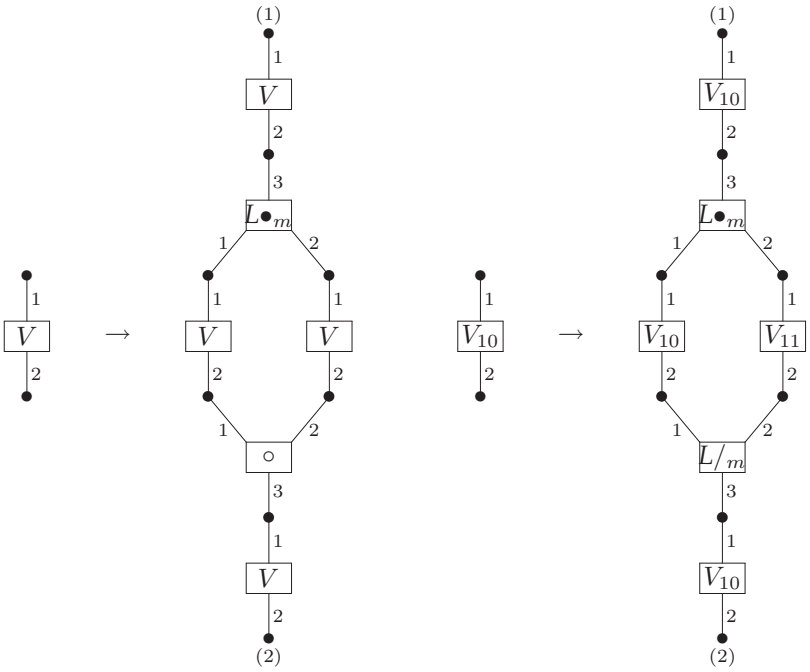



Figure 4.7: Hyperedge replacement rules with and without separating the different $V$ edges

der three to a hyperedge replacement grammar of order two. Figure 4.7 shows how this addition changes the hyperedge replacement rules.

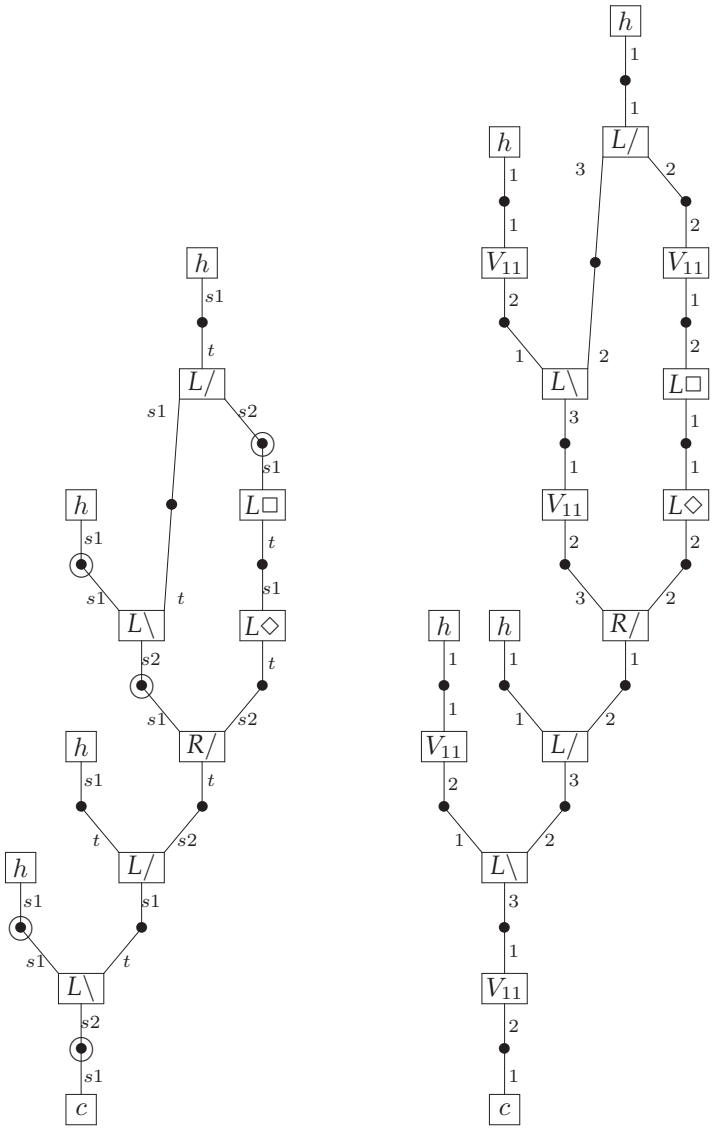



Figure 4.8: The example structure of Figure 3.5 with the axiom vertices encircled (left) and with explicit axiom hyperedges (right)

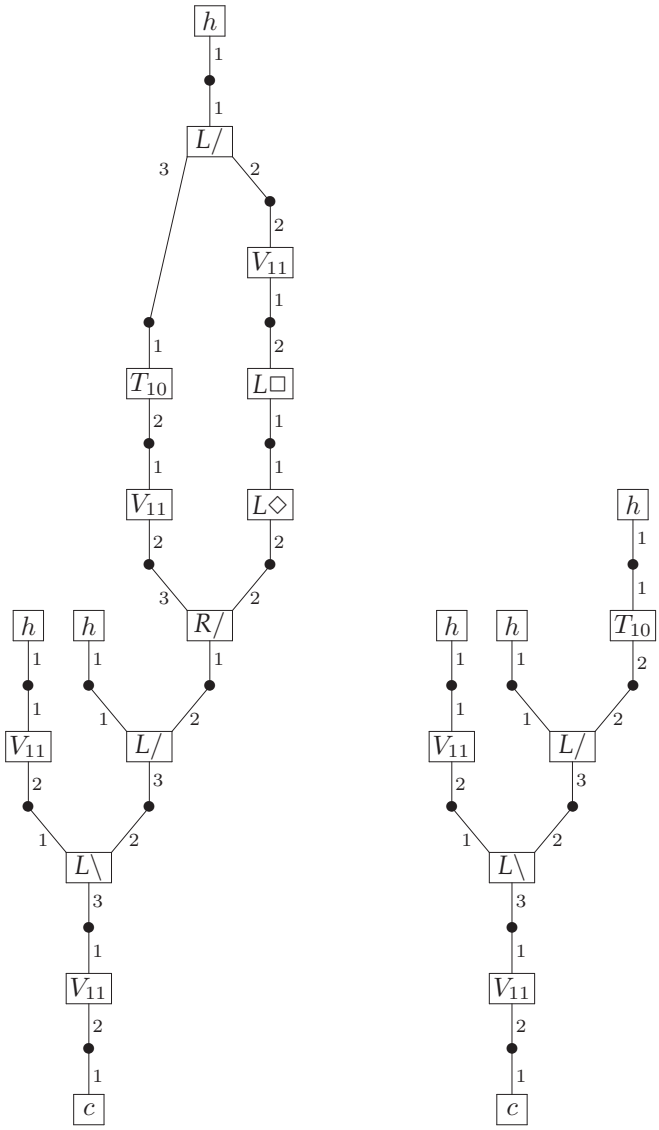



Figure 4.9: Applying a $T_{01}$ rule

# Chapter 5

# Conclusions

We have shown that proof nets for different multimodal Lambek calculi and different fragments of the Lambek-Grishin calculus can be generated by means of hyperedge replacement grammars.

# Bibliography


Ajdukiewicz, K. (1935), 'Die syntaktische Konnexität', *Studies in Philosophy* **1**, 1–27.

Bar-Hillel, Y. (1964), *Language and Information. Selected Essays on their Theory and Application*, Addison-Wesley, New York.

Bauderon, M. & Courcelle, B. (1987), 'Graph expressions and graph rewritings', *Mathematical Systems Theory* **20**(1), 83–127.

Bernardi, R. & Moortgat, M. (2007), Continuation semantics for symmetric categorial grammar, *in* 'Proceedings of WoLLIC 2007', Vol. 4567 of *LNCS*, Springer, pp. 53–71.

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

Courcelle, B. (1987), 'An axiomatic definition of context-free rewriting and its application to NLC graph grammars', *Theoretical Computer Science* **55**(2–3), 141–181.

Danos, V. (1990), La Logique Linéaire Appliquée à l'étude de Divers Processus de Normalisation (Principalement du $\lambda$-Calcul), PhD thesis, University of Paris VII.

Danos, V. & Regnier, L. (1989), 'The structure of multiplicatives', *Archive for Mathematical Logic* **28**, 181–203.

de Groote, P. (1999), The non-associative Lambek calculus with product in polynomial time, *in* N. V. Murray, ed., 'Automated Reasoning With Analytic Tableaux and Related Methods', Vol. 1617 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 128–139.

Drewes, F., Habel, A. & Kreowski, H.-J. (1997), Hyperedge replacement graph grammars, *in* G. Rozenberg, ed., 'Handbook of Graph Grammars and Computing by Graph Transformations', Vol. I, World Scientific, pp. 95–162.

Engelfriet, J. (1997), Context-free graph grammars, *in* G. Rosenberg & A. Salomaa, eds, 'Handbook of Formal Languages 3: Beyond Words', Springer, New York, pp. 125–213.

Engelfriet, J. & Heyker, L. (1992), 'Context-free hypergraph grammars have the same term-generating power as attribute grammars', *Acta Informatica* **29**(2), 161–210.

Girard, J.-Y. (1987), 'Linear logic', *Theoretical Computer Science* **50**, 1–102.

Grishin, V. N. (1983), On a generalization of the Ajdukiewics-Lambek system, *in* A. I. Mikhailov, ed., 'Studies in non-classical logics and formal systems', Nauka, Moscow, pp. 315–334.

Habel, A. & Kreowski, H.-J. (1987), May we introduce to you: Hyperedge replacement, *in* 'Graph Grammars and Their Application to Computer Science', Vol. 291 of *Lecture Notes in Computer Science*, Springer, pp. 15–26.

Johnson, M. (1998), 'Proof nets and the complexity of processing center-embedded constructions', *Journal of Logic, Language and Information* **7**(4), 443–447.

Lambek, J. (1958), 'The mathematics of sentence structure', *American Mathematical Monthly* **65**, 154–170.

Lautemann, C. (1990), 'The complexity of graph languages generated by hyperedge replacement', *Acta Informatica* **27**(5), 399–421.

Moortgat, M. (1997), Categorial type logics, *in* J. van Benthem & A. ter Meulen, eds, 'Handbook of Logic and Language', Elsevier/MIT Press, chapter 2, pp. 93–177.

Moortgat, M. (1999), Constants of grammatical reasoning, *in* G. Bouma, E. Hinrichs, G.-J. Kruijff & R. T. Oehrle, eds, 'Constraints and Resources in Natural Language Syntax and Semantics', CSLI, Stanford, pp. 195–219.

Moortgat, M. & Oehrle, R. T. (1993), 'Logical parameters and linguistic variation. Lecture notes on categorial grammar'. Fifth European Summer School in Logic, Language and Information, Lisbon.

Moortgat, M. & Oehrle, R. T. (1994), Adjacency, dependency and order, *in* 'Proceedings 9th Amsterdam Colloquium', pp. 447–466.

Moot, R. (2002), Proof Nets for Linguistic Analysis, PhD thesis, Utrecht Institute of Linguistics OTS, Utrecht University.

Moot, R. (2007), Proof nets for display logic, Technical report, CNRS and INRIA Futurs.

Moot, R. & Puite, Q. (2002), 'Proof nets for the multimodal Lambek calculus', *Studia Logica* **71**(3), 415–442.

Morrill, G. (1994), *Type Logical Grammar: Categorial Logic of Signs*, Kluwer Academic Publishers, Dordrecht.

Morrill, G. (1998), Incremental processing and acceptability, Technical Report LSI–98–46–R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.

Pentus, M. (1997), 'Product-free Lambek calculus and context-free grammars', *Journal of Symbolic Logic* **62**, 648–660.

Vermaat, W. (2005), The Logic of Variation. A cross-linguistic account of wh-question formation, PhD thesis, Utrecht Institute of Linguistics OTS, Utrecht University.

Vijay-Shanker, K., Weir, D. & Joshi, A. (1987), Characterizing structural descriptions produced by various grammatical formalisms, *in* 'Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics', Association for Computational Linguistics, Stanford, California, pp. 104–111.

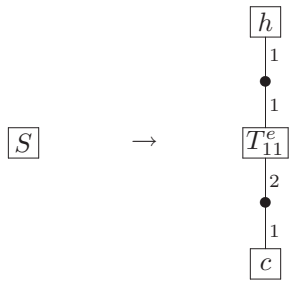
# APPENDIX A

# COMPLETE HR GRAMMAR FOR **NL**$\diamond$

This is the full **NL**$\diamond$ $HR_2$ grammar, that is to say it is a hyperedge replacement grammars where all non-terminals have two tentacles. All ternary non-terminals haven been compiled away and the axiom/cut/flow disctintions have been factored in. The non-terminal symbols of the grammar are $S$, $T_{00}$ (tree, cut), $T_{01}$ (tree, flow down), $T_{10}$ (tree, flow up), $T_{11}$ (tree, axiom), $V_{00}$ (vertex, cut), $V_{01}$ (vertex, flow down), $V_{10}$ (vertex, flow up), $V_{11}$ (vertex, axiom).

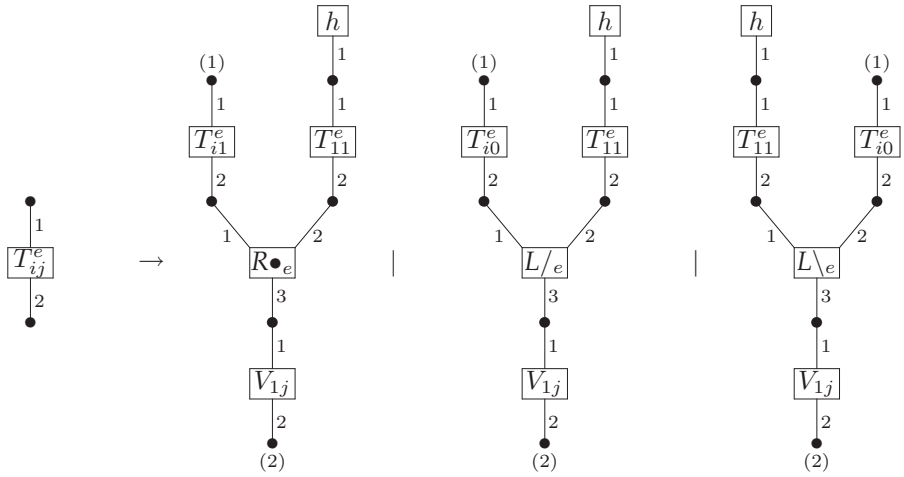
## A.1 Start: Initial Axiom

The initial axiom rewrites $S$ to a $T_{11}$ (axiom) vertex which is both a hypothesis and a conclusion of the abstract proof net.



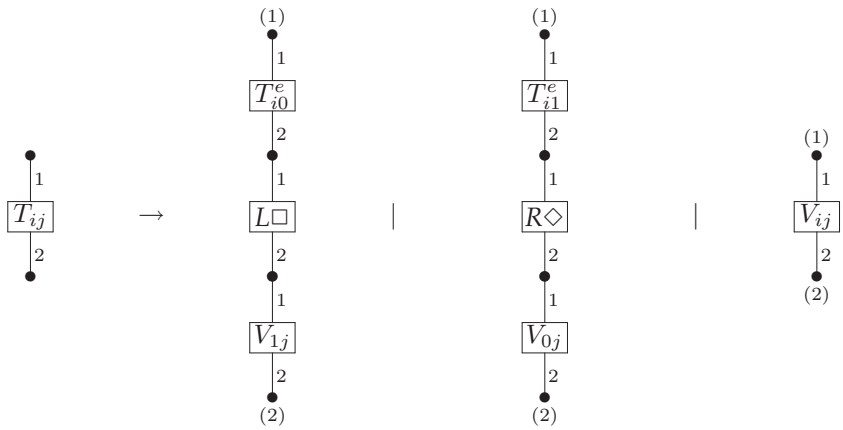

## A.2 Tensor: Binary

The $T$ binary tensor rules construct the final tensor tree where every vertex in the tensor tree is a $V$ edge.



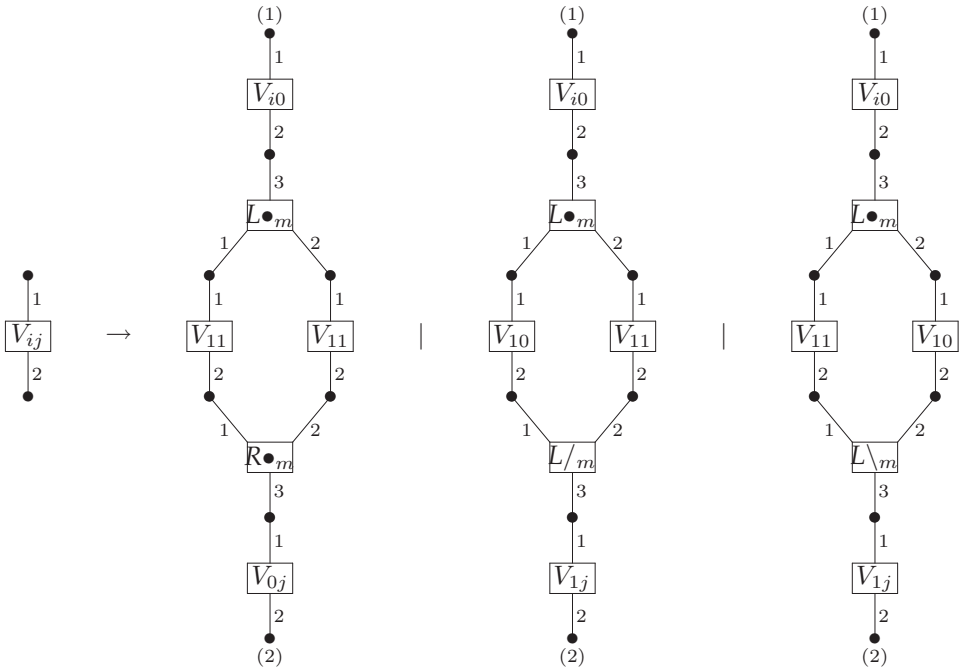

## A.3 Tensor: Unary

The two unary tensor rules and the base case, where we rewrite a tensor tree $T_{ij}$ to a single vertex $V_{ij}$.

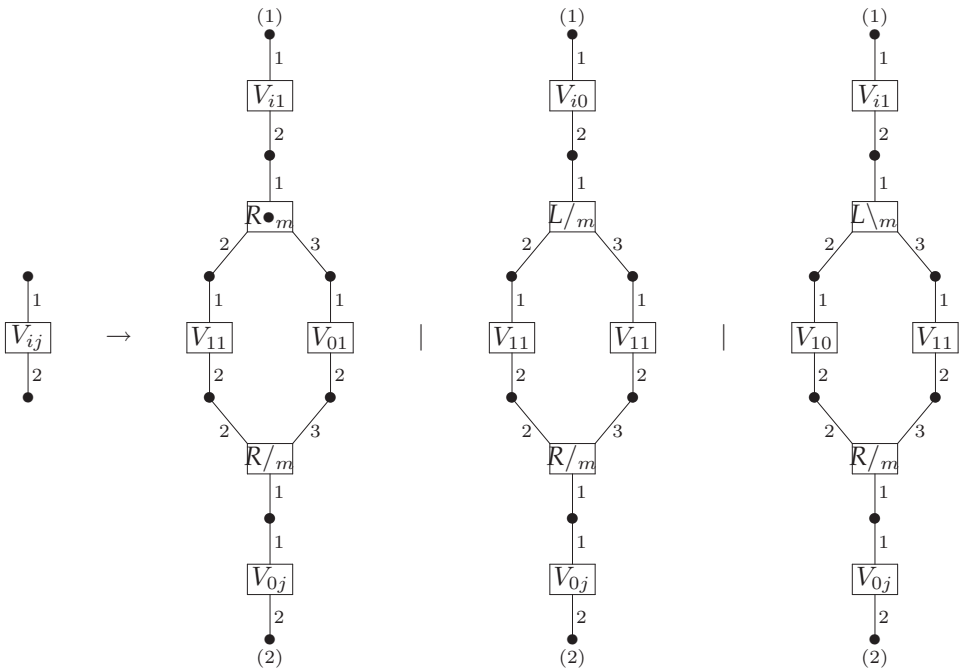



## A.4 Par: L•

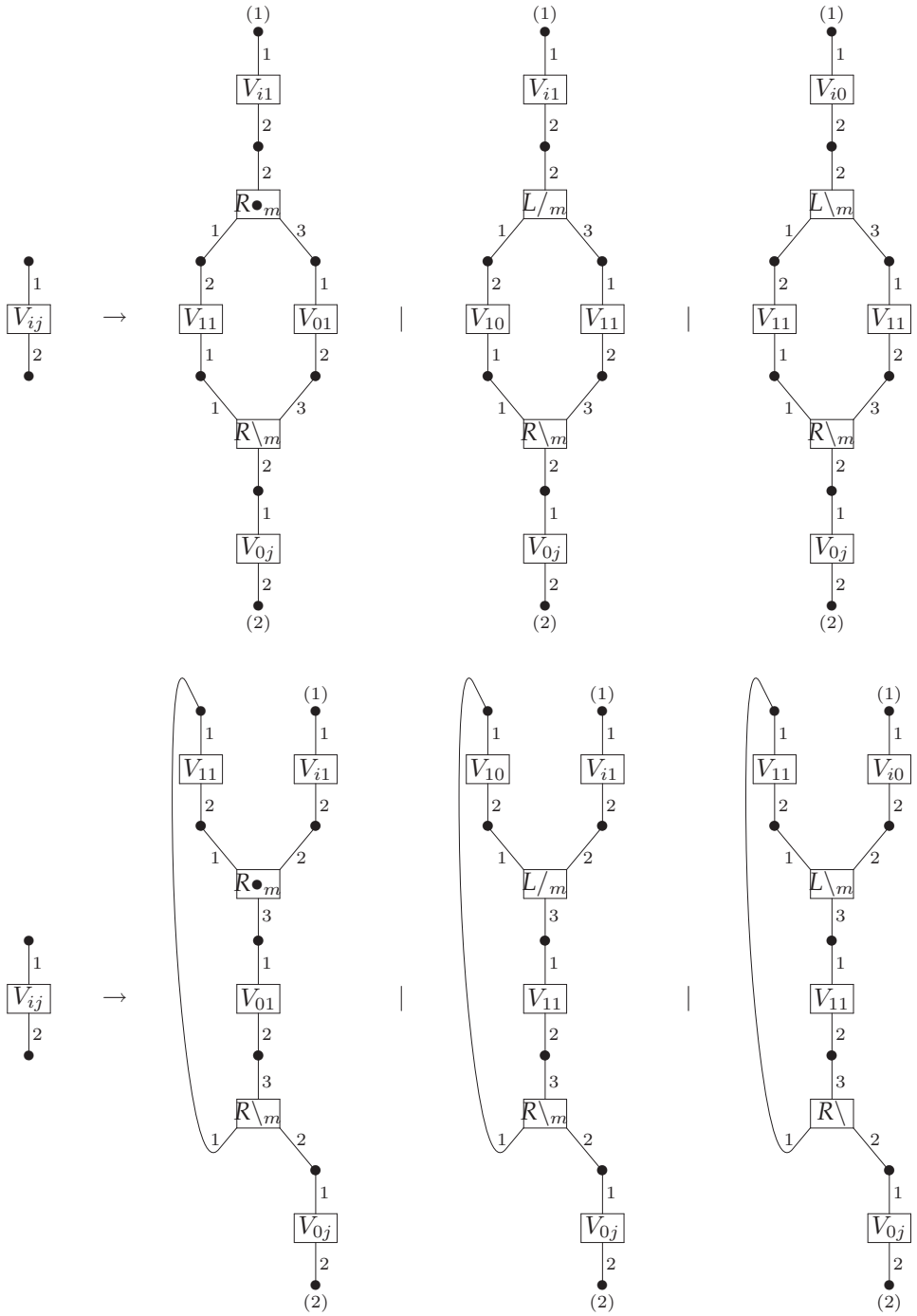
The par rules are all instances of the same schema: a par link is linked to a tensor link by all its tentacles except one, making sure the connecting tentacles have the same selector at both ends.


(1)
$V_{i0}$
$L\bullet_m$
$V_{ij}$
$V_{11}$
$V_{11}$
$R\bullet_m$
$V_{0j}$
(2)
(1)
$V_{i0}$
$L\bullet_m$
$V_{10}$
$V_{11}$
$L/_m$
$V_{1j}$
(2)
(1)
$V_{i0}$
$L\bullet_m$
$V_{11}$
$V_{10}$
$L\backslash_m$
$V_{1j}$
(2)

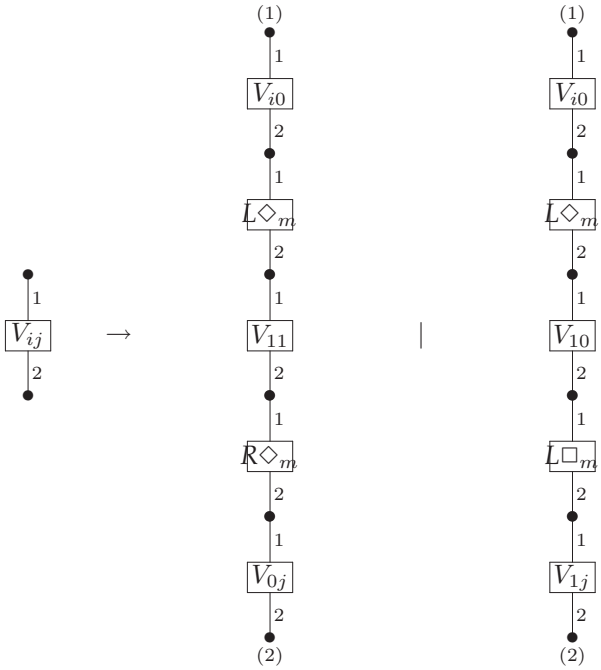
## A.5 Par: R/


(1)
$V_{i1}$
$R\bullet_m$
$V_{ij}$
$V_{11}$
$V_{01}$
$R/_m$
$V_{0j}$
(2)
(1)
$V_{i0}$
$L/_m$
$V_{11}$
$V_{11}$
$R/_m$
$V_{0j}$
(2)
(1)
$V_{i1}$
$L\backslash_m$
$V_{10}$
$V_{11}$
$R/_m$
$V_{0j}$
(2)

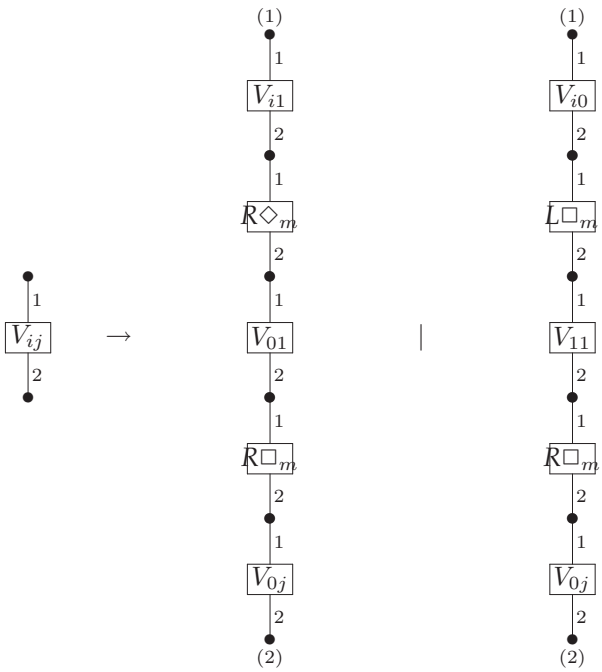
## A.6 Par: R\


(1)
$V_{i1}$
$R\bullet_m$
$V_{ij}$
$V_{11}$
$V_{01}$
$R\backslash_m$
$V_{0j}$
(2)
(1)
$V_{i1}$
$L/_m$
$V_{10}$
$V_{11}$
$R\backslash_m$
$V_{0j}$
(2)
(1)
$V_{i0}$
$L\backslash_m$
$V_{11}$
$V_{11}$
$R\backslash_m$
$V_{0j}$
(2)
(1)
$V_{11}$
$V_{i1}$
$R\bullet_m$
$V_{ij}$
$V_{01}$
$R\backslash_m$
$V_{0j}$
(2)
(1)
$V_{10}$
$V_{i1}$
$L/_m$
$V_{11}$
$R\backslash_m$
$V_{0j}$
(2)
(1)
$V_{11}$
$V_{i0}$
$L\backslash_m$
$V_{11}$
$R\backslash$
$V_{0j}$
(2)

## A.7 Par: L◇

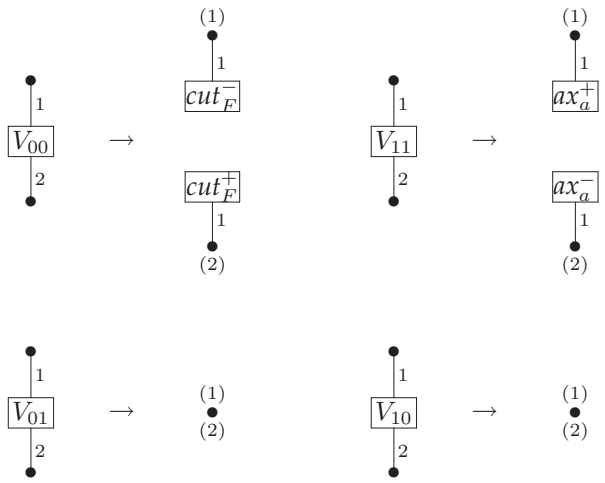



## A.8 Par: R□



## A.9 End: Cut/Flow/Axiom

These rules end the derivation. Flow up ($V_{01}$) and flow down ($V_{10}$) edges are deleted. Axiom and cut edges are separated into their positive and negative parts. To obtain cut-free proof nets, simply erase all rules containing occurrences of either $T_{00}$ or $V_{00}$.



# Appendix B

# Adding Structural Rules: HRG for $\mathbf{NL}\Diamond_{\mathcal{R}}$

## B.1 Multimodality

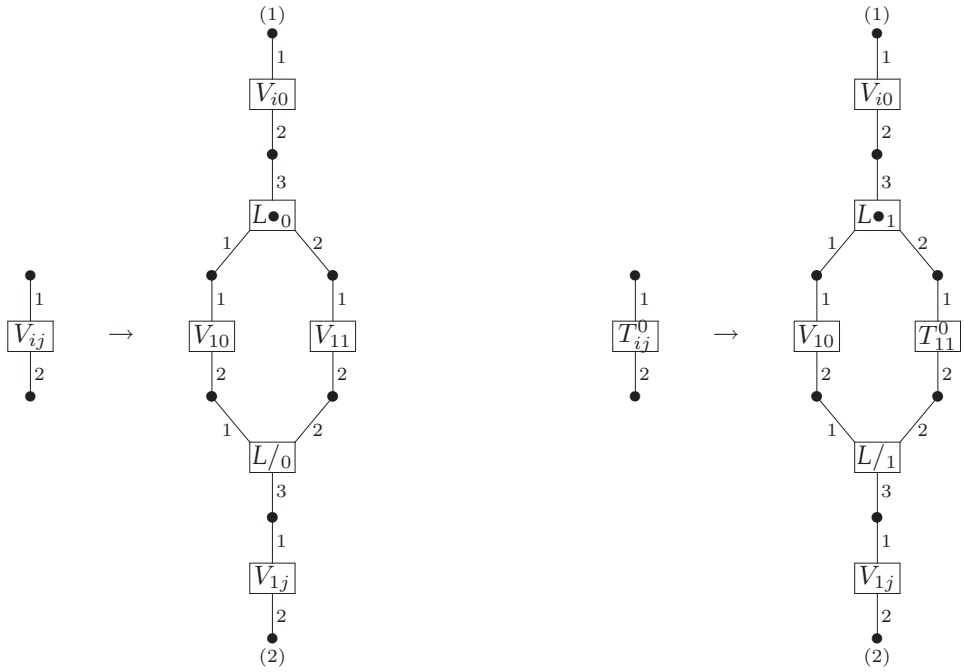
A first extension to $\mathbf{NL}\Diamond$ is multimodality. Instead of having a single family of connectives, an $\mathbf{NL}\Diamond_{\mathcal{R}}$ grammar $G$ can have a set $M$ of modes. For every $m \in M$ there is a set of connectives $\{/_m, \bullet_m, \backslash_m, \Diamond_m, \Box_m\}$. In addition there are structural connectives $\{\circ_m, \langle\,\rangle^m\}$ for each mode. Finally, there is a set $E \subseteq M$ of *external* modes. These are the modes which are allowed to appear as structural connectives on the final tensor tree.

The rules of Appendix A already implement the multimodal version of $\mathbf{NL}\Diamond$. For every external mode $e \in E$ there is a $T^e$ tree rule constructing a tree with a structural connective $e$ and for every mode $m \in M$, the internal modes include, there is a $V$ vertex rule contracting a tensor and par link of mode $m$.

## B.2 Mixed Associativity and Mixed Commutativity

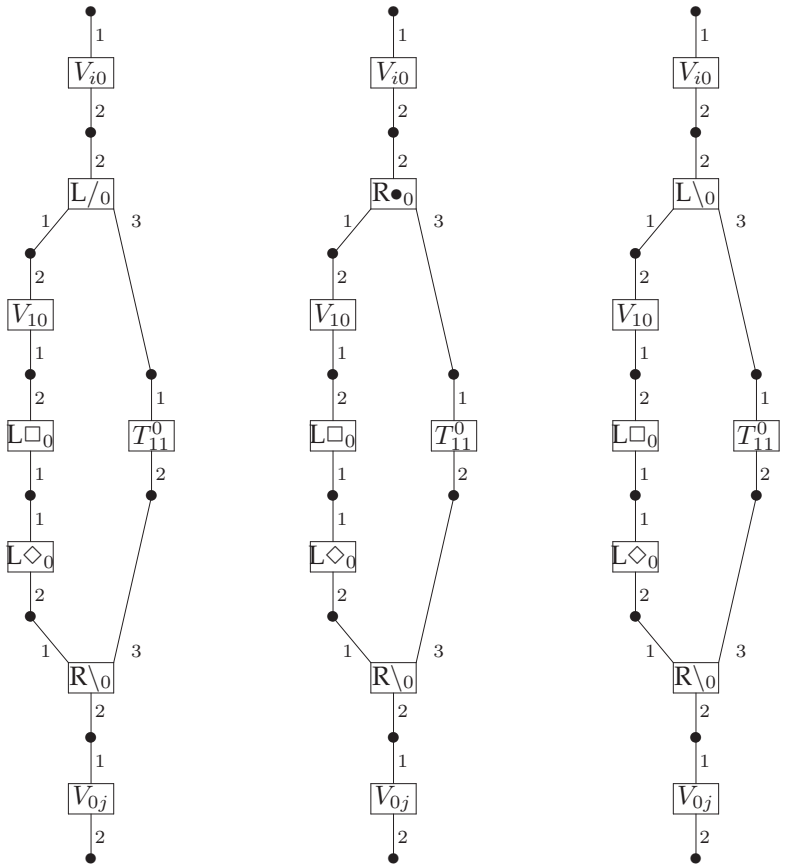
Mixed assiociativity and commutativity use an external mode 0 and an internal mode 1.

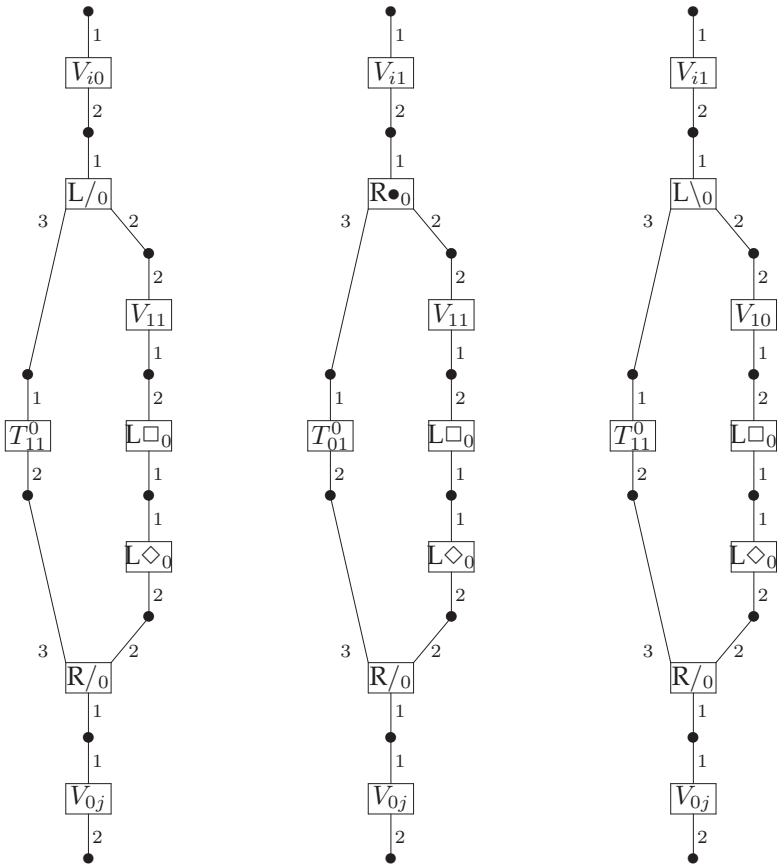
The case for embedding tree adjoining grammars, using $(A/_1B) \bullet_1 B$ uses the following schema. The contraction for mode 0 uses the normal HR rule, while mode 1 next to it differs just in the choice of $T$ instead of $V$, which permets a tensor tree to insert itself at the place of the vertex in the original rule. To show this rule is correct, we need to show that for any tensor tree $T$ in the position indicate in the contraction the structural rules allow us to rewrite them.




## B.3 Mixed Associativity and Mixed Commutativity With Unary Control

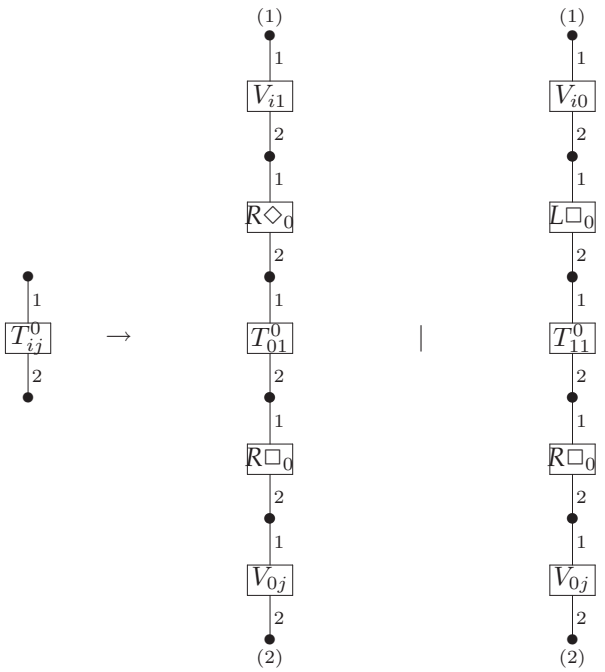
For mixed associativity and mixed commutativity with unary control, we only consider two very restricted cases: either positive occurrences of $A/_0\diamond_0\Box_0 B$ together with $U1$ and $U2$ or positive occurences of $\diamond_0\Box_0 B\backslash_0 A$ together with $U3$ and $U4$.


$V_{i0}$
L/$_0$
$V_{10}$
L□$_0$
L◇$_0$
R\$_0$
$V_{0j}$
$T^0_{11}$
R•$_0$
L\$_0$




## B.4 The K1, K2 interaction rules

The $K1$ and $K2$ interaction rules are easy to implement. Given that the unary mode 0 is internal to the grammar, we know that every $R\diamond_0$ and $L\Box_0$ rule must be paired to either an $L\diamond_0$ or an $R\Box_0$ rule.


1
$T^0_{ij}$
2
→
(1)
1
$V_{i1}$
2
1
$R\diamond_0$
2
1
$T^0_{01}$
2
1
$R\square_0$
2
1
$V_{0j}$
2
(2)
|
(1)
1
$V_{i0}$
2
1
$L\square_0$
2
1
$T^0_{11}$
2
1
$R\square_0$
2
1
$V_{0j}$
2
(2)

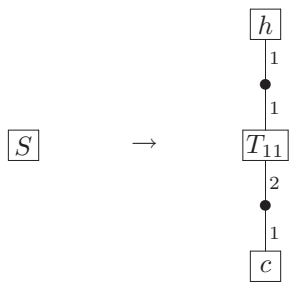
# Appendix C

# HR Rules for Lambek-Grishin

The grammars of Appendix A and B extend easily to the Lambek-Grishin calculus. We can even incorporate the Grishin class IV interactions in a way similar to the incorporation of mixed associativity and mixed commutativity in Appendix B.2

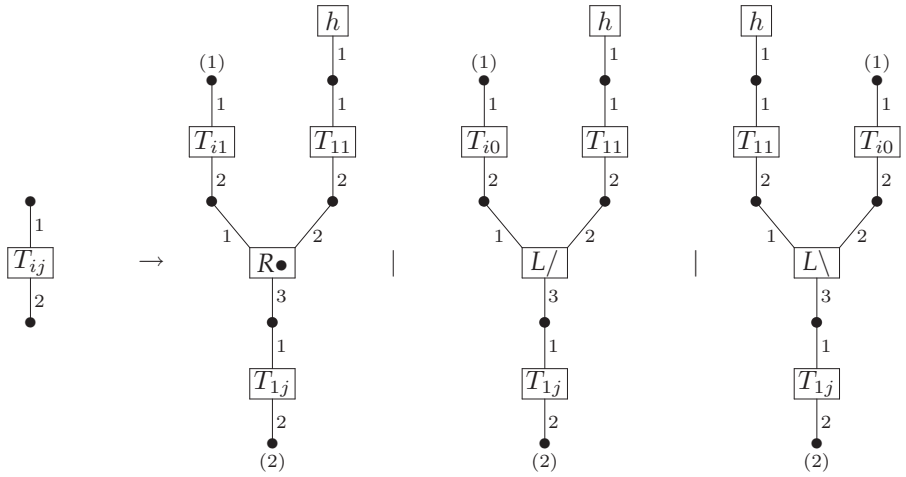
## C.1 Start: Initial Axiom

The initial axiom rewrites $S$ to a $T_{11}$ (axiom) vertex which is both a hypothesis and a conclusion of the abstract proof net.

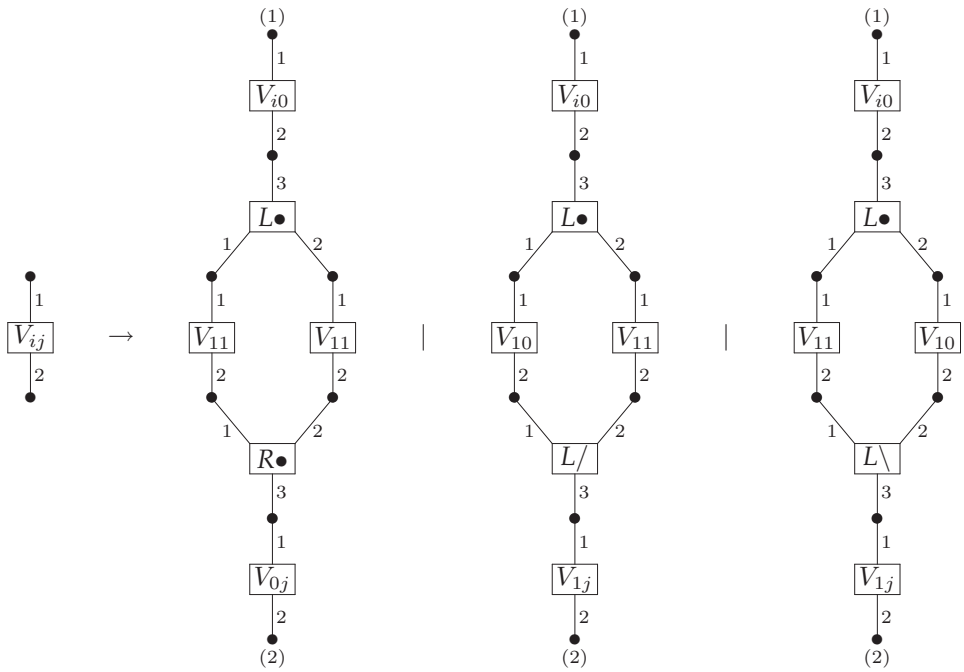



## C.2 Tensor: Binary

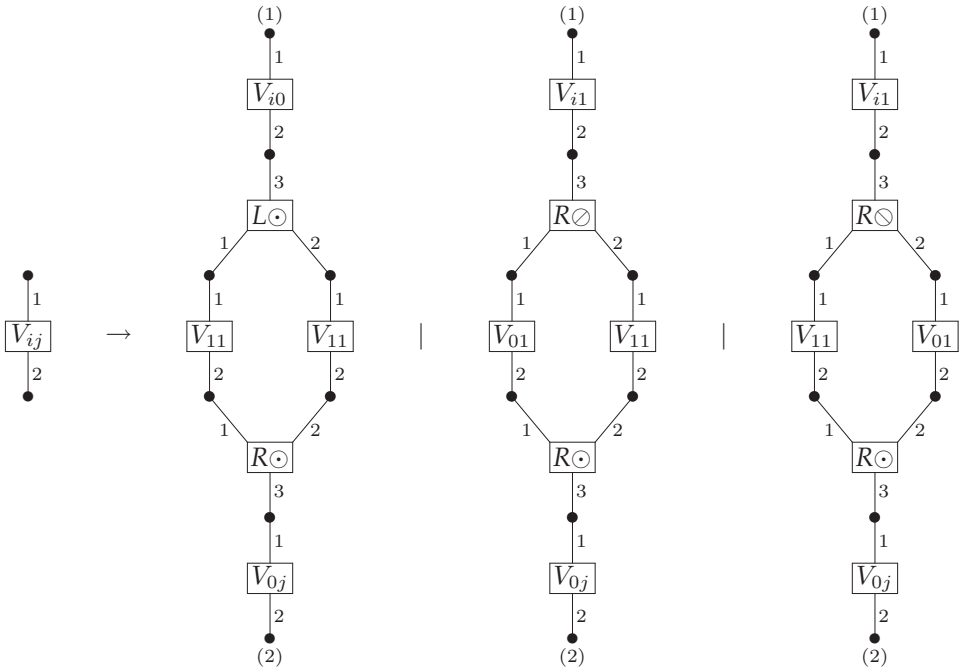
As usual, the $T$ rules construct the final tensor tree, but we require it consists of Lambek constructors only.


$T_{ij}$
$\rightarrow$
$h$
$T_{i1}$
$T_{11}$
$R\bullet$
$T_{1j}$
$T_{i0}$
$L/$
$L\backslash$
(1)
(2)

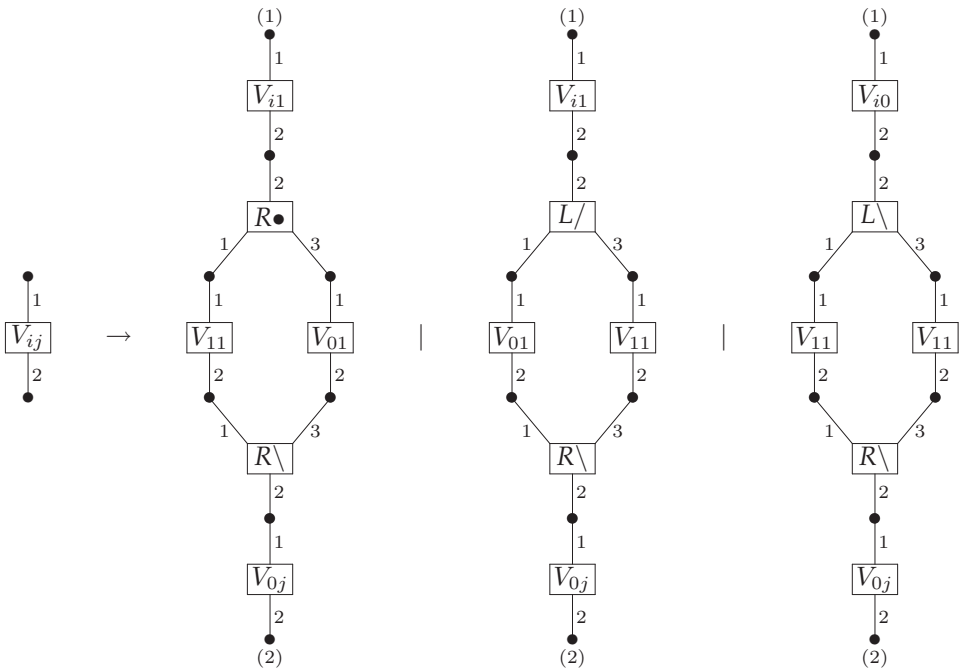
## C.3 Par: L•


$V_{ij}$
$\rightarrow$
$V_{i0}$
$L\bullet$
$V_{11}$
$R\bullet$
$V_{0j}$
$V_{10}$
$L/$
$V_{1j}$
$L\backslash$
(1)
(2)

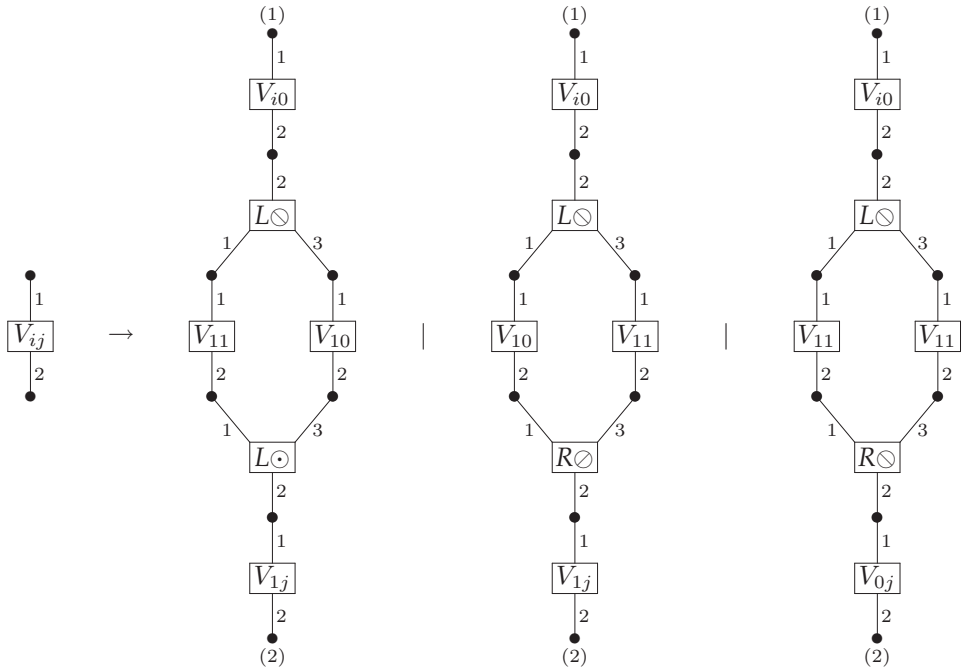
## C.4 Par: R⊙



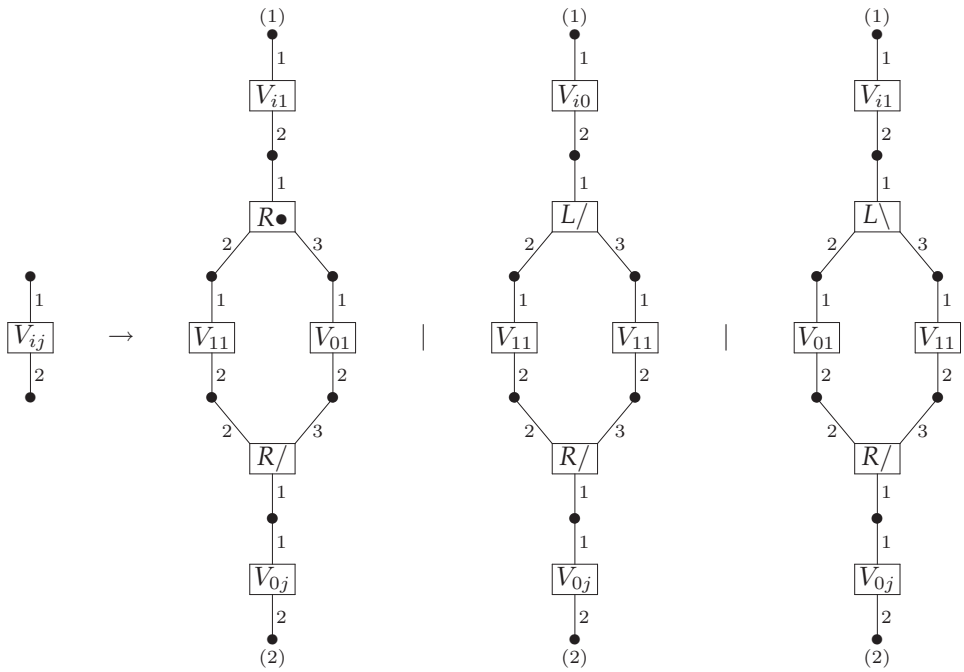

## C.5 Par: R\



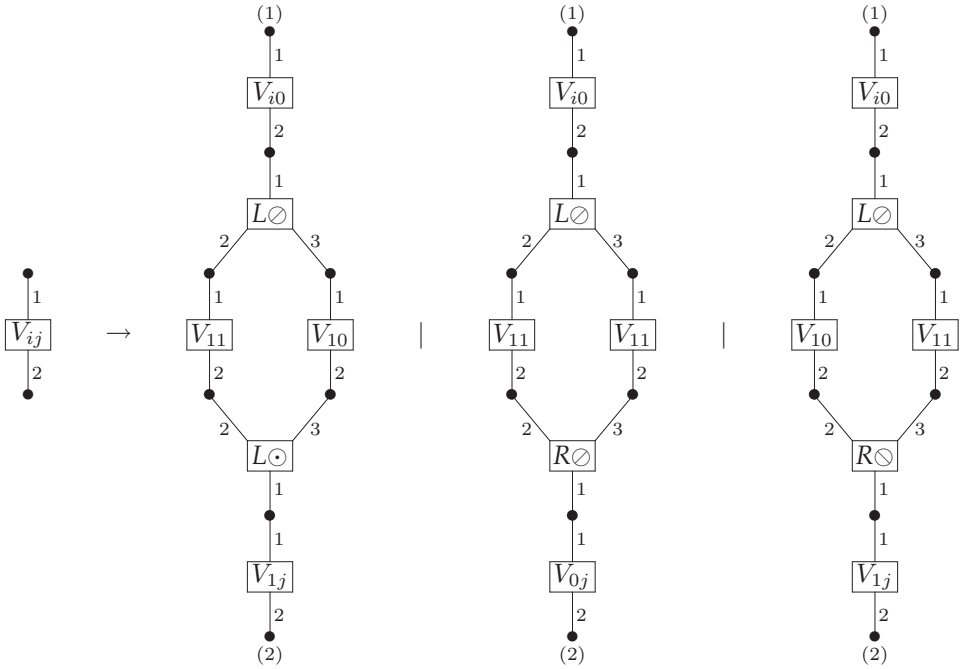
## C.6 Par: L⊘




## C.7 Par: R/

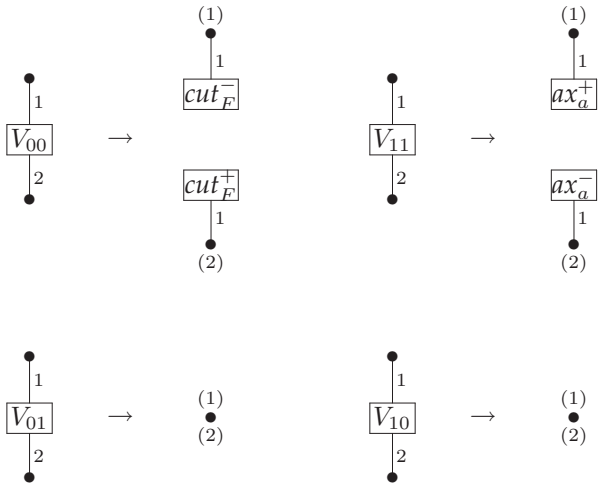
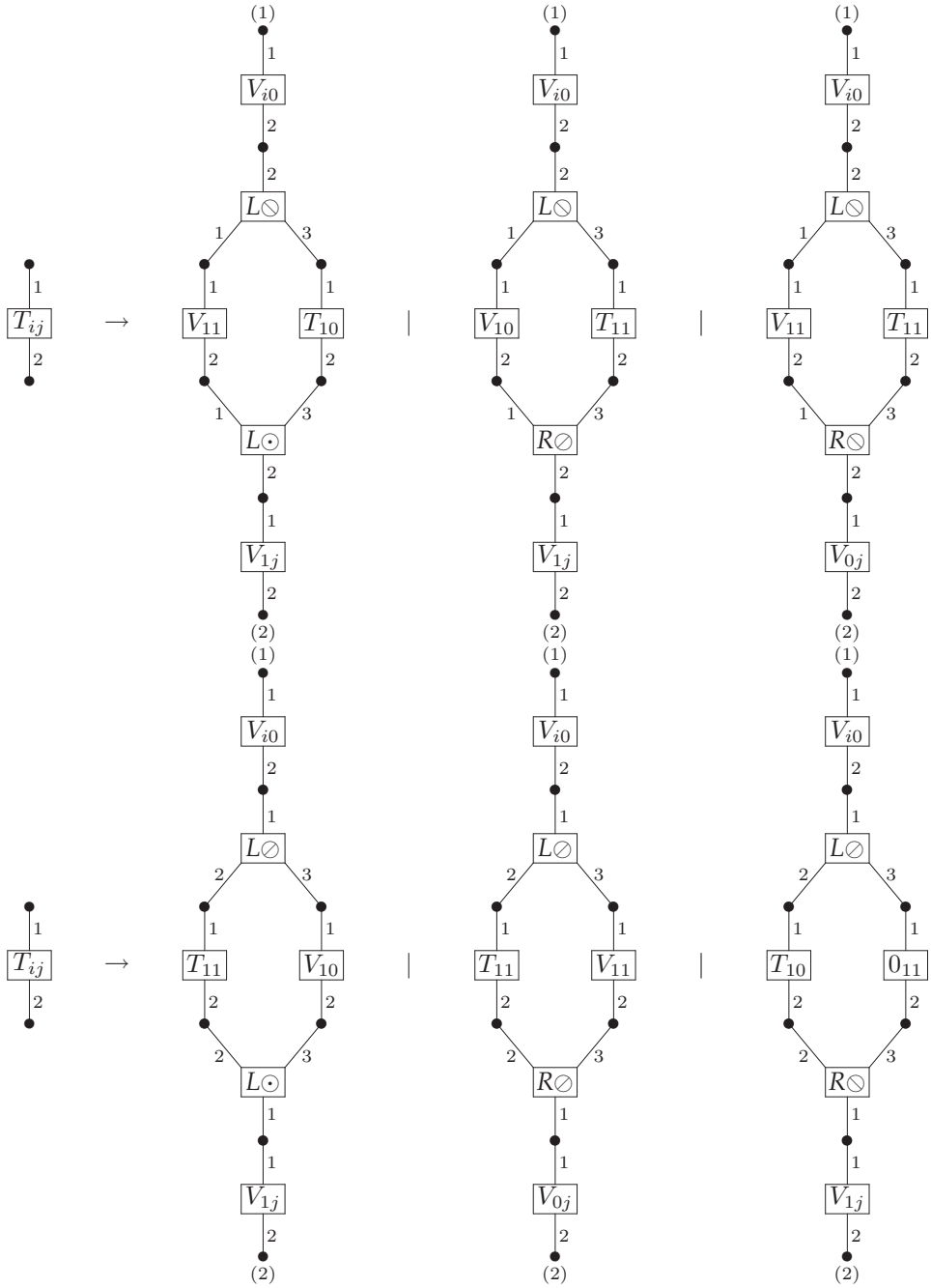


## C.8 Par: L⊘




## C.9 End: Cut/Flow/Axiom

The end rules remain unchanged.



## C.10 The Class IV Interactions