
Proof Nets for Linguistic Analysis

Bewijsnetten voor Taalkundige Analyse
(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit Utrecht op gezag van de Rector Mag-
nificus, Prof. dr. W. H. Gispen, ingevolge het
besluit van het College voor Promoties in het
openbaar te verdedigen op vrijdag 1 februari 2002
des middags te 12.45 uur

door

Richard Cornelis Antonius Moot

geboren op 24 december 1972 te Alkmaar

Promotor: Prof. dr. M. J. Moortgat
Utrecht Institute of Linguistics OTS
Utrecht University

Printed by PrintPartners Ipskamp, Enschede

©2002 Richard Moot

THIS book concludes five years of research into the linguistic applications of proof nets. While mostly self-contained, people already familiar with either linear logic or Lambek calculi will probably have an easier time reading through it, and people familiar with both can probably skip Part I altogether.

Part II contains your recommended daily dose of proof nets. Proof nets for **MLL**, **MILL**, **MILL1**, L_e , labeled proof nets and proof nets for $NL \diamond_{\mathcal{R}}$. This last proof net calculus is new, and we will provide a correction criterion for this calculus and prove soundness and completeness results.

Part III builds on the proof nets for $NL \diamond_{\mathcal{R}}$ and contains reflections on automated deduction using proof nets, an analysis of the complexity of the logic and results on the relation between proof nets and lexicalized tree adjoining grammars.

Chapter 5 consists of joint work with Mario Piazza, which has appeared before in (Moot & Piazza 2001).

Chapter 7 consists of joint work with Quintijn Puite and contains work which has appeared before in (Puite & Moot 1999), (Moot & Puite 1999) and (Moot & Puite 2001).

ACKNOWLEDGMENTS

MANY chapters of this thesis were difficult to write, but none was as difficult as the current one, because omissions or mistakes in this chapter will have more far-reaching consequences than mistakes in any other chapter.

First of all, I would like to thank Michael Moortgat for his support, insight and enthusiasm.

Secondly, I would like to thank Quintijn Puite. The many meetings we had discussing linear logic and proof nets helped me a lot and our joint work resulted in Chapter 7 of this thesis.

I would also like to thank Mario Piazza for our collaboration on Chapter 5.

Many thanks go to Jan van Eijck, Philippe de Groote, François Lamarche, Glyn Morrill, Christian Retoré, Harold Schellinx and Ed Stabler for their discussion and comments on several aspects of this thesis during its development.

I am grateful to the thesis committee, Johan van Benthem, Jan van Eijck, Aravind Joshi, François Lamarche and Albert Visser, for taking the time to study this thesis.

Special thanks go to Raffaella Bernardi, Patrick Brandt, Christophe Costa Florêncio, Dirk Heylen, Willem-Olaf Huijsen and Esther Kraak, my roommates at room 2.05 over the years, for creating a pleasant environment for me to work in.

I would like to thank my friends and colleagues at UiL-OTS: Olga Borik, Jenny Doetjes, Herman Hendriks, Ellen Gerrits, Paz Gonzalez, Silke Hamann, Heleen Hoekstra, Maarten Janssen, Oele Koornwinder, Laura Korte, Steven Krauwer, Anne-Marie Mineur, Anna Mlynarczyk, Paola Monachesi, Iris Mulders, Øystein Nilsen, Rick Nouwen, Renée Pohlmann, Louis des Tombe, Sharon Unsworth, Henk Verkuyl, Willemijn Vermaat, Jules van Weer-

den, Yoad Winter, Ton van der Wouden and Inge Zwitserlood.

Extra special thanks go to the band members of *Evisceration* and *Horizon*, Marcel 'Homi' Hommes, Robert 'Hout' Houtenbos, Peter 'Pieke' Kops, Ronald Kools, Niek Kuijper, John Ruiten, Michel 'Mick' Switser and Finus Tromp, for making my life considerably less quiet and more entertaining.

Furthermore, I would like to thank the D&D crew, Robert Dijkman, Jan Klinkspoor, Karel Klinkspoor and Wim Pool for keeping my imagination running.

I would like to thank my parents, my sister and Dave for their care and for supporting me, even though my research subject has remained very mysterious to them.

Finally, I want to thank Adriana for her love and support and understanding.

With all these wonderful people supporting me, I have of course only myself left to blame for any remaining mistakes.

| | | |
|----------|--|----------|
| 1 | Introduction | 1 |
| 1.1 | Overview | 3 |
| I | Logics and Grammars | 5 |
| 2 | Linear Logic | 7 |
| 2.1 | Sequent Calculus | 8 |
| 2.1.1 | Multiplicatives | 8 |
| 2.1.2 | Additives | 9 |
| 2.1.3 | Exponentials | 10 |
| 2.1.4 | Quantifiers | 11 |
| 2.1.5 | Cut Elimination | 11 |
| 2.2 | Fragments of Linear Logic | 13 |
| 2.3 | One Sided Sequent Calculus | 14 |
| 2.4 | Intuitionistic Linear Logic | 14 |
| 2.4.1 | Natural Deduction | 15 |
| 2.4.2 | The Curry-Howard Isomorphism | 17 |
| 2.4.3 | Normalisation | 19 |
| 2.5 | Linguistic Applications | 20 |
| 2.5.1 | Multiplicatives | 20 |
| 2.5.2 | Additives | 21 |
| 2.5.3 | Units | 23 |
| 2.5.4 | Exponentials | 23 |
| 2.5.5 | Quantifiers | 24 |
| 2.5.6 | Structural Rules | 25 |
| 2.5.7 | Semantics | 26 |
| 2.6 | Complexity | 28 |
| 2.7 | Conclusions | 29 |

| | | |
|-----------|---|-----------|
| 3 | Lambek Calculi | 31 |
| 3.1 | Lambek Calculus | 31 |
| 3.2 | Non-Associative Lambek Calculus | 33 |
| 3.3 | Multimodal Lambek Calculus | 35 |
| 3.4 | Unary Operators | 38 |
| 3.5 | Natural Deduction | 42 |
| 3.6 | Model Theory | 45 |
| 3.7 | Conclusions | 46 |
| II | Proof Nets and Linguistics | 47 |
| 4 | Proof Nets for Multiplicative Linear Logic | 49 |
| 4.1 | Proof Nets | 50 |
| 4.2 | Proof Structures | 52 |
| 4.3 | Soundness and Completeness | 53 |
| 4.4 | Cut Elimination | 58 |
| 4.5 | Contractions | 61 |
| 4.6 | The Intuitionistic Fragment | 63 |
| 4.7 | Non-commutative Proof Nets | 65 |
| 4.8 | Conclusions | 68 |
| 5 | Proof Nets for First Order Linear Logic | 69 |
| 5.1 | Proof Theory | 69 |
| 5.1.1 | Sequent Calculus | 70 |
| 5.1.2 | Proof Structures | 71 |
| 5.1.3 | Proof Nets | 72 |
| 5.1.4 | Embedding the Lambek Calculus | 73 |
| 5.2 | Linguistic Applications | 80 |
| 5.2.1 | Quantifier Scope | 81 |
| 5.2.2 | Relative Pronouns | 83 |
| 5.2.3 | Locality | 86 |
| 5.3 | Conclusions | 86 |
| 6 | Algebraic Criteria | 89 |
| 6.1 | Labeled Sequent Calculus | 90 |
| 6.2 | Labeled Natural Deduction | 91 |
| 6.3 | Labeled Proof Nets | 93 |
| 6.4 | Conclusion | 98 |
| 7 | Contraction Criteria | 99 |
| 7.1 | Two Sided Proof Nets | 99 |
| 7.2 | Sequent Calculus | 103 |
| 7.3 | Proof Structures | 107 |
| 7.4 | Abstract Proof Structures | 109 |
| 7.5 | Proof Nets | 112 |
| 7.6 | Cut Elimination | 119 |

| | | |
|---------------------------------------|---|------------|
| 7.7 | Abstract Proof Structures and Labels | 122 |
| 7.8 | Lambek Calculus | 124 |
| 7.9 | Discussion | 128 |
| III Relations and Computations | | 131 |
| 8 | Automated Deduction | 133 |
| 8.1 | Invariants | 136 |
| 8.2 | Compiling the Lexicon | 137 |
| 8.3 | Acyclicity and Connectedness | 138 |
| 8.4 | Axiomatic Connections | 138 |
| 8.5 | Components | 141 |
| 8.6 | Focusing | 144 |
| 8.7 | Word Order | 145 |
| 8.8 | Parallel Computation | 146 |
| 8.9 | Rule Filtering | 149 |
| 8.10 | Conclusions | 151 |
| 9 | Complexity | 153 |
| 9.1 | NP Complete Fragments | 154 |
| 9.2 | Restricted Structural Rules | 155 |
| 9.3 | $NL \diamond_{\mathcal{R}}$ | 166 |
| 9.4 | Conclusions | 172 |
| 10 | Proof Nets and Tree Adjoining Grammars | 173 |
| 10.1 | Tree Adjoining Grammars | 174 |
| 10.2 | Substitution Only Tree Adjoining Grammars | 178 |
| 10.3 | Lexicalized Tree Adjoining Grammars | 182 |
| 10.4 | Adjoining Constraints | 193 |
| 10.5 | Multi Component Tree Adjoining Grammars | 196 |
| 10.6 | Discussion | 199 |
| 11 | Conclusions | 201 |
| 11.1 | Further Research | 202 |
| A | The Grail Theorem Prover | 203 |
| A.1 | History | 203 |
| A.2 | Tutorial | 204 |
| A.2.1 | Getting Started | 204 |
| A.2.2 | The Lexicon | 204 |
| A.2.3 | The Theorem Prover | 210 |
| A.2.4 | The Structural Postulates | 217 |
| A.3 | Reference Guide | 221 |
| A.3.1 | The Main Window | 221 |

| | | |
|-------|---------------------------------------|------------|
| A.3.2 | The Status Window | 224 |
| A.3.3 | The Proof Net Window | 225 |
| A.3.4 | The Rewrite Window | 227 |
| A.3.5 | The Lexicon Window | 229 |
| A.3.6 | Editing a Lexical Entry | 230 |
| A.3.7 | The Postulate Window | 233 |
| A.3.8 | Editing a Postulate | 235 |
| A.3.9 | The Analysis Window | 236 |
| A.4 | Conclusions | 238 |
| | Bibliography | 239 |
| | Index | 247 |
| | Samenvatting in het Nederlands | 251 |
| | Curriculum Vitae | 253 |

CHAPTER 1

INTRODUCTION

LOGIC is a powerful set of tools which have found applications in many areas of artificial intelligence. The Lambek calculus (Lambek 1958) and its extensions and generalizations (Morrill 1994, van Benthem 1995, Moortgat 1997) have been the principal systems used for giving a logical account of natural language.

The advantages of a logical view of grammar are manifold.

- we can prove *soundness* of our system, that is we can show all operations in our grammar are meaningful and well-defined with respect to a model.
- we can prove *completeness* of our system, that is we can show that if something is valid in our model, it is therefore derivable in our logic as well.
- we can prove *consistency* of our system. This means that it is fundamentally impossible in our logic that a statement and its negation are both true.

Of course, we can also prove the correctness of algorithms given a precise mathematical specification of the preconditions and the postconditions of the algorithm, as advocated by Dijkstra (1979). However, in this case, changes in the algorithm will force us to reprove correctness, whereas for a logic we prove soundness and completeness of the logic itself and if we should choose to apply this logic to a different domain no new proofs will be necessary.

Figure 1.1 gives an overview of the logical approach to grammar. A parser for a logical grammar consists of a lexicon, which assigns sets of formulas to words and a theorem prover which proves that statements in the logic are either provable or unprovable.

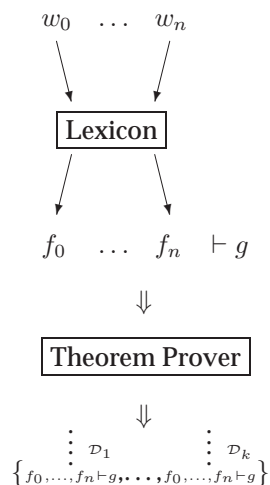


Figure 1.1: A schematic overview of the logical approach to grammar

The slogan here is *parsing as deduction*. This term was introduced by Pereira, Warren and others (Pereira & Warren 1983), (Pereira & Warren 1980), who used the definite clause grammar facilities of the general purpose logic programming language Prolog as a logical way of describing grammars. A good overview of using Prolog for natural language analysis can be found in (Pereira & Shieber 1987) and (Shieber, Schabes & Pereira 1995).

Our goal here is more ambitious, instead of using a general purpose logic to give a description of our grammar and run this description as a program, we want to use a logic which is inherently suitable for linguistic descriptions. We are interested in what Moortgat calls the ‘constants of grammatical reasoning’ (Moortgat 1999); the logical universals when we reason about grammars.

A Lambek calculus theorem prover should return a *set* of proofs: the set of different proofs for the given logical statement. A potential problem here is that we are only interested in proofs which are different for interesting reasons, as opposed to proofs which differ only because of an overly bureaucratic proof system. The introduction of a ‘redundancy free’ formulation of proofs, proof nets, will solve this problem for us. Different proof nets are semantically different linguistic objects. In Section 3.5, we will see that there is a very general way of assigning lambda term semantics to Lambek calculus proofs, which is an attractive property of Lambek calculi and it is one we would be a shame to lose this property because of the syntactic peculiarities of our proof system. Finding and analysing proof net calculi suitable for linguistic analysis will be the central theme of this book.

Though we advocate a lexicalist account of grammar we do not want to shift too much of the explanatory burden to the lexicon. As shown by Carpenter (1991), adding recursive lexical rules to even a very simple grammar results in an undecidable system. Therefore, we will assume that the lexicon

assigns a finite number of formulas to every word in the lexicon.

We also want to avoid assigning formulas to the empty string, because this will increase the expressiveness of the system in much the same way as adding lexical rules does; every logical formula we pass to the theorem prover is keyed to a word in the input sentence.

1.1 Overview

This book is structured as follows.

Part I is an introduction to the use of logic for grammatical analysis.

Chapter 2 will introduce linear logic as a first approximation and suggest linguistic applications for the different groups of logical connectives. A clear limitation of linear logic is the global availability of the structural rule of commutativity, which incorrectly predicts any permutation of a grammatical sentence is also grammatical.

In Chapter 3 we will introduce the Lambek calculus, a non-commutative precursor of linear logic, and show how the addition of multiple families of connectives and of unary modal connectives increases the linguistic sophistication of the theory.

Part II is the main part of this book and focuses on how to use proof nets for linguistic analysis.

Chapter 4 introduces proof nets as a redundancy free representation for proofs in multiplicative linear logic and for proofs in the associative Lambek calculus.

Chapter 5 consists of joint work with Mario Piazza, which has appeared before in (Moot & Piazza 2001). We will study proof nets for the first order fragment of multiplicative linear logic and give an embedding translation for both the associative and the non-associative Lambek calculus into the first order fragment. We will also see how to use the first order fragment for a treatment of linguistic phenomena which have no satisfactory treatment in the Lambek calculus.

In Chapter 6 we will look at labeled deduction as a way of adding structural constraints to our logical calculi and to the proof nets for multiplicative linear logic.

Chapter 7 consists of joint work with Quintijn Puite, and large parts of it have appeared before in (Puite & Moot 1999, Moot & Puite 1999, Moot & Puite 2001), contains the main result of this book: by reformulating proof nets in a more symmetric way, we give a proof net system for the multimodal Lambek calculus and prove it is sound and complete with respect to the sequent formulation of the same calculus.

In Part III we will discuss some computational implications of the proof net calculus of Chapter 7 and relate it to other formalisms.

Chapter 8 will be devoted to automated deduction. We will present an algorithm for proof search using proof nets and some heuristics for improving the performance of this algorithm.

In Chapter 9 we will establish a PSPACE complexity result for the multimodal Lambek calculus with a restriction of the form of the structural rules and show that without this restriction the logic is undecidable.

Chapter 10 will give an embedding of Lexicalized Tree Adjoining Grammars into multimodal proof nets.

Finally, Chapter 11 contains my concluding remarks and a discussion of possible future research.

An Appendix gives an introduction to the Grail automated theorem prover for the multimodal Lambek calculus, which was developed as part of this research project.

PART I

LOGICS AND GRAMMARS

CHAPTER 2

LINEAR LOGIC

LINEAR logic was introduced by Girard (1987) as a resource conscious logic, that is a logic where the number of occurrences of a formula matters. This is not true in classical logic. The structural rule of contraction from classical logic

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} [LC] \quad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} [RC]$$

tells us that whenever we can prove something using the formula A as a premiss twice, we can also prove it using A just once. When we interpret A as, for example, a lemma we use in a proof, it makes sense to say that re-proving the lemma each time we use it is unnecessary. But when we want to interpret A as a *resource*, a physical as opposed to an ideal entity, then it makes sense to reject the rule of contraction, at least globally. The number of occurrences of our resources generally *do* matter to us.

Similarly, the rule of weakening

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} [LW] \quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \Delta} [RW]$$

states that whenever we can prove Δ with assumptions Γ we can add an extra assumption A and still have a valid proof. Again, if we interpret A as a resource we can't just get it from thin air, we have to justify how we obtained it. So global availability of weakening is also undesirable from a resource conscious point of view.

For our intended linguistic applications, the absence of contraction and weakening seems like a good starting point, since, in general, the number of occurrences of words in a sentence is quite relevant to the grammaticality of the sentence.

When we remove contraction and weakening, however, we are faced with some choices to make when we want to write down the rules for conjunction. Two possible pairs of sequent rules exist

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} [L\wedge] \quad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \wedge B, \Delta, \Delta'} [R\wedge]$$

$$\frac{\Gamma, A_i \vdash \Delta}{\Gamma, A_0 \wedge A_1 \vdash \Delta} [L\wedge'] \quad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} [R\wedge']$$

depending on whether in the right rules we take the union of the contexts of the two premisses or demand the contexts are the same. These choices force the corresponding left rule upon us if we want our system to satisfy cut elimination.

We will call the conjunction which takes the union of the contexts the *multiplicative* conjunction in linear logic, and write it as ' \otimes '. We will call the conjunction which demands the context formulas of the premisses to be equal the *additive* conjunction, and write it as '&'

We face the symmetrical choice for the rules for disjunction, where we will also have a multiplicative instantiation ' \wp ' and an additive instantiation ' \oplus '.

2.1 Sequent Calculus

I will now present the sequent calculus for linear logic. The group of identity rules should contain no surprises. Note that the $[Cut]$ rule is multiplicative.

$$\text{Identity}$$

$$\frac{}{A \vdash A} [Ax] \quad \frac{\Gamma, A \vdash \Delta \quad \Gamma' \vdash A, \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} [Cut]$$

Linear negation $(.)^\perp$ has the obvious rules.

$$\text{Negation}$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, A^\perp \vdash \Delta} [L^\perp] \quad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A^\perp, \Delta} [R^\perp]$$

2.1.1 Multiplicatives

For the multiplicative conjunction ' \otimes ' (tensor) and disjunction ' \wp ' (par) the context of the conclusion is the union of the contexts of the premisses of the rule. A way to look at a formula $A \otimes B$ is that it gives *both* an A and a B resource. It is difficult to come up with a similar intuition for ' \wp ' except perhaps by noting it is the De Morgan dual of ' \otimes '.

Linear implication ‘ \multimap ’ is not a primitive connective of classical linear logic, and is defined as $A \multimap B =_{\text{def}} A^\perp \wp B$. For completeness, I will present the rules for linear implication along with the other multiplicatives. In an intuitionistic setting ‘ \multimap ’ will replace ‘ \wp ’, as par makes essential use of multiple formulas on the right hand side of the sequent.

Multiplicatives

$$\begin{array}{c} \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta} [L\otimes] \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \otimes B, \Delta, \Delta'} [R\otimes] \\ \\ \frac{\Gamma, A \vdash \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \wp B \vdash \Delta, \Delta'} [L\wp] \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \wp B, \Delta} [R\wp] \\ \\ \frac{\Gamma \vdash A, \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \multimap B \vdash \Delta, \Delta'} [L\multimap] \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \multimap B, \Delta} [R\multimap] \end{array}$$

The multiplicative units $\mathbf{1}$ and \perp are the identities for \otimes and \wp respectively. $A \otimes \mathbf{1} \dashv\vdash A$, which should remind you of multiplication, and $A \wp \perp \dashv\vdash A$.

Multiplicative Units

$$\begin{array}{c} \frac{\Gamma \vdash \Delta}{\Gamma, \mathbf{1} \vdash \Delta} [L\mathbf{1}] \qquad \frac{}{\vdash \mathbf{1}} [R\mathbf{1}] \\ \\ \frac{}{\perp \vdash} [L\perp] \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \perp, \Delta} [R\perp] \end{array}$$

2.1.2 Additives

For the additives ‘ $\&$ ’ (with) and ‘ \oplus ’ (plus) the contexts of the premisses of the rule must be the same. A formula $A \& B$ represents a *choice* between A or B (as opposed to $A \otimes B$, where you will receive both A and B). Similarly $A \oplus B$ means it is unknown whether we have an A or a B resource.

Additives

$$\begin{array}{c} \frac{\Gamma, A_i \vdash \Delta}{\Gamma, A_0 \& A_1 \vdash \Delta} [L\&] \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \& B, \Delta} [R\&] \\ \\ \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta} [L\oplus] \qquad \frac{\Gamma \vdash A_i, \Delta}{\Gamma \vdash A_0 \oplus A_1, \Delta} [R\oplus] \end{array}$$

The additive units $\mathbf{0}$ and \top are the identities for \oplus and $\&$ respectively. $A \oplus \mathbf{0}$ is equivalent to A , again arithmetic serves as a mnemonic, and $A \& \top$ is equivalent to A . Note that there is no $[L\top]$ or $[R\mathbf{0}]$ rule.

Additive Units

$$\frac{}{\Gamma, \mathbf{0} \vdash \Delta} [L\mathbf{0}] \qquad \frac{}{\Gamma \vdash \top, \Delta} [R\top]$$

Another equivalence reminiscent of arithmetic, relating the multiplicatives and the additives is

$$\begin{aligned} A \otimes (B \oplus C) &\vdash (A \otimes B) \oplus (A \otimes C) \\ (A \otimes B) \oplus (A \otimes C) &\vdash A \otimes (B \oplus C) \end{aligned}$$

whereas we have a similar relation between $\&$ and \wp .

$$\begin{aligned} A \wp (B \& C) &\vdash (A \wp B) \& (A \wp C) \\ (A \wp B) \& (A \wp C) &\vdash A \wp (B \& C) \end{aligned}$$

2.1.3 Exponentials

The connectives ‘?’ (why not) and ‘!’ (bang) are called the exponentials. The idea is that on the left hand side $!A$ should be interpreted as an arbitrary number of occurrences of the formula A (we choose how many), whereas $?A$ specifies an unknown quantity of occurrences of the formula A . In the structural rules of the system this will be reflected by licensing the use of the structural rules of weakening and contraction for resources marked with an exponential, as shown below.

Under this interpretation, the left rule for ‘!’ specifies that if something is derivable with a single formula A then it is also derivable if we are allowed to use an arbitrary number of occurrences of A .

The left rule for ‘?’ would then indicate that we could accommodate for an unknown quantity of A formulas only if all context formulas can be used as many times as necessary.

Readers familiar with modal logic will recognize these logical rules as those of the modal logic **S4**. We will discuss this connection in more detail in Section 3.4, where we will introduce more fine-grained structural modalities.

Exponentials

$$\begin{aligned} \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta} [L!] & \quad \frac{! \Gamma \vdash A, ? \Delta}{! \Gamma \vdash !A, ? \Delta} [R!] \\ \frac{! \Gamma, A \vdash ? \Delta}{! \Gamma, ?A \vdash ? \Delta} [L?] & \quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash ?A, \Delta} [R?] \end{aligned}$$

Structural Rules

$$\begin{aligned} \frac{\Gamma, B, A, \Gamma' \vdash \Delta}{\Gamma, A, B, \Gamma' \vdash \Delta} [LP] & \quad \frac{\Gamma \vdash \Delta, B, A, \Delta'}{\Gamma \vdash \Delta, A, B, \Delta'} [RP] \\ \frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta} [LC] & \quad \frac{\Gamma \vdash ?A, ?A, \Delta}{\Gamma \vdash ?A, \Delta} [RC] \\ \frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta} [LW] & \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash ?A, \Delta} [RW] \end{aligned}$$

The general theme here is important: we note that certain global structural rules are too strong for our purposes, and we reintroduce controlled versions of these structural rules. This theme will return in Chapter 3.

We again have an equivalence relating the multiplicatives, additives and exponentials, reminding us of the arithmetical $2^{a+b} = 2^a \cdot 2^b$.

$$\begin{array}{l} !(A \& B) \vdash !A \otimes !B \\ !A \otimes !B \vdash !(A \& B) \\ ?(A \oplus B) \vdash ?A \wp ?B \\ ?A \wp ?B \vdash ?(A \oplus B) \end{array}$$

2.1.4 Quantifiers

The first order quantifiers, which quantify over entities in our model, and second order quantifiers, which quantify over formulas in our logic can be added to propositional linear logic to produce first and second order linear logic.

First Order Quantifiers

$$\begin{array}{l} \frac{\Gamma, A[x := v] \vdash \Delta}{\Gamma, \forall x. A \vdash \Delta} [L\forall] \quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \forall x. A, \Delta} [R\forall] \\ \frac{\Gamma, A \vdash \Delta}{\Gamma, \exists x. A \vdash \Delta} [L\exists] \quad \frac{\Gamma \vdash A[x := v], \Delta}{\Gamma \vdash \exists x. A, \Delta} [R\exists] \end{array}$$

Second Order Quantifiers

$$\begin{array}{l} \frac{\Gamma, A[X := B] \vdash \Delta}{\Gamma, \forall X. A \vdash \Delta} [L\forall] \quad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \forall X. A, \Delta} [R\forall] \\ \frac{\Gamma, A \vdash \Delta}{\Gamma, \exists X. A \vdash \Delta} [L\exists] \quad \frac{\Gamma \vdash A[X := B], \Delta}{\Gamma \vdash \exists X. A, \Delta} [R\exists] \end{array}$$

Both the first and the second order rules for $[R\forall]$ and $[L\exists]$ have the condition that there is no free occurrence of x (resp. X) in Γ or Δ .

2.1.5 Cut Elimination

One of the most important questions to ask of any logical system is the question whether it is consistent, or in other words if there are sequents which are *unprovable* in the logic.

Suppose, for example, there exists a proof \mathcal{D} of the sequent $\vdash 0$. We could extend this proof as follows.

$$\frac{\frac{\Gamma, 0 \vdash \Delta}{\Gamma \vdash \Delta} [L0] \quad \begin{array}{c} \vdots \\ \mathcal{D} \\ \vdots \\ \vdash 0 \end{array}}{\Gamma \vdash \Delta} [Cut]$$

Therefore, if there exists a proof of $\vdash 0$ every sequent would be derivable in the sequent calculus, and our logic would be inconsistent. But if we look at the hypothetical proof \mathcal{D} , which rule could be the last rule in this proof? There is no $[R0]$ rule and all other logical rules introduce logical connectives other than 0 . None of the structural rules apply either, as $\vdash 0$ is not a valid conclusion for any of those rules. This means that the last rule must have been $[Cut]$. So if we are able to eliminate the $[Cut]$ rule we thereby show the non-existence of a proof of $\vdash 0$.

Another way to look at consistency is to say it is impossible to have both a proof of a formula A and a proof of the negation A^\perp of that formula. If we have a proof \mathcal{D}_1 of A and a proof \mathcal{D}_2 of A^\perp we can combine them to prove the empty sequent and finally to conclude $\vdash \perp$.

$$\frac{\frac{\frac{\vdots \mathcal{D}_1}{\vdash A} \quad \frac{\frac{\vdots \mathcal{D}_2}{\vdash A^\perp} [R^\perp]}{A \vdash} [Cut]}{\vdash} [R_\perp]}{\vdash \perp} [R_\perp]$$

Here, again, to prove the sequent $\vdash \perp$ without using the cut rule, the last rule must have been $[R_\perp]$, but none of the sequent rules except $[Cut]$ can have the empty sequent as its conclusion. So eliminating the cut rule guarantees proofs \mathcal{D}_1 and \mathcal{D}_2 cannot both exist.

If we can prove that the cut rule is redundant, then we prove our calculus is consistent as well. Gentzen (1934) introduced the carefully symmetric sequent calculus specifically for proving cut elimination and his original proof can be adapted to linear logic as well, as stated by the following theorem.

Theorem 2.1 *The cut rule is redundant in the sequent calculus for linear logic.*

We will not prove this theorem here, but refer the reader to Roorda (1991) or Troelstra (1992) for a proof of the cut elimination theorem, together with a proof of strong normalization of the cut elimination operation. I will merely give an example of the important cases.

In the case of a trivial cut, where one of the premisses of the $[Cut]$ rule was obtained by an axiom, we can eliminate the cut entirely, replacing

$$\frac{\frac{}{A \vdash A} [Ax] \quad \frac{\vdots \mathcal{D}}{\Gamma \vdash A, \Delta} [Cut]}{\Gamma \vdash A, \Delta} [Cut]$$

by

$$\frac{\vdots \mathcal{D}}{\Gamma \vdash A, \Delta}$$

and symmetrically if the other premiss was obtained by an axiom.

| | |
|-------------|---|
| LL | Full Propositional Linear logic |
| MLL | Multiplicative Linear Logic |
| MALL | Multiplicative Additive Linear Logic |
| MELL | Multiplicative Exponential Linear Logic |
| 1 | Suffix Indicating the First Order Fragment |
| 2 | Suffix Indicating the Second Order Fragment |

Table 2.1: The different fragments of linear logic

Many of the cases of proving the theorem involve changing the order in which rules are applied, but the crucial case occurs when the cut formula is the main formula of both rules which provide the premisses to the $[Cut]$ rule. In the case of ‘ \otimes ’ this will look as follows.

$$\frac{\frac{\frac{\vdots \mathcal{D}_1}{\Gamma, A, B \vdash \Delta} [L\otimes] \quad \frac{\frac{\vdots \mathcal{D}_2}{\Gamma' \vdash A, \Delta'} \quad \frac{\vdots \mathcal{D}_3}{\Gamma'' \vdash B, \Delta''}}{\Gamma', \Gamma'' \vdash A \otimes B, \Delta', \Delta''} [R\otimes]}{\Gamma, \Gamma', \Gamma'' \vdash \Delta, \Delta', \Delta''} [Cut]}$$

Now, we can replace the cut on $A \otimes B$ by two simpler cuts: one on A and one on B , as follows.

$$\frac{\frac{\frac{\vdots \mathcal{D}_1}{\Gamma, A, B \vdash \Delta} \quad \frac{\vdots \mathcal{D}_2}{\Gamma' \vdash A, \Delta'}}{\Gamma, \Gamma', B \vdash \Delta, \Delta'} [Cut] \quad \frac{\vdots \mathcal{D}_3}{\Gamma'' \vdash B, \Delta''} [Cut]}{\Gamma, \Gamma', \Gamma'' \vdash \Delta, \Delta', \Delta''} [Cut]}$$

Corollary 2.2 *Linear logic is consistent.*

Apart from consistency, another important consequence of cut elimination is the *subformula property*, which states that for a sequent proof we only need to use subformulas of the end-sequent. Observing the sequent rules, we see that for all rules except $[Cut]$ the premisses of the rules use only subformulas of the conclusion. From a computational point of view, this is very interesting in that it restricts the sequents we have to consider when trying to prove a sequent. However, cut elimination alone is insufficient to prove decidability of the calculus. For the quantifiers, for example, we might have to consider an infinite number of subformulas. In Section 2.6 we will discuss the computational complexity of linear logic.

2.2 Fragments of Linear Logic

Often, we will be interested in *fragments* of linear logic. A number of fragments and their abbreviations are shown in Table 2.1.

Sometimes, the intuitionistic fragment of a logic is indicated by an **I** infix. For example, **MILL2** is the second order fragment of multiplicative, intuitionistic linear logic.

2.3 One Sided Sequent Calculus

Proposition 2.3 *The following formulas are derivably equivalent.*

$$\begin{aligned}
\mathbf{1}^\perp &= \perp \\
\perp^\perp &= \mathbf{1} \\
\top^\perp &= \mathbf{0} \\
\mathbf{0}^\perp &= \top \\
(A^\perp)^\perp &= A \\
(A \otimes B)^\perp &= A^\perp \wp B^\perp \\
(A \wp B)^\perp &= A^\perp \otimes B^\perp \\
(A \& B)^\perp &= A^\perp \oplus B^\perp \\
(A \oplus B)^\perp &= A^\perp \& B^\perp \\
(!A)^\perp &= ?A^\perp \\
(?A)^\perp &= !A^\perp \\
(\forall x.A)^\perp &= \exists x.A^\perp \\
(\exists x.A)^\perp &= \forall x.A^\perp
\end{aligned}$$

Proof Trivial. □

Because of the properties of negation given in proposition 2.3 (De Morgan dualities, elimination of double negation) we can restrict negation to atomic formulas, and move all formulas to the right hand side of the sequent. This is just a matter of economy: a syntactic manipulation to reduce the number of rules. The one sided sequent calculus is shown in Table 2.2 on the next page.

In the $[Ax]$ and $[Cut]$ rules the formula A^\perp refers to the formula A with the outer negation distributed to the atomic formulas and double negations removed according to the equivalences of Proposition 2.3. So the negation in these rules is a *defined* operation.

A valid instance of the axiom rule would be, for example

$$\frac{}{\vdash (a^\perp \oplus ?b) \otimes c, (a \& !b^\perp) \wp c^\perp} [Ax]$$

2.4 Intuitionistic Linear Logic

To obtain the intuitionistic sequent calculus for linear logic from the classical formulation we simply restrict the succedent to contain at most one formula. This means the connectives ‘?’ and ‘ \wp ’, which make essential use of multiple succedent formulas in their sequent rules, need to be dropped.

$$\begin{array}{c}
\frac{}{\vdash A, A^\perp} [Ax] \quad \frac{\vdash \Gamma, A \quad \vdash A^\perp, \Delta}{\vdash \Gamma, \Delta} [Cut] \\
\frac{\vdash \Gamma, B, A, \Delta}{\vdash \Gamma, A, B, \Delta} [P] \\
\frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A} [C] \quad \frac{\vdash \Gamma}{\vdash \Gamma, ?A} [W] \\
\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} [!] \quad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A} [?] \\
\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B} [\otimes] \quad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \wp B} [\wp] \\
\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \& B} [\&] \quad \frac{\vdash \Gamma, A_i}{\vdash \Gamma, A_0 \oplus A_1} [\oplus]
\end{array}$$

Table 2.2: One sided sequent calculus for linear logic

In fact, it turns out to be natural to restrict intuitionistic linear logic to sequents with *exactly* one succedent formula, which means linear negation ‘ $(.)^\perp$ ’ and the constant ‘ \perp ’ are also out. This leaves us with the calculus shown in Table 2.3 on the following page for (propositional) **ILL**.

2.4.1 Natural Deduction

The multiplicative, intuitionistic fragment of linear logic has an alternative formulation to the sequent calculus. This is the *natural deduction* formulation of multiplicative, intuitionistic linear logic, which are a natural restriction of Gentzen’s (1934) natural deduction rules for intuitionistic logic. In a sense, the proof nets introduced for multiplicative linear logic in Chapter 4 are the classical counterpart of natural deduction.

Natural deduction proofs are tree-like structures where the conclusion is the root of the tree and the hypotheses are the leaves. Instead of sequent left (resp. right) rules we will have elimination (resp. introduction) rules for our connectives.

Leaves can be either ‘active’ or ‘inactive’, some of the rules ‘discharging’ a single active hypothesis occurrence, for example

$$\frac{\begin{array}{c} [A]^n \\ \vdots \\ B \end{array}}{A \multimap B} [\multimap I]^n$$

where the bracketing indicates the hypothesis A is no longer active beyond the rule with which it is coindexed.

Multiplicatives

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} [L\otimes] \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} [R\otimes]$$

$$\frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} [L\multimap] \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} [R\multimap]$$

Multiplicative Unit

$$\frac{\Gamma \vdash C}{\Gamma, \mathbf{1} \vdash C} [L\mathbf{1}] \quad \frac{}{\vdash \mathbf{1}} [R\mathbf{1}]$$

Additives

$$\frac{\Gamma, A_i \vdash C}{\Gamma, A_0 \& A_1 \vdash C} [L\&] \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} [R\&]$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C} [L\oplus] \quad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_0 \oplus A_1} [R\oplus]$$

Additive Units

$$\frac{}{\Gamma, \mathbf{0} \vdash C} [L\mathbf{0}] \quad \frac{}{\Gamma \vdash \top} [R\top]$$

Exponentials

$$\frac{\Gamma, A \vdash C}{\Gamma, !A \vdash C} [L!] \quad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A} [R!]$$

Structural Rules

$$\frac{\Gamma, B, A, \Delta \vdash C}{\Gamma, A, B, \Delta \vdash C} [LP]$$

$$\frac{\Gamma, !A, !A \vdash C}{\Gamma, !A \vdash C} [LC] \quad \frac{\Gamma \vdash C}{\Gamma, !A \vdash C} [LW]$$

Table 2.3: Sequent calculus for intuitionistic linear logic

If there is a natural deduction proof with undischarged leaves Γ for A , we will write this as $\Gamma \vdash A$ or as a tree like

$$\begin{array}{c} \Gamma \\ \vdots \\ \vdots \\ A \end{array}$$

The natural deduction rules for multiplicative intuitionistic linear logic are shown in Table 2.4 on the next page.

The square brackets in the $[\multimap I]$ and $[\otimes E]$ rules indicate these rules dis-

| | |
|---|---|
| Hypothesis | |
| A | |
| Logical Rules | |
| $\frac{A \quad A \multimap B}{B} [\multimap E]$ | $\frac{[A]^i \quad \dots \quad B}{A \multimap B} [\multimap I]^i$ |
| $\frac{A \otimes B \quad C}{C} [\otimes E]^i$ | $\frac{A \quad B}{A \otimes B} [\otimes I]$ |

Table 2.4: The natural deduction calculus **MILL**

charge exactly one occurrence of the named formulas.

The $[\otimes E]$ rule appears a bit strange: in intuitionistic logic, there are two $[\wedge E]$ rules, which look as follows.

$$\frac{A \wedge B}{A} [\wedge E] \quad \frac{A \wedge B}{B} [\wedge E]$$

From the linear point of view, however, these would be the natural deduction rules for the additive conjunction ‘&’. What we would like to write for the multiplicative conjunction would be the following rule

$$\frac{A \otimes B}{A \quad B} [\otimes E']$$

but this rule does not conform to the tree-like structure of natural deduction proofs. Comparing rule $[\otimes E']$ to rule $[\otimes E]$, we see that the latter is merely a way of writing the former rule in such a way that it respects the structure of natural deduction proofs. We will see in Chapter 7 that moving to a calculus with multiple conclusions allows us to give a rule very similar to the $[\otimes E']$ rule above. For the remainder of this chapter, however, we will use the $[\otimes E]$ rule.

2.4.2 The Curry-Howard Isomorphism

The main interest of natural deduction lies in the Curry-Howard isomorphism; by interpreting functional types as implicational formulas and pairing types as conjunctive formulas, we can see that constructing a term t of type A in the simply typed lambda calculus corresponds to finding a natural deduction proof \mathcal{D} of the formula corresponding to A .

We will make this a bit more precise below

Definition 2.4 (Types) *The types are defined inductively as follows*

- (i) *we have a finite number T_1, \dots, T_n of atomic types.*
- (ii) *if T and U are types, $T \rightarrow U$ is a type (function space).*
- (iii) *if T and U are types, $T \times U$ is a type (Cartesian product).*

Atomic formulas correspond to atomic types, the implication ‘ \multimap ’ will correspond to ‘ \rightarrow ’, and the product ‘ \otimes ’ will correspond to ‘ \times ’.

We can now define the set of linear lambda terms as follows

Definition 2.5 (Linear Lambda Terms) *The linear lambda terms are defined inductively as follows*

- (i) *for each type T we have a (countably infinite) supply x, y, z, \dots of variables of that type.*
- (ii) *if t is a term of type $T \rightarrow U$ and u is a term of type T then $(t u)$ is a term of type U .*
- (iii) *if t is a term of type U and x is a term of type T which occurs exactly once in t , then $\lambda x.t$ is a term of type $T \rightarrow U$.*
- (iv) *if u is a term of type $U \times V$ and t is a term of type T in which a term x of type U and a term y of type V occur exactly once then t with $\pi^1 u$ substituted for x and $\pi^2 u$ substituted for y is also a term of type T .*
- (v) *if t is a term of type T and u is a term of type U then $\langle u, v \rangle$ is a term of type $T \times U$.*

An alternative way to define the set of linear lambda terms is given in (Abramsky 1993).

Just as the set of MILL proofs is a proper subset of the set of intuitionistic proofs, the set of linear lambda terms is a proper subset of the set of lambda terms. We get the usual definition of lambda terms by dropping the ‘occurs exactly once’ restriction of case (iii) and case (iv). Case (iv) without this restriction is equivalent to the more familiar.

- (iv) *if u is a term of type $U \times V$ then $\pi^1 u$ is a term of type U and $\pi^2 u$ is a term of type V .*

We will make the isomorphism fully explicit by presenting a ‘semantically’ labeled version of the natural deduction calculus from the previous section in Table 2.5 on the facing page, where the term operations mirror the rules. x, y, z denote fresh variables and t, u, v denote arbitrary lambda term labels. A term of the form $t[x := u]$ corresponds to the term t where the term u is substituted for all free occurrences of the variable x .

For the sequent calculus we don’t have such a 1–1 correspondence between proofs and lambda terms. The following two sequent proofs, for example

$$\begin{array}{c}
\textbf{Hypothesis} \\
x : A \\
\textbf{Logical Rules} \\
\frac{[x : A]^n \quad [y : B]^n \quad \vdots}{\frac{u : A \otimes B \quad t : C}{t[x := \pi^1 u, y := \pi^2 u] : C} [\otimes E]^n} [\otimes I] \quad \frac{t : A \quad u : B}{\langle t, u \rangle : A \otimes B} [\otimes I] \\
\frac{t : A \multimap B \quad u : A}{(tu) : B} [-\circ E] \quad \frac{[x : A]^n \quad \vdots \quad t : B}{\lambda x.t : A \multimap B} [-\circ I]^n
\end{array}$$

Table 2.5: The natural deduction calculus **MILL** with semantic labeling

$$\frac{\frac{B \vdash B \quad C \vdash C}{B, B \multimap C \vdash C} [L-\circ] \quad A \vdash A}{A, A \multimap B, B \multimap C \vdash C} [L-\circ] \quad \frac{\frac{A \vdash A \quad B \vdash B}{A, A \multimap B \vdash B} [L-\circ] \quad C \vdash C}{A, A \multimap B, B \multimap C \vdash C} [L-\circ]$$

translate to the same natural deduction proof

$$\frac{\frac{A \quad A \multimap B}{B} [-\circ E] \quad B \multimap C}{C} [-\circ E]$$

2.4.3 Normalisation

For the linear lambda terms we have the following set of equivalences, a beta and eta equivalence for each type. The equivalences for the implicational types have the restriction of t being free for u resp. x . We can always meet this restriction by renaming variables when conflicts occur.

$$\begin{array}{l}
(\lambda x.t)u =_{\beta} t[x := u] \quad \lambda x.(tx) =_{\eta} t \\
\pi^i \langle t_1, t_2 \rangle =_{\beta} t_i \quad \langle \pi^1 t, \pi^2 t \rangle =_{\eta} t
\end{array}$$

Though equivalences, we will usually apply them from left to right as asymmetric *conversions*, in which case we will call the left hand side of the equivalence a *redex* and the right hand side its *contractum*.

Definition 2.6 (Reduction) We will say a term t converts to a term u if it can be obtained by replacing a subterm which is a redex by its contractum.

We will say a term t reduces to a term u ($t \rightsquigarrow u$) if it can be obtained by zero or more conversion steps from t .

Definition 2.7 (Normal form) A term is beta normal or just normal iff it does not contain any beta redexes.

If a term t reduces to a term u and u is normal, we will call u a normal form of t .

Proposition 2.8 For every term t there exists a normal form u such that $t \rightsquigarrow u$, moreover if $t \rightsquigarrow u'$ and u' is normal then we have $u \equiv u'$.

We will not prove this proposition here, but refer the interested reader to (Girard, Lafont & Taylor 1988).

According to the Curry-Howard isomorphism, the conversions on terms correspond to conversions on proofs. For example, beta conversion for the implicational types corresponds to replacing the proof

$$\frac{\frac{[x : B]^k \quad \vdots \mathcal{D}_2}{t : A} \quad \frac{[\neg \circ I]^k}{(\lambda x.t)u : A} \quad \frac{\vdots \mathcal{D}_1}{u : B}}{[\neg \circ E]}$$

by a (simpler) proof, where instead of hypothesising a proof of B , we use proof \mathcal{D}_1 of B directly

$$\frac{\vdots \mathcal{D}_1}{u : B} \quad \frac{\vdots \mathcal{D}_2}{t[x := u] : A}$$

We will call a natural deduction proof *normal* iff its corresponding lambda term is normal. In (Girard et al. 1988, p41), Girard calls normal natural deduction proofs ‘morally equivalent’ to cut free sequent proofs, and indeed many of the pleasant properties of the cut free sequent calculus, such as the subformula property and consistency have corresponding notions in natural deduction.

2.5 Linguistic Applications

In this section, I will present an overview of the linguistic applications of the various connectives of linear logic. We will note that most of the groups suggest at least some possible application to linguistic phenomena.

2.5.1 Multiplicatives

The core of our linguistic applications, and the main focus of this book are the multiplicatives.

Using s as the formula expressing a sentence, np as the formula expressing a noun phrase and n as the formulas representing a common noun, we can use linear implication to assign formulas to words, for example like.

For example a verb like ‘to believe’ can express a relation between two persons, np ’s in our interpretation, or between a person and a statement, an s in our interpretation, as indicated by the following examples.

(2.3) Sollozzo believes Vito.

(2.4) Sollozzo believes Vito trusts him.

We can express this by two lexical assignments as follows

$$\begin{aligned} l(\text{believes}) &= np \multimap (np \multimap s) \\ l(\text{believes}) &= s \multimap (np \multimap s) \end{aligned}$$

or we can collapse the two assignment to the single.

$$l(\text{believes}) = np \multimap (np \multimap s) \ \& \ s \multimap (np \multimap s)$$

But now we can use the following theorem

$$\frac{\frac{\frac{\overline{A \vdash A} \ [Ax] \quad \overline{C \vdash C} \ [Ax]}{A, A \multimap C \vdash C} \ [L \multimap]}{A, (A \multimap C) \& (B \multimap C) \vdash C} \ [L \&]}{\frac{\frac{\overline{B \vdash B} \ [Ax] \quad \overline{C \vdash C} \ [Ax]}{B, B \multimap C \vdash C} \ [L \multimap]}{B, (A \multimap C) \& (B \multimap C) \vdash C} \ [L \&]}{A \oplus B, (A \multimap C) \& (B \multimap C) \vdash C} \ [R \oplus]}{(A \multimap C) \& (B \multimap C) \vdash (A \oplus B) \multimap C} \ [R \multimap]}$$

to move the additive ‘inside’ and generate a more compact lexical entry, where the part of the initial two assignments which is identical is shared.

$$l(\text{believes}) = (s \oplus np) \multimap (np \multimap s)$$

Note that from the point of view of the one sided sequent calculus all examples use ‘ \oplus ’. It is unclear if we can find a similar use for ‘ $\&$ ’.

A second use of additives is given by Bayer & Johnson (1995) and which is also used by Dörre & Manandhar (1995). They make use of the additives to represent grammatical features, operating on the restriction that additives can never have scope over multiplicatives, only over other additives and atomic formulas.

As an example, assume we add atomic formulas $1st$, $2nd$ and $3rd$, indicating first, second and third person, and atomic formulas sng and plr indicating singular and plural, we can give a lexicon with features as follows.

$$\begin{aligned} l(I) &= np \& (1st \& sng) \\ l(\text{we}) &= np \& (1st \& plr) \\ l(\text{you}) &= np \& (1st \& (sng \oplus plr)) \\ l(\text{Luca}) &= np \& (3rd \& sng) \\ l(\text{sleeps}) &= np \& (3rd \& sng) \multimap s \\ l(\text{sleep}) &= np \& ((1st \oplus 2nd) \oplus plr) \multimap s \end{aligned}$$

Now we can derive ‘I sleep’ as follows.

The basic idea is that an np island is represented by the formula $!np$. Because $!np \vdash np$ an np island can function as a normal np , but because $np \not\vdash !np$ a normal np isn't necessarily an np island.

Now if we use the following lexical assignments, where we give some complex expressions a single entry for the sake of simplicity

$$\begin{aligned} l(\text{cannoli}) &= n \\ l(\text{is simple}) &= !np \multimap s \\ l(\text{tastes good}) &= !np \multimap s \\ l(\text{the cooking of}) &= np \multimap np \\ l(\text{which}) &= (!np \multimap s) \multimap (n \multimap n) \end{aligned}$$

we can derive the first but not the second example, as required. We can derive 'tastes good' as an $!np \multimap s$ by simply using the axiom rule. On the other hand 'the cooking of is simple' cannot be derived as an $!np \multimap s$, as demonstrated by the proof attempt shown below.

$$\frac{\frac{\frac{}{np \vdash np} [Ax]}{!np \vdash np} [L!]}{\frac{}{np \vdash !np} [Ax]} \text{ FAIL} \quad \frac{\frac{}{s \vdash s} [Ax]}{np, !np \multimap s \vdash s} [L \multimap]}{!np, np \multimap np, !np \multimap s \vdash s} [L \multimap]}{np \multimap np, !np \multimap s \vdash !np \multimap s} [R \multimap]}$$

2.5.5 Quantifiers

The first order quantifiers provide us with yet another way of encoding linguistic features. We will discuss more linguistic applications of the first order quantifiers in Chapter 5, where the arguments of predicates will be used to encode string positions.

The second order quantifiers give us a way of encoding so-called 'polymorphic' words, like 'and' and 'or', which can combine almost any two expressions into a single expression of the same type.

(2.7) Michael and Sollozzo meet.

(2.8) Vito stumbles and falls.

(2.9) A dangerous and influential man.

Given the following lexicon

$$\begin{aligned} l(\text{and}) &= \forall X. X \multimap (X \multimap X) \\ l(\text{falls}) &= np \multimap s \\ l(\text{or}) &= \forall X. X \multimap (X \multimap X) \\ l(\text{stumbles}) &= np \multimap s \\ l(\text{vito}) &= np \end{aligned}$$

we can derive 'Vito stumbles and falls' as follows.

But this would be a proof of ‘An buys Vito orange’, which is clearly ungrammatical. Even languages with relatively free word order are not globally commutative. So the rule of commutativity, if we need it, needs to be restricted to certain formulas, as already done with the structural rules of weakening and contraction.

There is a final pair of structural rules in linear logic, which is usually treated even more implicitly than the commutativity rule. It is the rule of associativity, which states the the structure imposed by the comma in the sequent calculus is irrelevant.

$$\frac{\Gamma[(\Delta_1, (\Delta_2, \Delta_3))] \vdash \Gamma'}{\Gamma[(\Delta_1, \Delta_2), \Delta_3] \vdash \Gamma'} [LAss1] \quad \frac{\Gamma[(\Delta_1, \Delta_2), \Delta_3] \vdash \Gamma'}{\Gamma[(\Delta_1, (\Delta_2, \Delta_3))] \vdash \Gamma'} [LAss2]$$

We will discuss non-associative logics in more detail in Section 3.2.

2.5.7 Semantics

The Curry-Howard isomorphism, as discussed in Section 2.4.2, is a correspondence between natural deduction proofs and linear lambda terms. Montague (1974) used first order intensional logic, which included the lambda operator, for his compositional semantics of natural language and stipulated a semantic rule for every syntactic rule in his system. But, as noted by van Benthem (1987) in the context of proofs for the Lambek calculus with permutation, we can use the Curry-Howard interpretation of natural deduction proofs to generate the semantics of a derivation and afterwards substitute the lexical meaning recipes. This makes the syntax-semantics interface completely regular and nonstipulative. Also, we can use any typed logical language which includes the lambda operator as our semantic language. For example, Muskens (1994) shows a type theoretic implementation of Discourse Representation Theory (Kamp & Reyle 1993).

Example 2.9 *Given the lexicon below, where we have given every word a lexical semantics in addition to a linear logic formula.*

$$\begin{aligned} l(\mathit{shot}) &= np \multimap (np \multimap s) - \lambda v. \lambda w. \mathit{shot}(v, w) \\ l(\mathit{someone}) &= (np \multimap s) \multimap s - \lambda x. \exists y. (x y) \\ l(\mathit{vito}) &= np - \mathit{vito} \end{aligned}$$

Note the difference between a regular np , like ‘Vito’ and a generalized quantifier quantifier like ‘someone’. Because we want the generalized quantifier ‘someone’ to take scope at sentence level, we assign it the type $(np \multimap s) \multimap s$. In this way semantic intuitions, together with the Curry-Howard isomorphism, help us to assign the right formulas to words in the lexicon. Conversely, the formulas correspond directly to the semantics types, which helps us in assigning the right lambda term semantics to words in our lexicon.

Now, consider the proof of ‘someone shot Vito’, in natural deduction format labeled with semantic terms, below.

$$\frac{\frac{[x : np]^1 \quad \frac{z : np \quad y : np \multimap (np \multimap s)}{(y z) : np \multimap s} [-\multimap E]}{((y z) x) : s} [-\multimap E]}{\lambda x.((y z) x) : np \multimap s} [-\multimap I]^1 \quad v : (np \multimap s) \multimap s} [-\multimap E]}{(v \lambda x.((y z) x)) : s} [-\multimap E]$$

Substituting **vito** for x , $\lambda v.\lambda w.\mathbf{shot}(v, w)$ for y and $\lambda x.\exists y.(x y)$ for v .

$$\begin{aligned} (\lambda x.\exists y.(x y) \lambda z.((\lambda v.\lambda w.\mathbf{shot}(v, w) z) \mathbf{vito})) &=_{\beta} \\ (\lambda x.\exists y.(x y) \lambda z.(\lambda w.\mathbf{shot}(z, w) \mathbf{vito})) &=_{\beta} \\ (\lambda x.\exists y.(x y) \lambda z.\mathbf{shot}(z, \mathbf{vito})) &=_{\beta} \\ \exists y.(\lambda z.\mathbf{shot}(z, \mathbf{vito}) y) &=_{\beta} \\ \exists y.\mathbf{shot}(y, \mathbf{vito}) \end{aligned}$$

There is an alternative natural deduction proof for the sentence ‘someone shot Vito’, which is presented below.

$$\frac{\frac{[z : np]^1 \quad y : np \multimap (np \multimap s)}{(y z) : np \multimap s} [-\multimap E]}{((y z) x) : s} [-\multimap E]}{\lambda z.((y z) x) : np \multimap s} [-\multimap I]^1 \quad v : (np \multimap s) \multimap s} [-\multimap E]}{(v \lambda z.((y z) x)) : s} [-\multimap E]$$

The only difference with the previous proof is that the $[-\multimap I]$ rule discharges the other np formula. The resulting lambda term for this proof, after substitution and beta reduction, would be

$$\exists y.\mathbf{shot}(\mathbf{vito}, y)$$

which we would expect as the semantics of ‘Vito shot someone’. It turns out that the problem here is the structural rule of commutativity, which is implicit in the natural deduction formulation we have given. In Section 3.5, we will present a natural deduction calculus which is explicit about the structural rules of associativity and commutativity.

An interesting alternative take on natural language semantics and linear logic is given by de Groote & Retoré (1996), where instead of performing a substitution with lexical semantics, the lexical semantics is incorporated directly as a separate derivation which is connected to the rest of the proof by means of the cut rule.

| | |
|-------------------|------------------|
| NP Complete | MLL, MLL1 |
| PSPACE Complete | MALL |
| NEXPTIME Complete | MALL1 |
| EXSPACE Hard | MELL |
| RE Complete | LL, MLL2 |

Table 2.6: Complexity results for different fragments of linear logic

2.6 Complexity

Now that we have seen linguistic applications of all families of connectives in linear logic, we will look at the computational complexity of the different fragments of linear logic.

There have been a number of results on the complexity of various fragments of linear logic. An overview is given by Lincoln (1995). Table 2.6 summarizes the most important results.

At the moment, the only big unknown is **MELL** for which only a lower bound has been proven by encoding the reachability problem of Petri nets in it.

The NP Completeness result of **MLL** was proved by Kanovich (1991), and also holds for the intuitionistic and the Horn clause fragment. This means that it is unlikely we will find efficient theorem proving algorithms even for this simplest fragment of linear logic.

Adding the additives moves us up to a PSPACE complete problem, while adding the first order quantifiers will cause to logic to remain NP complete in the multiplicative case, but move us to NEXPTIME completeness in the multiplicative, additive case.

The exponentials and the second order quantifiers are computationally quite powerful, the former making the logic EXSPACE hard in the multiplicative case and undecidable in the multiplicative, additive case, the latter already making the logic undecidable in the multiplicative case.

For our linguistic applications, we will be somewhat conservative with respect to the complexity of the logics we will use. It is desirable to use a logic which is in PSPACE, as this would not allow our logic to be overly expressive. Some computational linguists include polynomial parsability as a demand on linguistic theory. I will confront this claim in Chapter 10, where I will give a logical fragment for which derivability is decidable in polynomial time.

For the remainder of this thesis, however, I will be perfectly happy to use logics for which the decision procedures are in NP or in PSPACE. This is already a restriction, and one which will probably make it difficult for our logic to deal with parasitic gapping and polymorphism, phenomena which seem to require the exponentials and the second order quantifiers respectively. It is unclear if we can restrict the exponentials in such a way that the decision procedure is in PSPACE, but with respect to the second order quantifiers, Perrier (1999) gives a PSPACE complete fragment of **MALL2**, which allows

at least the polymorphic cases we have discussed in Section 2.5.5.

2.7 Conclusions

Looking back, we see that by restricting the structural rules of contraction and weakening, which apply freely in classical logic, we obtain a resource sensitive logic, which lends itself well to linguistic applications. Most fragments of linear logic suggest at least some linguistic phenomenon to which they can be applied. However, a number of these fragments use computationally heavy machinery.

The multiplicative intuitionistic fragment already allows us to capture a large set of linguistic data, provided we restrict the structural rules even further. This will be the main topic of the next chapter.

CHAPTER 3

LAMBEK CALCULI

WE have seen in the previous chapter how the various fragments of linear logic could be applied to linguistic purposes and what some of the problems were with the logic as formulated.

In this chapter, we will discuss Lambek calculi, relate them to the multiplicative intuitionistic fragment of linear logic, and show how using them solves many of the overgeneration problems we previously encountered.

3.1 Lambek Calculus

One of the main problems of applying linear logic to linguistics is the global availability of the permutation rule. It makes the rather incorrect predication that every natural language is closed under permutation. So, the commutativity rule should at least be restricted, much like contraction and weakening are restricted in linear logic.

When we drop commutativity, we are immediately faced with the fact that it is implicit in many of the sequent rules of multiplicative linear logic. For example, the $[L-\circ]$ rule, repeated below, assumes that the main formula of a rule can always be moved to the rightmost position of the antecedent, which we can generally not demand in a non-commutative setting. The $[R-\circ]$ rule on the other hand, assumes that the position of the active formula A with respect to the antecedent Γ is irrelevant.

$$\frac{\Delta \vdash A \quad \Gamma, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} [L-\circ] \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} [R-\circ]$$

Let's present two alternative formulations for the sequent rules for linear implication.

$$\begin{array}{c}
\frac{}{A \vdash A} [Ax] \quad \frac{\Delta \vdash A \quad \Gamma, A, \Gamma' \vdash C}{\Gamma, \Delta, \Gamma' \vdash C} [Cut] \\
\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \bullet B, \Delta \vdash C} [L\bullet] \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \bullet B} [R\bullet] \\
\frac{\Delta \vdash B \quad \Gamma, A, \Gamma' \vdash C}{\Gamma, A/B, \Delta, \Gamma' \vdash C} [L/] \quad \frac{\Gamma, B \vdash A}{\Gamma \vdash A/B} [R/] \\
\frac{\Delta \vdash B \quad \Gamma, A, \Gamma' \vdash C}{\Gamma, \Delta, B \setminus A, \Gamma' \vdash C} [L\setminus] \quad \frac{B, \Gamma \vdash A}{\Gamma \vdash B \setminus A} [R\setminus]
\end{array}$$

Table 3.1: The sequent calculus **L**

$$\begin{array}{c}
\frac{\Delta \vdash A \quad \Gamma, B, \Gamma' \vdash C}{\Gamma, B \circ - A, \Delta, \Gamma' \vdash C} [L\circ-] \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash B \circ - A} [R\circ-] \\
\frac{\Delta \vdash A \quad \Gamma, B, \Gamma' \vdash C}{\Gamma, \Delta, A \circ - B, \Gamma' \vdash C} [L\circ-] \quad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \circ - B} [R\circ-]
\end{array}$$

Under global commutativity all left and right rules presented so far are equivalent. Without it, $A \circ - B$, with the left rule shown above, looks for its A argument to the left, while $B \circ - A$ looks for its argument to the right. When we write $A \setminus B$ instead of $A \circ - B$, A/B instead of $A \circ - B$ and $A \bullet B$ instead of $A \otimes B$, the logical rules for non-commutative multiplicative intuitionistic linear logic are shown in Table 3.1.

Interestingly, this logic is known as the Lambek calculus **L** (Lambek 1958) and it precedes linear logic by many years. Lambek proved cut elimination for his calculus as well and this result holds for all of the extensions to the original calculus discussed in this chapter. This means the Lambek calculus is consistent, has the subformula property and, as a consequence of the subformula property and the absence of contraction and weakening, that the Lambek calculus is decidable. We will look at an alternate proof of cut elimination for Lambek calculi in Section 7.6.

Lambek's formulation of the calculus included the additional restriction that antecedents cannot be empty. Though from a logical point of view the derivability of $\vdash a \circ - a$ or $\vdash a/a$ makes perfect sense, there are linguistic reasons to reject it.

Example 3.1 *Given the lexicon of Table 3.2 on the facing page we can derive the sentence 'Ripley saves a colonist' as shown in Figure 3.1 on the next page.*

Some characteristic theorems of multiplicative linear logic are not derivable in **L**, for example.

$$\begin{aligned}
l(\text{ripley}) &= np \\
l(\text{bishop}) &= np \\
l(\text{newt}) &= np \\
l(\text{colonist}) &= n \\
l(\text{a}) &= np/n \\
l(\text{sleeps}) &= np \setminus s \\
l(\text{likes}) &= (np \setminus s)/np \\
l(\text{saves}) &= (np \setminus s)/np
\end{aligned}$$

Table 3.2: Example L lexicon

$$\frac{\frac{\frac{}{np \vdash np} [Ax] \quad \frac{}{s \vdash s} [Ax]}{np, np \setminus s \vdash s} [L\setminus] \quad \frac{\frac{}{n \vdash n} [Ax] \quad \frac{}{np \vdash np} [Ax]}{np/n, n \vdash np} [L/]}{np, (np \setminus s)/np, np/n, n \vdash s} [L/]}$$

Figure 3.1: Example L derivation

$$\begin{array}{ll}
A \bullet B \not\vdash B \bullet A & A \otimes B \vdash B \otimes A \\
A/B \not\vdash B \setminus A & B \multimap A \vdash B \multimap A \\
(A \bullet B) \setminus C \not\vdash (B \setminus C)/A & (A \otimes B) \multimap C \vdash A \multimap (B \multimap C) \\
\quad \not\vdash A/A & \quad \vdash A \multimap A
\end{array}$$

3.2 Non-Associative Lambek Calculus

Where the Lambek calculus is only sensitive to the order of the antecedent, conventional linguistic wisdom teaches that the *structure* of the antecedent also plays a role. Whereas the structural rule of commutativity is usually applied implicitly in the sequent formulation of linear logic, the structural rule of associativity is implicit in the sequent formulation of L. Lambek (1961) also introduced the non-associative Lambek calculus NL, where the antecedents are *trees* of formulas.

In the sequent formulation of NL, which is shown in Table 3.3 on the following page, $\Gamma[\Delta]$ indicates an antecedent tree Γ with a distinguished subtree occurrence Δ .

Again, removing structural rules restricts the derivability of sequents. Below, we present some typical sequents which are derivable in L but undervivable in NL.

$$\begin{array}{ll}
A/B \not\vdash_{NL} (A/C)/(B/C) & A/B \vdash_L (A/C)/(B/C) \\
(A/B) \bullet (B/C) \not\vdash_{NL} A/C & (A/B) \bullet (B/C) \vdash_L A/C \\
(A \setminus B)/C \not\vdash_{NL} A \setminus (B/C) & (A \setminus B)/C \vdash_L A \setminus (B/C) \\
A/(B \bullet C) \not\vdash_{NL} (A/C)/B & A/(B \bullet C) \vdash_L (A/C)/B
\end{array}$$

$$\begin{array}{c}
\frac{}{A \vdash A} [Ax] \quad \frac{\Delta \vdash A \quad \Gamma[A] \vdash C}{\Gamma[\Delta] \vdash C} [Cut] \\
\\
\frac{\Gamma[A \circ B] \vdash C}{\Gamma[A \bullet B] \vdash C} [L\bullet] \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma \circ \Delta \vdash A \bullet B} [R\bullet] \\
\\
\frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[A/B \circ \Delta] \vdash C} [L/] \quad \frac{\Gamma \circ B \vdash A}{\Gamma \vdash A/B} [R/] \\
\\
\frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[\Delta \circ B \setminus A] \vdash C} [L\setminus] \quad \frac{B \circ \Gamma \vdash A}{\Gamma \vdash B \setminus A} [R\setminus]
\end{array}$$

Table 3.3: The sequent calculus NL

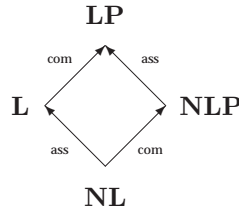


Figure 3.2: The different Lambek calculi and their relations

We can recover the associative Lambek calculus **L** from **NL** by adding the structural rules of associativity.

$$\frac{\Gamma[\Delta_1 \circ (\Delta_2 \circ \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ \Delta_2) \circ \Delta_3] \vdash C} [Ass1] \quad \frac{\Gamma[(\Delta_1 \circ \Delta_2) \circ \Delta_3] \vdash C}{\Gamma[\Delta_1 \circ (\Delta_2 \circ \Delta_3)] \vdash C} [Ass2]$$

We can also add the rule of commutativity to our system again, collapsing the two implications again and moving us to the Lambek calculus with Permutation **LP** which is an alternative formulation of **MILL** without empty antecedents.

$$\frac{\Gamma[\Delta_2 \circ \Delta_1] \vdash C}{\Gamma[\Delta_1 \circ \Delta_2] \vdash C} [Com]$$

It is also possible to add the structural rule of commutativity without adding associativity. The resulting logic is the non-associative Lambek calculus with permutation **NLP**, though the practical applications of this logic appear quite limited.

Summarizing, the logical systems discussed so far in this chapter can be related to each other as indicated by Figure 3.2.

3.3 Multimodal Lambek Calculus

A problem with the setup describe in the previous section is that while we can add structural rules when we need them, we can do so only at a global level. But this means that once we encounter a linguistic phenomenon which requires commutativity or associativity, adding the structural rules we need will collapse our logic to the next higher level.

For a solution to this, we could add controlled versions of these structural rules, as linear logic did with contraction and weakening, with structural modalities licensing the use of structural rules. We will discuss that solution in the next section. Here we will use a different solution, proposed by Moortgat & Oehrle (1993), who introduced a multimodal version of the non-associative Lambek calculus.

In the multimodal setting, we will have different *families* of connectives, representing different modes of composition. Given a mode i , it induces a family of connectives $\{A/_i B, A \bullet_i B, A \backslash_i B\}$ and a corresponding structural connective $\Gamma \circ_i \Delta$. More formally.

Definition 3.2 (Multimodal Formulas) *Over a finite set of atomic formulas \mathcal{A} and for each member i of a finite set of indices I , the set of formulas is defined as follows.*

$$\begin{aligned} \mathcal{F} ::= & \mathcal{A} \\ & | \mathcal{F}/_i \mathcal{F} \\ & | \mathcal{F} \bullet_i \mathcal{F} \\ & | \mathcal{F} \backslash_i \mathcal{F} \end{aligned}$$

Definition 3.3 (Antecedent Terms) *Over the set of formulas \mathcal{F} and all elements i of the set of indices I , we define the set of antecedent terms \mathcal{T} as follows*

$$\begin{aligned} \mathcal{T} ::= & \mathcal{F} \\ & | \mathcal{T} \circ_i \mathcal{T} \end{aligned}$$

In Table 3.4 we see the definition of $\mathbf{NL}_{\mathcal{R}}$, which is the sequent calculus of \mathbf{NL} with the addition of mode information and a separate set of structural rules \mathcal{R} . As before, the notation $\Gamma[\Delta]$ will mean the antecedent term Γ has a distinguished subterm occurrence Δ . In all logical rules the logical connective is coindexed with the structural punctuation. For the rules $[R/_i], [R \backslash_i]$ and $[L \bullet_i]$ this coindexing acts as a condition in forward chaining proof search, allowing the connective to be eliminated only if it appears in the right context.

The rules in the identity group are mode-independent.

Every structural rule is schematically of the following form shown in Table 3.4: inside an antecedent term Γ we replace a tree Ξ , which is an antecedent term with variables Δ_1 to Δ_n as its leaves, by a tree Ξ' where π is a permutation on the leaves.

Definition 3.4 (Linear Structural Rules) *A structural rule is linear if it is of the following form.*

$$\begin{array}{c}
\textbf{Identity} \\
\frac{}{A \vdash A} [\text{Ax}] \quad \frac{\Gamma[B] \vdash C \quad \Delta \vdash B}{\Gamma[\Delta] \vdash C} [\text{Cut}] \\
\textbf{Logical Rules} \\
\frac{\Gamma[A \circ_i B] \vdash C}{\Gamma[A \bullet_i B] \vdash C} [\text{L}\bullet_i] \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma \circ_i \Delta \vdash A \bullet_i B} [\text{R}\bullet_i] \\
\frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[A/_i B \circ_i \Delta] \vdash C} [\text{L}/_i] \quad \frac{\Gamma \circ_i B \vdash A}{\Gamma \vdash A/_i B} [\text{R}/_i] \\
\frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[\Delta \circ_i B \setminus_i A] \vdash C} [\text{L}\setminus_i] \quad \frac{B \circ_i \Gamma \vdash A}{\Gamma \vdash B \setminus_i A} [\text{R}\setminus_i] \\
\textbf{Structural Rules} \\
\frac{\Gamma[\Xi'[\Delta_1, \dots, \Delta_n]] \vdash C}{\Gamma[\Xi[\Delta_{\pi_1}, \dots, \Delta_{\pi_n}]] \vdash C} [\text{SR}]
\end{array}$$

Table 3.4: The sequent calculus $\text{NL}_{\mathcal{R}}$

$$\frac{\Gamma[\Xi'[\Delta_1, \dots, \Delta_n]] \vdash C}{\Gamma[\Xi[\Delta_{\pi_1}, \dots, \Delta_{\pi_n}]] \vdash C} [\text{SR}]$$

- (i) Ξ' is non-empty.
- (ii) π is a permutation on the leaves.

The main advantage of formulating the logic this way is that we can now provide modalized versions of the structural rules. For example, if we want to claim that mode a is associative, we can add the structural rules for this specific mode as follows.

$$\frac{\Gamma[\Delta_1 \circ_a (\Delta_2 \circ_a \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_a \Delta_2) \circ_a \Delta_3] \vdash C} [\text{Ass1}] \quad \frac{\Gamma[(\Delta_1 \circ_a \Delta_2) \circ_a \Delta_3] \vdash C}{\Gamma[\Delta_1 \circ_a (\Delta_2 \circ_a \Delta_3)] \vdash C} [\text{Ass2}]$$

Suppose we also have a mode n , representing a non-associative family of connectives. We can relate this mode to the associative mode a by means of a structural rule as follows.

$$\frac{\Gamma[\Delta_1 \circ_a \Delta_2] \vdash C}{\Gamma[\Delta_1 \circ_n \Delta_2] \vdash C} [\text{Link}]$$

Now, using this postulate we can derive $a/_a b \vdash a/_n b$ as follows.

$$\frac{\frac{\frac{\overline{a \vdash a} \quad [Ax]}{a/a \circ_a b \vdash a} \quad \frac{\overline{b \vdash b} \quad [Ax]}{b \vdash a} \quad [L/a]}{a/a \circ_a b \vdash a} \quad [Link]}{a/a \vdash a/n b} \quad [R/n]$$

The converse will be excluded, because we can only apply the $[L/]$ rule when the mode on the logical and the structural connective are identical.

$$\frac{\text{FAIL}}{\frac{a/n b \circ_a b \vdash a}{a/n b \vdash a/a b} \quad [R/a]}$$

In other words, given a formula $a/a b$ which has the right to reassociate, we can forfeit that right and turn it into a non-associative formula $a/n b$, but the converse does not hold. This is in the spirit of the $[L!]$ rule from linear logic, where a formula which does allow contraction and weakening can be used as a weaker formula which doesn't use these rules.

Interestingly, the opposite linking relation is also possible, where $a/n b \vdash a/a b$ is a theorem and the converse is not. Hepple (1994a) and de Groote (1996) subscribe to this point of view, where the intuition is that if you can derive a formula $a/n b$, that is without using associativity, then you can certainly derive $a/a b$ where the use of associativity is allowed.

We can also formulate more complex interactions between the different modes, for example modes which have special interaction properties when they occur together.

Example 3.5 *An example of these 'mixed' structural rules are the following versions of associativity and commutativity, proposed by Moortgat & Oehrle (1994) to deal with phenomena like Dutch verb raising, which are outside the scope of unimodal L.*

$$\frac{\Gamma[(\Delta_1 \circ_0 \Delta_3) \circ_1 \Delta_2] \vdash C}{\Gamma[(\Delta_1 \circ_1 \Delta_2) \circ_0 \Delta_3] \vdash C} \quad [MC]$$

$$\frac{\Gamma[\Delta_1 \circ_1 (\Delta_2 \circ_0 \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_1 \Delta_2) \circ_0 \Delta_3] \vdash C} \quad [MA]$$

We can use these postulates, in combination with the Dutch lexicon of Table 3.5 on the next page, to give an account of verb raising in Dutch subordinate clauses.

Figure 3.3 shows a derivation of '(dat) Ripley Newt wil redder'.

Observe, however, that we can now also derive '* (dat) Ripley wil Newt redder' by omitting the last rule of the proof, resulting in a sentence which is an ungrammatical subordinate clause. In the next section we will introduce tools to restrict the generation of these kinds of ungrammatical sentences.

$$\Box^\perp A, \Box^\perp(A \setminus B) \not\vdash \Box^\perp B \quad \mathbf{K}$$

Another difference with the modal operators for linear logic is that the logical rules for the unary connectives don't refer explicitly to modalized formulas, as in the $[R!]$ rule, but that the unary connectives have their own *structural* counterpart, $\langle \cdot \rangle^i$.

More formally, when we allow different modes of composition for the unary as well as the binary modes, the system $\mathbf{NL}\diamond_{\mathcal{R}}$ looks as follows.

Definition 3.6 (Multimodal Formulas) *Over a set of atomic formulas \mathcal{A} , a set I of binary indices and a set J of unary indices, we have the following set of formulas*

$$\begin{aligned} \mathcal{F} ::= & \mathcal{A} \\ & | \diamond_j \mathcal{F} \\ & | \Box_j^\perp \mathcal{F} \\ & | \mathcal{F} /_i \mathcal{F} \\ & | \mathcal{F} \bullet_i \mathcal{F} \\ & | \mathcal{F} \setminus_i \mathcal{F} \end{aligned}$$

Definition 3.7 (Antecedent Terms) *Over the set of formulas \mathcal{F} , we define the set of antecedent terms \mathcal{T} as follows*

$$\begin{aligned} \mathcal{T} ::= & \mathcal{F} \\ & | \langle \mathcal{T} \rangle^i \\ & | \mathcal{T} \circ_i \mathcal{T} \end{aligned}$$

The unary connectives greatly extend the possible structural postulates. We can have postulates which determine the properties of the unary operators, as in the case of modal logic.

$$\frac{\Gamma[\langle \Delta \rangle^1] \vdash C}{\Gamma[\Delta] \vdash C} [T] \quad \frac{\Gamma[\langle \Delta \rangle^1] \vdash C}{\Gamma[\langle \langle \Delta \rangle^1 \rangle^1] \vdash C} [4]$$

We can also have interaction between the unary and binary modes of composition with distribution principles, as in the modal \mathbf{K} postulate.

$$\frac{\Gamma[\langle \Delta_1 \rangle^1 \circ_0 \langle \Delta_2 \rangle^1] \vdash C}{\Gamma[\langle \Delta_1 \circ_0 \Delta_2 \rangle^1] \vdash C} [K]$$

Example 3.8 *Adding the $[K]$ structural rule makes the \mathbf{K} postulate a theorem as follows.*

$$\begin{aligned} & \frac{\frac{\overline{A \vdash A} [Ax] \quad \overline{B \vdash B} [Ax]}{A \circ_0 A \setminus_0 B} [L \setminus_0]}{A \circ_0 \langle \Box_1^\perp(A \setminus_0 B) \rangle^1 \vdash B} [L \Box_1^\perp] \\ & \frac{\langle \Box_1^\perp A \rangle^1 \circ_0 \langle \Box_1^\perp(A \setminus_0 B) \rangle^1 \vdash B}{\langle \Box_1^\perp A \circ_0 \Box_1^\perp(A \setminus_0 B) \rangle^1 \vdash B} [K] \\ & \frac{\langle \Box_1^\perp A \circ_0 \Box_1^\perp(A \setminus_0 B) \rangle^1 \vdash B}{\Box_1^\perp A \circ_0 \Box_1^\perp(A \setminus_0 B) \vdash \Box_1^\perp B} [R \Box_1^\perp] \end{aligned}$$

Similarly, the $[T]$ and the $[4]$ structural rule make the \mathbf{T} and $\mathbf{4}$ postulate derivable.

Identity

$$\frac{}{A \vdash A} \text{ [Ax]} \quad \frac{\Gamma[B] \vdash C \quad \Delta \vdash B}{\Gamma[\Delta] \vdash C} \text{ [Cut]}$$

Unary Connectives

$$\frac{\Gamma[\langle A \rangle^i] \vdash C}{\Gamma[\diamond_i A] \vdash C} \text{ [L}\diamond_i] \quad \frac{\Gamma \vdash C}{\langle \Gamma \rangle^i \vdash \diamond_i C} \text{ [R}\diamond_i]$$

$$\frac{\Gamma[A] \vdash C}{\Gamma[\langle \square_i A \rangle^i] \vdash C} \text{ [L}\square_i] \quad \frac{\langle \Gamma \rangle^i \vdash C}{\Gamma \vdash \square_i C} \text{ [R}\square_i]$$

Binary Connectives

$$\frac{\Gamma[A \circ_i B] \vdash C}{\Gamma[A \bullet_i B] \vdash C} \text{ [L}\bullet_i] \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma \circ_i \Delta \vdash A \bullet_i B} \text{ [R}\bullet_i]$$

$$\frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[A/_i B \circ_i \Delta] \vdash C} \text{ [L}/_i] \quad \frac{\Gamma \circ_i B \vdash A}{\Gamma \vdash A/_i B} \text{ [R}/_i]$$

$$\frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[\Delta \circ_i B \setminus_i A] \vdash C} \text{ [L}\setminus_i] \quad \frac{B \circ_i \Gamma \vdash A}{\Gamma \vdash B \setminus_i A} \text{ [R}\setminus_i]$$

Structural Rules

$$\frac{\Gamma[\Xi'[\Delta_1, \dots, \Delta_n]] \vdash C}{\Gamma[\Xi[\Delta_{\pi_1}, \dots, \Delta_{\pi_n}]] \vdash C} \text{ [SR]}$$

Table 3.6: The sequent calculus $\mathbf{NL}\diamond_{\mathcal{R}}$

It is also possible to have weaker versions of the **K** postulate where the unary mode distributes only to the left or the right branch of the binary mode.

$$\frac{\Gamma[\langle \Delta_1 \rangle^i \circ_j \Delta_2] \vdash C}{\Gamma[\langle \Delta_1 \circ_j \Delta_2 \rangle^i] \vdash C} \text{ [K1]} \quad \frac{\Gamma[\Delta_1 \circ_j \langle \Delta_2 \rangle^i] \vdash C}{\Gamma[\langle \Delta_1 \circ_j \Delta_2 \rangle^i] \vdash C} \text{ [K2]}$$

Finally, we can have ‘modally licensed’ versions of the usual structural rules in the same way linear logic had for the exponentials.

$$\frac{\Gamma[\Delta_2 \circ_a \langle \Delta_1 \rangle^c] \vdash C}{\Gamma[\langle \Delta_1 \rangle^c \circ_a \Delta_2] \vdash C} \text{ [Com]} \quad \frac{\Gamma[\langle \Delta_1 \rangle^c \circ_a \Delta_2] \vdash C}{\Gamma[\Delta_2 \circ_a \langle \Delta_1 \rangle^c] \vdash C} \text{ [Com]}$$

The unary connectives will also be the main way to encode linguistic features in the multimodal Lambek calculus, but see Heylen (1999) for an overview and comparison of several different possibilities.

$$\begin{array}{c}
\frac{\Gamma[\langle \Delta \rangle^0] \vdash C}{\Gamma[\langle \Delta \rangle^1] \vdash C} [I] \\
\frac{\Gamma[\langle \Delta_1 \rangle^1 \circ_1 \Delta_2] \vdash C}{\Gamma[\langle \Delta_1 \circ_1 \Delta_2 \rangle^1] \vdash C} [K1] \\
\frac{\Gamma[(\Delta_1 \circ_0 \Delta_3) \circ_1 \Delta_2] \vdash C}{\Gamma[(\Delta_1 \circ_1 \Delta_2) \circ_0 \Delta_3] \vdash C} [MC]
\end{array}
\qquad
\begin{array}{c}
\frac{\Gamma[\langle \Delta_1 \rangle^0 \circ_0 \langle \Delta_2 \rangle^0] \vdash C}{\Gamma[\langle \Delta_1 \circ_1 \Delta_2 \rangle^0] \vdash C} [K] \\
\frac{\Gamma[\Delta_1 \circ_1 \langle \Delta_2 \rangle^1] \vdash C}{\Gamma[\langle \Delta_1 \circ_1 \Delta_2 \rangle^1] \vdash C} [K2] \\
\frac{\Gamma[\Delta_1 \circ_1 (\Delta_2 \circ_0 \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_1 \Delta_2) \circ_0 \Delta_3] \vdash C} [MA]
\end{array}$$

Table 3.7: Structural rules for Dutch verb raising

$$\begin{array}{l}
l(\text{ripley}) = np \\
l(\text{newt}) = np \\
l(\text{wil}) = \square_0^1((np \setminus_1 s) /_0 inf) \\
l(\text{redden}) = \square_0^1(np \setminus_1 inf)
\end{array}$$

Table 3.8: Revised verb raising lexicon

Example 3.9 In Example 3.5, we noted that the lexicon given there allowed us to derive ungrammatical sentences in addition to grammatical ones. Moortgat & Oehrle (1994) present the following solution to this: in addition to the two binary modes of Example 3.5 we have two unary modes 0, representing lexical heads, and 1, representing phrasal heads. The structural rules are given in Table 3.7. Structural rule [I] is an inclusion postulate indicating every lexical head is also a phrasal head. The crucial structural rule is [K]: it states that two lexical heads combined in the binary head adjunction mode 0 can be combined into one complex lexical head. The [K1] and [K2] structural rules allow a phrasal head to move up through a phrasal binary composition mode, and the [MC] and [MA] structural rules are unchanged from before.

The lexicon for this fragment is shown in Table 3.8. Compared to Table 3.5 on page 38 the only difference is that the verbs have been marked as lexical heads by virtue of the ‘ \square^1 ’ as their main formula constructor.

Figure 3.4 shows a derivation of ‘(dat) Ripley Newt wil redden’.

Kurtonina & Moortgat (1997) show how we can use the unary modalities to give embedding translations between the different unimodal logics discussed in Section 3.2, which is reminiscent of the way the structural rules of contraction and weakening are modally licensed in linear logic. What is interesting, however, is that Kurtonina & Moortgat also give embedding translations which *restrict* the structural rules of a system. For example, by enriching \mathbf{L} with the unary modalities, we can give a faithful translation of \mathbf{NL} formulas into the resulting logic $\mathbf{L}\diamond$.

In Chapter 9, when we have more tools available to analyze $\mathbf{NL}\diamond_{\mathcal{R}}$, we will look at the complexity of this logic, and carve out a fragment which is

| | |
|---|---|
| Hypothesis | |
| $A \vdash A$ | |
| Binary Connectives | |
| $[A \vdash A]^n \quad [B \vdash B]^n$ | |
| \vdots | |
| $\frac{\Delta \vdash A \bullet_i B \quad \Gamma[A \circ_i B] \vdash C}{\Gamma[\Delta] \vdash C} [\bullet E]^n$ | $\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma \circ_i \Delta \vdash A \bullet_i B} [\bullet I]^n$ |
| $\frac{\Gamma \vdash A /_i B \quad \Delta \vdash B}{\Gamma \circ_i \Delta \vdash A} [/E]$ | $\frac{[B \vdash B]^n \quad \vdots \quad \Gamma \circ_i B \vdash A}{\Gamma \vdash A /_i B} [/I]^n$ |
| $\frac{\Delta \vdash B \quad \Gamma \vdash B \setminus_i A}{\Delta \circ_i \Gamma \vdash A} [\setminus E]$ | $\frac{[B \vdash B]^n \quad \vdots \quad B \circ_i \Gamma \vdash A}{\Gamma \vdash B \setminus_i A} [\setminus I]^n$ |
| Unary Connectives | |
| $[A \vdash A]^n$ | |
| \vdots | |
| $\frac{\Delta \vdash \diamond_i A \quad \Gamma[\langle A \rangle^i] \vdash C}{\Gamma[\Delta] \vdash C} [\diamond E]^n$ | $\frac{\Gamma \vdash A}{\langle \Gamma \rangle^i \vdash \diamond_i A} [\diamond I]$ |
| $\frac{\Gamma \vdash \square_i^\perp A}{\langle \Gamma \rangle^i \vdash X} [\square^\perp E]$ | $\frac{\langle \Gamma \rangle^i \vdash A}{\Gamma \vdash \square_i^\perp A} [\square^\perp I]$ |
| Structural Rules | |
| $\frac{\Gamma[\Xi'[\Delta_1, \dots, \Delta_n]] \vdash C}{\Gamma[\Xi[\Delta_{\pi_1}, \dots, \Delta_{\pi_n}]] \vdash C} [SR]$ | |

Table 3.9: The natural deduction calculus $\mathbf{NL}\diamond_{\mathcal{R}}$

isomorphism to a Curry-Howard *homomorphism*, that is, instead of having a one-one correspondence between lambda terms and proofs we have one-many correspondence. Meaning we allow for a (possibly empty) set of natural deduction proofs to correspond to a lambda term.

For natural language semantics, option (ii) appears to be the most useful. The semantically labeled natural deduction calculus we get in this framework is shown in Table 3.10 on the following page. The unary connectives have their own term constructors, which are inspired by the binary ones,

| | |
|--|---|
| Hypothesis | |
| $x : A \vdash x : A$ | |
| Binary Connectives | |
| $[x : A \vdash x : A]^n \quad [y : B \vdash y : B]^n$ | |
| \vdots | |
| $\frac{\Delta \vdash u : A \bullet_i B \quad \Gamma[x : A \circ_i y : B] \vdash t : C}{\Gamma[\Delta] \vdash t[x := \pi^1 u, y := \pi^2 u] : C} [\bullet E]^n$ | $\frac{\Gamma \vdash t : A \quad \Delta \vdash u : B}{\Gamma \circ_i \Delta \vdash \langle t, u \rangle : A \bullet_i B} [\bullet I]$ |
| $\frac{\Gamma \vdash t : A /_i B \quad \Delta \vdash u : B}{\Gamma \circ_i \Delta \vdash (tu) : A} [/E]$ | $\frac{\Gamma \circ_i x : B \vdash t : A}{\Gamma \vdash \lambda x. t : A /_i B} [/I]^n$ |
| $\frac{\Delta \vdash u : B \quad \Gamma \vdash t : B \setminus_i A}{\Delta \circ_i \Gamma \vdash (tu) : A} [\setminus E]$ | $\frac{x : B \vdash x : B \quad \Gamma \vdash t : A}{\Gamma \vdash \lambda x. t : B \setminus_i A} [\setminus I]^n$ |
| Unary Connectives | |
| $[y : A \vdash y : A]^n$ | |
| \vdots | |
| $\frac{\Delta \vdash u : \diamond_i A \quad \Gamma[\langle y : A \rangle^i] \vdash t : C}{\Gamma[\Delta] \vdash t[y := \sphericalangle u] : C} [\diamond E]^n$ | $\frac{\Gamma \vdash x : A}{\Gamma \wedge x : \diamond_i A} [\diamond I]$ |
| $\frac{\Gamma \vdash t : \square_i^\perp A}{\langle \Gamma \rangle^i \vdash \sqcup t : A} [\square^\perp E]$ | $\frac{\langle \Gamma \rangle^i \vdash t : A}{\Gamma \sqsupset t : \square_i^\perp A} [\square^\perp I]$ |

Table 3.10: The natural deduction calculus $\mathbf{NL}_{\diamond \mathcal{R}}$ with semantic labeling

even though it is unclear if we can find useful semantic applications of these constructors.

Example 3.10 Now, if we return to Example 2.9 but add directionality to the implications, we see that our modified lexicon looks as follows.

$$\begin{aligned}
 l(\mathit{shot}) &= (np \setminus_a s) /_a np - \lambda v. \lambda w. \mathit{shot}(v, w) \\
 l(\mathit{someone}) &= s /_a (np \setminus_a s) - \lambda x. \exists y. (x y) \\
 l(\mathit{vito}) &= np - \mathit{vito}
 \end{aligned}$$

Suppressing the antecedent information for readability, we can now derive ‘some-one shot Vito’ as a sentence in the following way.

$$\begin{aligned}
v(\diamond_i A) &= \{x \mid \exists y.(R_i^2 xy \wedge y \in v(A))\} \\
v(\square_i^\downarrow A) &= \{y \mid \forall x.(R_i^2 xy \rightarrow x \in v(A))\} \\
v(A \bullet_i B) &= \{x \mid \exists y.z.(R_i^3 xyz \wedge y \in v(A) \wedge z \in v(B))\} \\
v(A/_i B) &= \{y \mid \forall x.z.((R_i^3 xyz \wedge z \in v(B)) \rightarrow x \in v(A))\} \\
v(B \backslash_i A) &= \{z \mid \forall x.y.((R_i^3 xyz \wedge y \in v(B)) \rightarrow x \in v(A))\}
\end{aligned}$$

Table 3.11: Model theoretic evaluation of complex formulas

$$\frac{\frac{y : (np \backslash_a s) /_a np \quad z : np}{(y z) : np \backslash_a s} [/_a E] \quad [x : np]^1}{((y z) x) : s} [\backslash_a E] \quad \frac{v : s /_a (np \backslash_a s) \quad \lambda x.((y z) x) : np \backslash_a s}{(v \lambda x.((y z) x)) : s} [\backslash_a I]^1 [/_a E]$$

3.6 Model Theory

Though this thesis is primarily concerned with proof theory, we will turn for a moment to the model theoretic side of the multimodal Lambek calculus. Restricting ourselves to a subset of the possible structural postulates will also give us soundness and completeness with respect to the model.

Formulas will be interpreted in multimodal Kripke frames: tuples of the form $\langle W, \{R_j^2\}_{j \in J}, \{R_i^3\}_{i \in I} \rangle$. The set of worlds W are our linguistic resources and the accessibility relations R_j^2 and R_i^3 model the unary and binary composition of resources. The valuation v is defined as follows. It assigns arbitrary subsets of W to the atomic formulas, and evaluates complex formulas as shown in Table 3.11.

Readers familiar with the Kripke semantics for modal logic will recognise the \diamond interpretation as that of the modal possibility operator, and the \square^\downarrow interpretation as that of the modal necessity operator with the direction reversed (as suggested by the downarrow superscript). It is important to see that the binary connectives are also modal operators. They can be seen as generalizations of the unary cases.

The valuation above gives us a base logic without structural rules for any of the modes. Structural postulates are translated into *restrictions* on the accessibility relations R . A structural postulate $[T]$ for a unary mode j , for example, will restrict the possible accessibility relations R_j^2 to those which satisfy $\forall x(R_j^2 xx)$, i.e. are reflexive.

Definition 3.11 A weak Sahlqvist structural rule (Kurtonina 1995) is a rule of the form

$$\frac{\Gamma[\Xi'[\Theta_1, \dots, \Theta_m]] \vdash C}{\Gamma[\Xi[\Delta_1, \dots, \Delta_n]] \vdash C}$$

subject to the following conditions

- (i) both Ξ and Ξ' contain only the structural connectives \circ_i and $\langle \cdot \rangle^i$.
- (ii) Ξ' contains at least one structural connective.
- (iii) there is no repetition of variables in $\Delta_1, \dots, \Delta_n$.
- (iv) all variables in $\Theta_1, \dots, \Theta_m$ occur in $\Delta_1, \dots, \Delta_n$.

All of the structural rules we have seen so far are Sahlqvist structural rules. The linearity condition on structural rules, where we demand that $\Theta_1, \dots, \Theta_m$ is a permutation of $\Delta_1, \dots, \Delta_n$, restricts the structural rules even further.

Example 3.12 *Some valid weak Sahlqvist structural rules which are not linear structural rules are the following.*

$$\frac{\Gamma[\langle \Delta_1 \rangle^0] \vdash C}{\Gamma[\Delta_1 \circ_0 \Delta_2] \vdash C} [LW] \quad \frac{\Gamma[\Delta \circ_0 \Delta] \vdash C}{\Gamma[\Delta] \vdash C} [LC]$$

However, the following are disallowed, violating conditions (i), (ii), (iii) and (iv) respectively.

$$\frac{\Gamma[B \setminus_0 A] \vdash C}{\Gamma[A /_0 B] \vdash C} [C] \quad \frac{\Gamma[\Delta] \vdash C}{\Gamma[\langle \Delta \rangle^0] \vdash C} [T^{-1}]$$

$$\frac{\Gamma[\langle \Delta \rangle^0] \vdash C}{\Gamma[\Delta \circ_0 \Delta] \vdash C} [P1] \quad \frac{\Gamma[\Delta_1 \circ_0 \Delta_2] \vdash C}{\Gamma[\Delta_1] \vdash C} [P2]$$

Theorem 3.13 (Kurtonina (1995)) *The multimodal sequent calculus is sound and complete for $NL\Diamond$ and an arbitrary set \mathcal{R} of weak Sahlqvist structural rules.*

Corollary 3.14 *The multimodal sequent calculus is sound and complete for $NL\Diamond$ and an arbitrary set \mathcal{R} of linear structural rules.*

3.7 Conclusions

In this chapter we have seen how removing associativity and commutativity as global structural rules makes the resulting logic very suitable for linguistic analysis. We have seen how a multimodal system containing both unary and binary connectives gives us enough flexibility to deal with the often subtle restrictions on word order encountered in natural language.

Our aim over the next chapters will be to incorporate some of the proof theoretic advances of linear logic, notably proof nets, into the multimodal Lambek calculus.

PART II

PROOF NETS AND LINGUISTICS

CHAPTER 4

PROOF NETS FOR MULTIPLICATIVE LINEAR LOGIC

FOR the multiplicative fragment of linear logic, we have a particularly elegant proof theory, called *proof nets*. Proof nets were introduced by Girard (1987), both for the multiplicative fragment and for full linear logic, though Girard already noted that the proof net calculus was considerably less elegant for the full system. Improvements to the original formulation of proof nets for the multiplicative fragment are due to Danos & Regnier (1989) and Danos (1990).

Let's take a closer look at the one sided sequent calculus for multiplicative linear logic, repeated here for convenience as Table 4.1. Throughout this chapter, with the exception of Section 4.7 where we discuss non-commutative proof nets, we will keep the commutativity rule implicit.

A problem with this calculus is that a sequent like

$$\vdash a, a^\perp \otimes b, b^\perp \otimes c, c^\perp$$

has two different sequent proofs, depending on whether we apply the \otimes rule first to $a^\perp \otimes b$ or to $b^\perp \otimes c$.

$$\frac{\frac{\frac{}{\vdash a, a^\perp} [Ax]}{\vdash a, a^\perp \otimes b, b^\perp \otimes c, c^\perp} [\otimes] \quad \frac{\frac{\frac{}{\vdash b, b^\perp} [Ax]}{\vdash b, b^\perp \otimes c, c^\perp} [\otimes] \quad \frac{}{\vdash c, c^\perp} [Ax]}{\vdash b, b^\perp \otimes c, c^\perp} [\otimes]}{\vdash a, a^\perp \otimes b, b^\perp \otimes c, c^\perp} [\otimes]}$$

$$\frac{\frac{\frac{}{\vdash a, a^\perp} [Ax]}{\vdash a, a^\perp \otimes b, b^\perp} [\otimes] \quad \frac{}{\vdash c, c^\perp} [Ax]}{\vdash a, a^\perp \otimes b, b^\perp \otimes c, c^\perp} [\otimes] \quad \frac{}{\vdash b, b^\perp} [Ax]}{\vdash a, a^\perp \otimes b, b^\perp \otimes c, c^\perp} [\otimes]}$$

$$\begin{array}{c}
\frac{}{\vdash A, A^\perp} \text{[Ax]} \quad \frac{\vdash \Gamma, A \quad \vdash A^\perp, \Delta}{\vdash \Gamma, \Delta} \text{[Cut]} \\
\frac{\vdash \Gamma, A \quad \vdash B, \Delta}{\vdash \Gamma, A \otimes B, \Delta} \text{[\otimes]} \quad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \wp B} \text{[\wp]}
\end{array}$$

Table 4.1: The sequent calculus **MLL**

We would like to claim that the ‘essence’ of these proofs is the same, that is, all logical rules in both proofs are applied to the same formula occurrences, only the context formulas Γ and Δ of the rules are managed differently. Computationally, this kind of derivational ambiguity, sometimes called spurious ambiguity, is also quite harmful, because when we search for proofs using the sequent calculus, we may find equivalent proofs many, many times.

In a proof net, the different logical rules are applied in parallel, and the result is a system which is much like a natural deduction system with many conclusions.

4.1 Proof Nets

The inductive construction rules for proof nets mimic the sequent rules quite closely, but they abstract away from the context formulas.

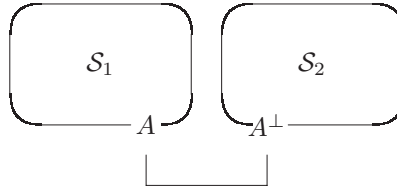
Definition 4.1 (Proof Net) *The set of proof nets is inductively defined as follows*

[Axiom] *If A is a formula of MLL, then the following is a proof net with conclusions A, A^\perp .*



The axiom link is symmetric, i.e. the order of the conclusions of the rule is irrelevant.

[Cut] *If S_1 is a proof net with conclusion A and S_2 is a proof net with conclusion A^\perp , then we can combine them with a cut link as follows.*



Like the axiom link, the cut link is symmetric.

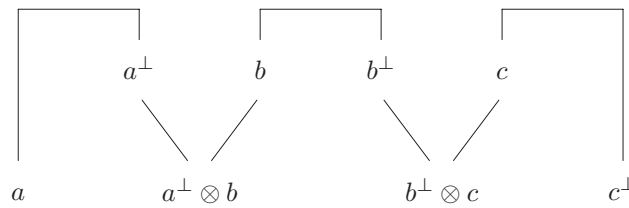
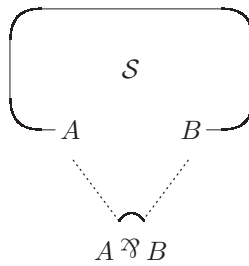


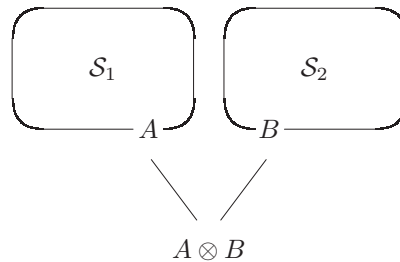
Figure 4.1: Proof net of $\vdash a, a^\perp \otimes b, b^\perp \otimes c, c^\perp$

[Par] If S is a proof net with conclusions A and B , then we can attach a par link to it as follows.



The order of the formulas A and B is relevant, as it determines whether the conclusion of the new proof net is $A \wp B$ or $B \wp A$.

[Tensor] If S_1 is a proof net with conclusion A and S_2 is a proof net with conclusion B , then we can combine them with a tensor link as follows.



Like the par link, the order of A and B with respect to the tensor link is relevant.

Example 4.2 When we follow the inductive definition to construct the proof net corresponding to the sequent proof on page 49 we see that the proof net will look as shown in Figure 4.1 regardless of the sequence in which we apply the two tensor rules.



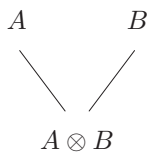
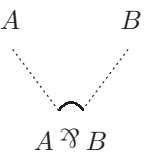
| | |
|--|---|
| Axiom | |
|  | Premises: None Conclusions: A, A^\perp |
| Cut | |
|  | Premises: A, A^\perp Conclusions: None |
| Tensor | |
|  | Premises: A, B Conclusion: $A \otimes B$ |
| Par | |
|  | Premises: A, B Conclusion: $A \wp B$ |

Table 4.2: Links for MLL

The question we are interested in is the following: given a list of formulas, is there a proof net with these formulas as a conclusion? We answer this question in the following way. First, in Section 4.2, we will give a definition of *proof structures*, candidate proof nets which can be enumerated for any given list of formulas. In Section 4.3, we will give a *correctness criterion* which will identify the proof nets among the proof structures. Finally, in Section 4.4, we show that the proof net calculus satisfies cut elimination, that is, to show that we can restrict ourselves, without loss of generality, to proof nets which do not contain cut links.

4.2 Proof Structures

Definition 4.3 (Proof Structure) A proof structure $\langle S, \mathcal{L} \rangle$ consists of a set S of formulas and a set \mathcal{L} of links in S , where the links are as shown in Table 4.2,

Furthermore, a proof structure must satisfy the following conditions.

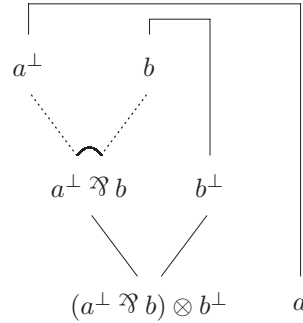


Figure 4.2: Incorrect proof structure

- every formula is at most once the premiss of a link,
- every formula is exactly once the conclusion of a link.

Formulas which are not the premiss of any link are called the conclusions of a proof structure.

Now, given a sequent, we can enumerate all possible proof structures for it by unfolding all connectives until we reach the atomic formulas and their negations and connect these using axiom links.

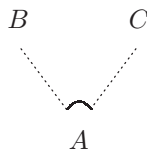
4.3 Soundness and Completeness

Obviously, not all proof structures are proof nets. For example, given that the sequent $\vdash (a^\perp \wp b) \otimes b^\perp, a$ is undervivable, the proof structure shown in Figure 4.2 is not a proof net.

We need a criterion which allows us to identify the proof nets from other proof structures. Girard (1987) gave a condition based on travel instructions through a proof net. The criterion presented below is an improvement due to Danos & Regnier (1989). We will discuss another criterion in Section 4.5.

Definition 4.4 For a proof structure S , a switching ω for S is a choice for every par link of one of its premisses.

Definition 4.5 From a proof structure S and a switching ω we obtain a correction graph ωS by replacing all par links



by one of the following links, depending on the premiss of the link selected by ω .

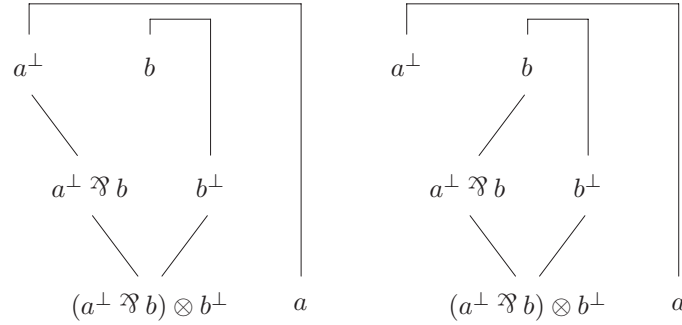
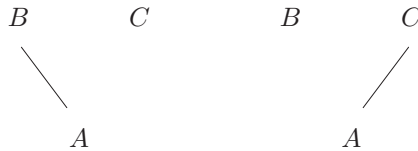


Figure 4.3: Correction graphs for the incorrect proof structure of Figure 4.2 on the preceding page



For a tensor link with premisses A and B and with conclusion C the correction graph has an (undirected) edge both from A to C and from B to C , whereas a par link, being an disjunction, produces an edge *either* from A to C or from B to C .

Example 4.6 Before sketching the proof of the main theorem, we show that the proof structure above is not a proof net. It has the following two correction graphs.

We can see the correction graph on the right is both cyclic and disconnected and conclude this proof structure is incorrect.

Lemma 4.7 A proof net without terminal par links has a splitting tensor link, that is, removing the tensor link and its conclusion will yield two disjoint proof nets.

Lemma 4.7 was first proved by Girard (1987). For a proof more in line with the acyclicity and connectedness criterion we are using, we refer the reader to Danos & Regnier (1989) or Bellin & van de Wiele (1995).

Lemma 4.8 A proof net which has at least one par link has a splitting par link. Removing the par link will yield two disjoint proof nets, where one of the proof nets will have the conclusion C of the par link as a hypothesis, that is, a formula which is not the conclusion of any link.

We refer the reader to Danos (1990) for a proof of this lemma.

The main theorem shows that the proof nets, i.e. the proof structures which are sequentializable, are exactly those proof structures of which all correction graphs are acyclic and connected.

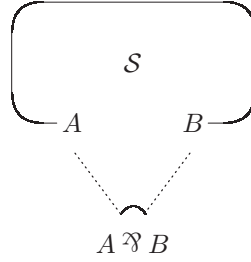


Figure 4.4: Terminal par link

Theorem 4.9 *A proof structure is a proof net iff all its correction graphs are acyclic and connected.*

Proof

[\Rightarrow] Induction.

[\Leftarrow] We have to prove that every proof structure of which all correction graphs are acyclic and connected corresponds to a sequent proof. To reduce the number of cases in the proof, we will treat a cut link as a tensor link with a special formula (Cut) as its conclusion.

We use induction on the number of logical links in the proof net. In the case there are no logical links in the proof net, then by disconnectedness it must consist of a single axiom link and the sequent proof corresponding to this proof net consists of just the axiom rule.

If the proof net does contain logical links, there are two basic ways to continue the sequentialization part of the proof. The splitting tensor sequentialization proof uses Lemma 4.7 and is perhaps the most familiar way of proving sequentialization. The splitting par sequentialization proof uses Lemma 4.8.

Splitting Tensor We proceed by a case analysis; if the proof net has a terminal par link, it must be of the form shown in Figure 4.4.

When we remove the par link and its conclusion, the resulting proof structure is still a proof net, so we can apply the induction hypothesis to give us a sequent proof \mathcal{D} of $\vdash \Gamma, A, B$, which we can extend as follows.

$$\frac{\vdots \mathcal{D}}{\vdash \Gamma, A, B} \quad [\text{⋈}]$$

When all terminal links are tensor links, then by Lemma 4.7 one of these tensor links is a splitting tensor. That is, a tensor link such that removing it and its conclusion yields two disjoint proof nets.

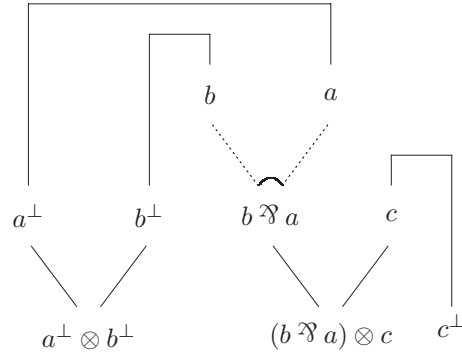


Figure 4.5: Proof net with a non-splitting tensor link

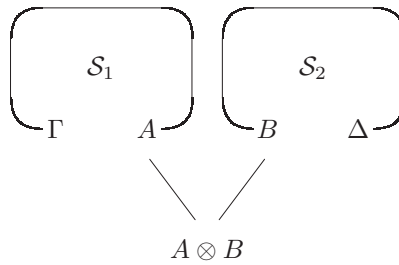


Figure 4.6: Splitting tensor link

Not every tensor link is a splitting tensor, because removing an arbitrary tensor link can result in a single, disconnected proof structure. For example, in the case shown in Figure 4.5 removing the leftmost tensor link will generate an incorrect proof structure, because the path from a to b through $a^\perp \otimes b^\perp$ will disappear, making both correction graphs disconnected. However, Lemma 4.7 guarantees the existence of a splitting tensor and in the case above, the other tensor link is indeed splitting.

Given that there is a splitting tensor link, we are schematically in the situation shown in Figure 4.6.

Induction hypothesis gives us a proof \mathcal{D}_1 of $\vdash \Gamma, A$ and a proof \mathcal{D}_2 of $\vdash B, \Delta$. We can combine these proofs as follows to produce a proof of $\vdash \Gamma, A \otimes B, \Delta$.

$$\frac{\begin{array}{c} \vdots \mathcal{D}_1 \\ \vdash \Gamma, A \end{array} \quad \begin{array}{c} \vdots \mathcal{D}_2 \\ \vdash B, \Delta \end{array}}{\vdash \Gamma, A \otimes B, \Delta} [\otimes] \quad \square$$

Splitting Par If there are par links in the proof net, Lemma 4.8 guarantees that way are schematically in the case shown in Figure 4.7 on the left,

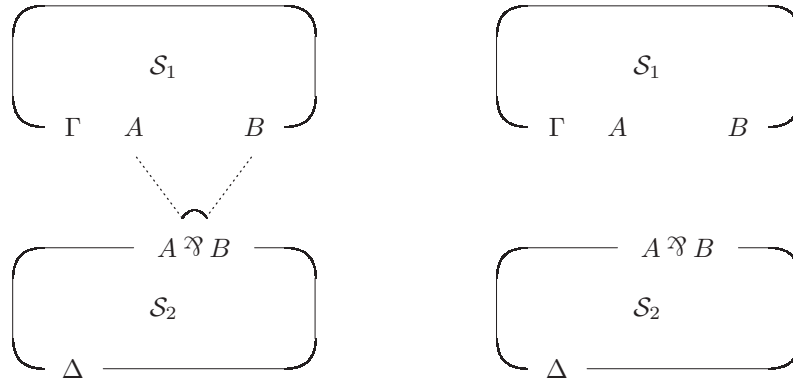


Figure 4.7: Splitting par link

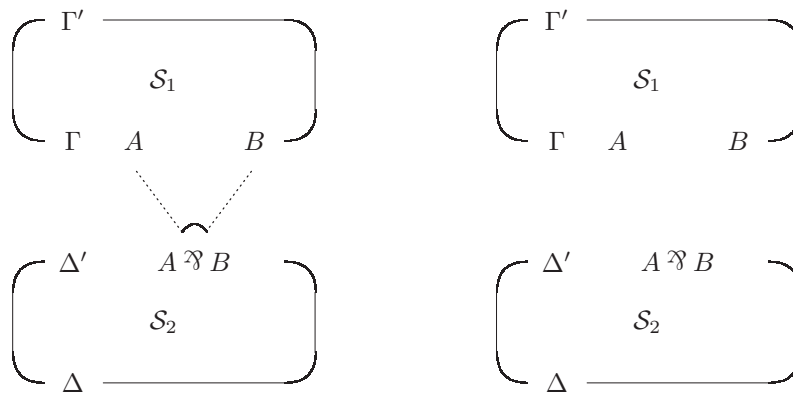


Figure 4.8: Splitting par link with hypotheses

and removing the par link results in the structure shown in Figure 4.7 on the right.

A problem with the structure on the right is that it is not even a proof structure! This is because the formula $A \wp B$ is not the conclusion of any link, violating condition 2 of Definition 4.3. However, we can modify the definitions of proof structures and proof nets to also allow formulas which are not the conclusion of any link. We will call these the hypotheses of the proof structure. We will explore the idea of proof nets with hypotheses further in Chapter 7, where we introduce two sided proof structures and proof nets. A proof structure with hypotheses Γ and conclusions Δ will correspond to a two sided sequent $\Gamma \vdash \Delta$.

Given that our proof nets are allowed to have hypotheses, the situation we are in is shown in Figure 4.8.

Now, induction hypothesis gives us a proof \mathcal{D}_1 of $\Gamma' \vdash \Gamma, A, B$ and a proof \mathcal{D}_2 of $\Delta', A \wp B \vdash \Delta$, which we can combine as follows.

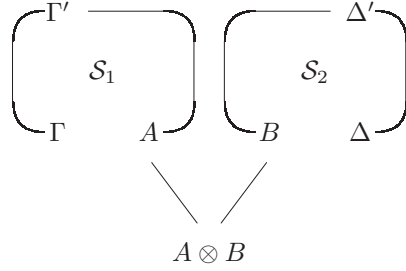


Figure 4.9: Splitting tensor with hypotheses

$$\frac{\frac{\vdots \mathcal{D}_1}{\Gamma' \vdash \Gamma, A, B} \quad \frac{\vdots \mathcal{D}_2}{\Delta', A \wp B \vdash \Delta} [\wp]}{\Gamma', \Delta' \vdash \Gamma, \Delta} [Cut]$$

If the proof net has no par links, then acyclicity and connectedness guarantee every terminal tensor link is splitting. So we are in the situation shown in Figure 4.9.

Induction hypothesis gives us a proof \mathcal{D}_1 of $\Gamma' \vdash \Gamma, A$ and a proof \mathcal{D}_2 of $\Delta' \vdash B, \Delta$. We can combine these proofs as follows to produce a proof of $\Gamma', \Delta' \vdash \Gamma, A \otimes B, \Delta$.

$$\frac{\vdots \mathcal{D}_1 \quad \vdots \mathcal{D}_2}{\Gamma' \vdash \Gamma, A \quad \Delta' \vdash B, \Delta} [\otimes] \quad \square$$

4.4 Cut Elimination

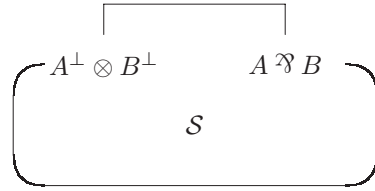
Because we have factored out all rule permutations which were present in the sequent calculus, proving cut elimination for the proof net calculus turns out to be quite simple, as shown by Girard (1987).

Before we prove cut elimination, we show in the following lemma we can restrict ourselves to proof nets with axiom links on atomic formulas without loss of generality.

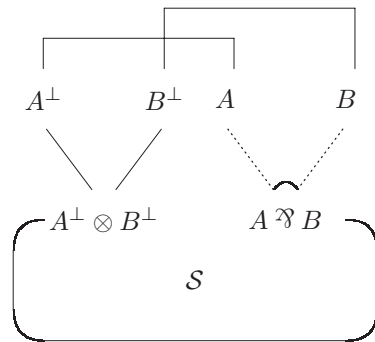
Lemma 4.10 *For each proof net S there exists a proof net S' with the same conclusions as S where all axiom links are atomic. We call this proof net eta expanded.*

Proof By induction on the total number of connectives of formulas which are the conclusions of non-atomic axiom links.

A proof net S with at least one complex axiom link has to be of the form.



We can replace this proof net by a proof net with axiom links on the direct subformulas of the conclusion of the axiom link as follows



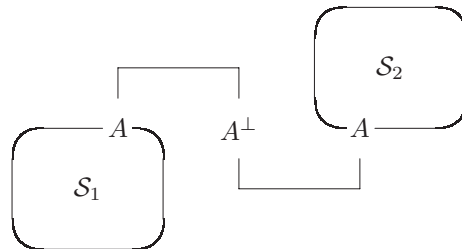
where we have reduced the total number of connectives of conclusions of axiom links by two.

We can see that any switching of the new par link will produce a path from the formula $A^\perp \otimes B^\perp$ to the formula $A \wp B$, so every correction graph of the new net will be acyclic and connected. \square

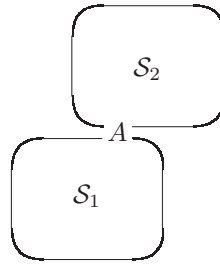
Theorem 4.11 For every proof net S there exists a proof net S' with the same conclusions as S which does not contain cut links.

Proof First, we replace S by S'' which is its eta expanded counterpart according to Lemma 4.10. We proceed by induction on the total number n of connectives of formulas which are premisses to cut links.

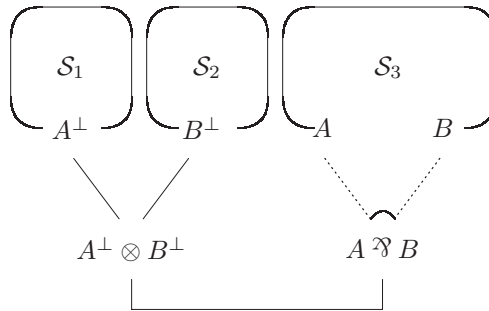
$[n = 0]$ We can successively remove all remaining cut links, which must be of the following form.



It is easy to see we can replace this proof net by the following, while maintaining acyclicity and connectedness.

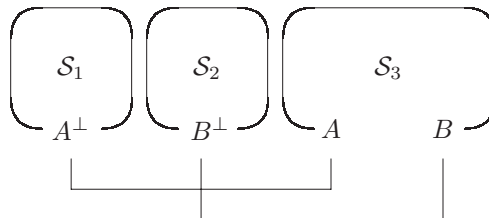


[$n > 0$] When we have a complex cut, our proof net has to be of the following form. The possibility that $A^\perp \otimes B^\perp$ or $A \wp B$ is the conclusion of an axiom link is excluded because our proof net is eta expanded.



No correction graph of this proof net can have connections between S_1 and S_2 , other than through $A^\perp \otimes B^\perp$, otherwise the proof structure would be cyclic. Similarly for S_1 and S_3 and for S_2 and S_3 . We also know that every correction graph of this proof net needs to have a path from A to B which is completely inside S_3 .

Now, when we replace the cut link by the following two cut links, the total complexity decreases by 2.



We show that this new proof structure is also a proof net, that is that for every correction graph and any two vertices in the correction graph

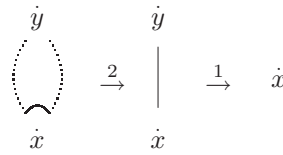


Figure 4.10: Graph contractions

there is a unique path which connects them. Take an arbitrary correction graph and two arbitrary vertices v_1 and v_2 in it.

If both are in the same component, they are connected by virtue of being connected for every correction graph of the original proof net.

If v_1 is in \mathcal{S}_1 and v_2 in \mathcal{S}_3 , then we know there is a unique path from v_1 to A^\perp and a unique path from v_2 to A . We can combine these two paths with the cut link to produce a unique path from v_1 to v_2 . Similarly for v_1 in \mathcal{S}_2 and v_2 in \mathcal{S}_3 .

In the final case v_1 is in \mathcal{S}_1 and v_2 in \mathcal{S}_2 . Now we know there is a unique path from v_1 to A^\perp , which we extend by the cut link to A . We also know there is a unique path from A to B through \mathcal{S}_3 , which we extend again with the cut link to B^\perp . Finally we know there is a unique path from B^\perp to v_2 . Combining these paths gives us the unique path from v_1 to v_2 we need. Notice that in the original proof net this path went directly through the formula $A^\perp \otimes B^\perp$. \square

4.5 Contractions

For a proof net with p par links, there will be 2^p different correction graphs, so naive application of the acyclicity and connectedness criterion will give us an exponential algorithm. Danos (1990) gives us a computationally more attractive method for checking whether a proof structure is acyclic and connected. In Chapter 7 we will adapt this criterion for the multimodal Lambek calculus.

Starting with the graph of the proof structure, we apply the contractions shown in Figure 4.10, with the following conditions.

($\xrightarrow{1}$) only if $x \neq y$

($\xrightarrow{2}$) only if the two edges come from the same link.

It is important to note the edges of a par link are paired, as suggested by the arc connecting them. This means that when multiple par links have the

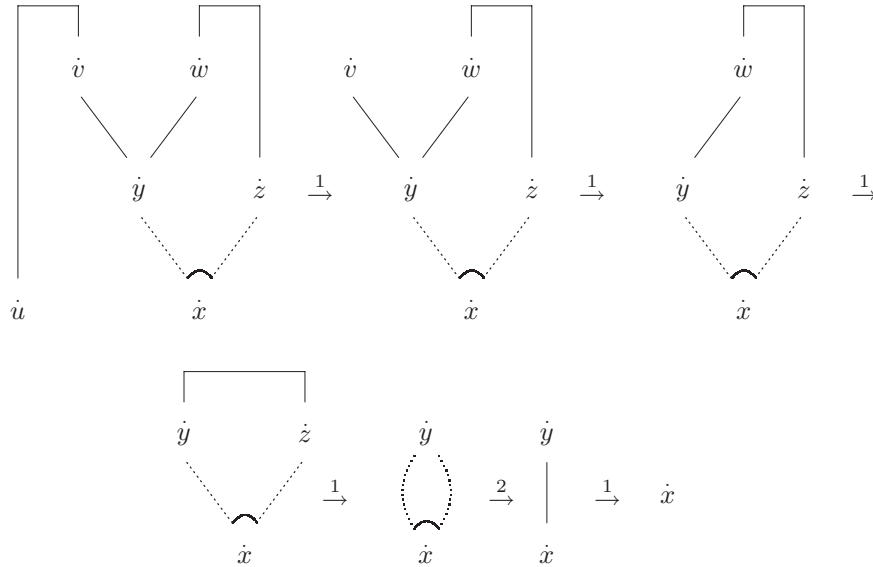


Figure 4.11: Contraction of a proof net

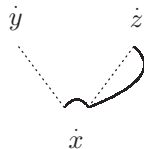
same vertex as a base, which can happen after applying some contractions, we keep track of which pairs belong together.

It is immediate that whenever it is possible to apply more than one contraction to a graph, the results will converge as the conflicting contractions will either 1) produce isomorphic graphs immediately, or 2) the two conversions are locally confluent, that is, applying conversion a after b and applying conversion b after a produces the same result.

Contraction of a proof net will result in a single vertex.

Example 4.12 An example of the contraction of a proof net is shown in Figure 4.11.

Reduction of a proof structure which is not a proof net will result in a graph which is not a single vertex. The incorrect proof structure of Figure 4.2 on page 53 will reduce in 4 steps to



which can be further reduced by a single 1 reduction after which no reductions are possible. This means we can never apply the 2 reduction to eliminate the par link and contract to a single vertex.

Theorem 4.13 A proof structure S is a proof net iff its graph reduces to a single vertex by applying reductions 1 and 2 above.

Proof

[\Rightarrow] We prove that for all proof nets its graph can be reduced to a single vertex. We do this by following the inductive proof net definition.

(Axiom) The proof net consists just of an axiom link, which we can reduce by a 1 reduction to a single vertex.

(Unary) We add a unary link to a proof net which reduces to a vertex. To the resulting graph we can apply a 1 reduction, resulting in a single vertex.

(Times) We have two proof nets reducing to a vertex. After adding the times link we can just apply the 1 reduction two times, and the resulting graph consists of a single vertex.

(Par) We have a proof net which reduces to a single vertex. Adding the par link will give us a 2 redex. Applying a 2 reduction followed by a 1 reduction gives us a single vertex.

(Cut) We have two proof nets reducing to a vertex. We can apply a 1 reduction to the cut link, which gives us a single vertex.

[\Leftarrow] It is easy to see that whenever we apply one of the reductions to reduce a proof structure S to a proof structure S' then S is acyclic and connected iff S' is. A single vertex is acyclic and connected, so if S reduces in a number of steps to a single vertex it is acyclic and connected. Application of theorem 4.9 gives us that S is also a proof net. \square

Beginning with a proof structure with n par links and m times links, we can first reduce all times links in $O(m)$ time. Then we can find a par link to which reduction 2 and 1 can be applied in at most $O(n)$ time. If such a par link cannot be found, we fail because the proof structure is disconnected. Reducing all par links will then take $n + (n - 1) + \dots + 1 (= \frac{1}{2}n(n + 1))$ time. The maximum time for determining a proof structure is a proof net will then be $O(\frac{1}{2}n(n + 1) + m) = O(n^2)$.

Two recent algorithms, proposed by Guerrini (1999) and by Murawski & Ong (2000), check correctness for multiplicative proof nets in linear time.

4.6 The Intuitionistic Fragment

It is relatively simple to adapt the classical proof nets we've seen so far to the intuitionistic fragment of multiplicative linear logic. This basically amounts to a restriction on the formulas of classical linear logic.

Definition 4.14 *A one sided intuitionistic sequent is of the form $\vdash \mathcal{N}_0, \dots, \mathcal{N}_n, \mathcal{P}$, where the \mathcal{N} and \mathcal{P} formulas are defined over a set of atomic formulas \mathcal{A} as follows.*

$$\begin{aligned} \mathcal{N} ::= & \mathcal{A}^\perp \\ & | \mathcal{P} \otimes \mathcal{N} \\ & | \mathcal{N} \wp \mathcal{N} \end{aligned}$$

$$\begin{aligned} \mathcal{P} ::= & \mathcal{A} \\ & | \mathcal{N} \wp \mathcal{P} \\ & | \mathcal{P} \otimes \mathcal{P} \end{aligned}$$

We will call the formulas of \mathcal{N} negative formulas and the formulas of \mathcal{P} positive formulas.

With this definition in hand, we can translate a multiplicative intuitionistic sequent

$$A_1, \dots, A_n \vdash B$$

into a one sided sequent

$$\vdash \|A_1\|^{-}, \dots, \|A_n\|^{-}, \|B\|^{+}$$

as follows.

$$\begin{aligned} \|a\|^{+} &= a \\ \|A \multimap B\|^{+} &= \|A\|^{-} \wp \|B\|^{+} \\ \|A \otimes B\|^{+} &= \|A\|^{+} \otimes \|B\|^{+} \\ \|a\|^{-} &= a^{\perp} \\ \|A \multimap B\|^{-} &= \|A\|^{+} \otimes \|B\|^{-} \\ \|A \otimes B\|^{-} &= \|A\|^{-} \wp \|B\|^{-} \end{aligned}$$

Note that the translation function $\|\cdot\|^{+}$ produces only formulas of \mathcal{P} and that the translation function $\|\cdot\|^{-}$ produces only formulas of \mathcal{N} .

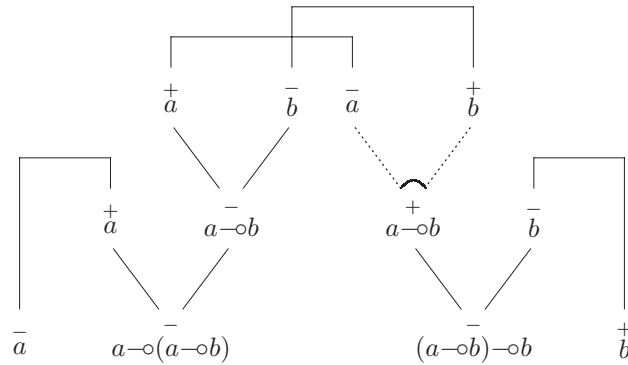
Instead of translating an intuitionistic sequent first into a classical sequent and then constructing the proof net, we can also construct an intuitionistic proof net directly, using polarity labels to indicate if we are operating on a positive or a negative formula. The links for an intuitionistic proof structure are shown in Table 4.3.

Using these links, the proof nets we are generating differ from classical proof nets built with intuitionistic formulas only in the formulas.

Example 4.15 For the intuitionistic sequent

$$a, a \multimap (a \multimap b), (a \multimap b) \multimap b \vdash b$$

we can construct the following proof net



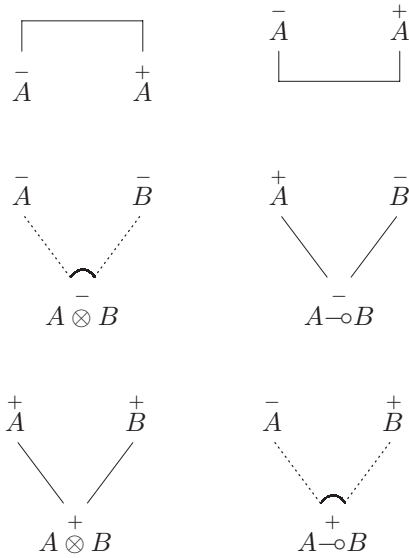
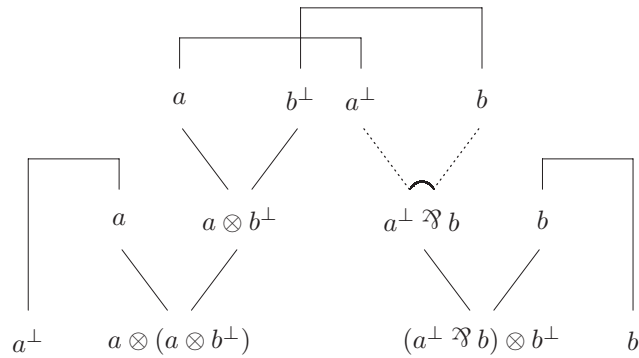


Table 4.3: Links for MILL

Translating the sequent above to a classical sequent would produce

$$\vdash a^\perp, a \otimes (a \otimes b^\perp), (a^\perp \wp b) \otimes b^\perp, b$$

and the corresponding proof net would be the following.



4.7 Non-commutative Proof Nets

As already noted by Girard (1987), it is possible to give a graph theoretic characterization of proof nets for non-commutative multiplicative linear logic as well. Non-commutative, or cyclic linear logic is obtained by replacing the commutativity rule

$$\frac{\vdash \Gamma, B, A, \Delta}{\vdash \Gamma, A, B, \Delta} [P]$$

by the cyclic permutation rule.

$$\frac{\vdash \Gamma, A}{\vdash A, \Gamma} [\text{Cyc}]$$

Cyclic permutations allow us to move formulas to the front or the back of the sequent, but the cyclic ordering of the formulas will remain the same.

For a one sided sequent calculus for cyclic linear logic, we have to realize that negation, when it distributes over a multiplicative connective due to the de Morgan laws, reverses the order of the subformulas, as follows.

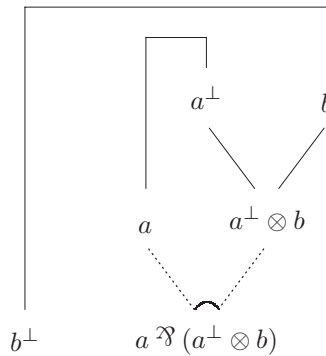
$$\begin{aligned} A^{\perp\perp} &= A \\ (A \otimes B)^{\perp} &= B^{\perp} \wp A^{\perp} \\ (A \wp B)^{\perp} &= B^{\perp} \otimes A^{\perp} \end{aligned}$$

Non-commutative proof nets, then, are those for which the axioms links are planar.

Example 4.16 We can derive the sequent $\vdash b^{\perp}, a \wp (a^{\perp} \otimes b)$ in cyclic linear logic as follows.

$$\frac{\frac{\frac{\frac{}{\vdash a, a^{\perp}} [Ax]}{\vdash a, a^{\perp} \otimes b, b^{\perp}} [\otimes]}{\vdash b^{\perp}, a, a^{\perp} \otimes b} [\text{Cyc}]}{\vdash b^{\perp}, a \wp (a^{\perp} \otimes b)} [\wp]}{\vdash b^{\perp}, a^{\perp} \otimes b} [\otimes]}{\vdash b^{\perp}, b^{\perp}} [Ax]}{\vdash b^{\perp}, a^{\perp}} [Ax]}$$

The proof net corresponding this sequent is the following.



We can translate the formulas of \mathbf{L}_e , which is the Lambek calculus which allows for empty antecedent derivations, into formulas of cyclic linear logic as follows.

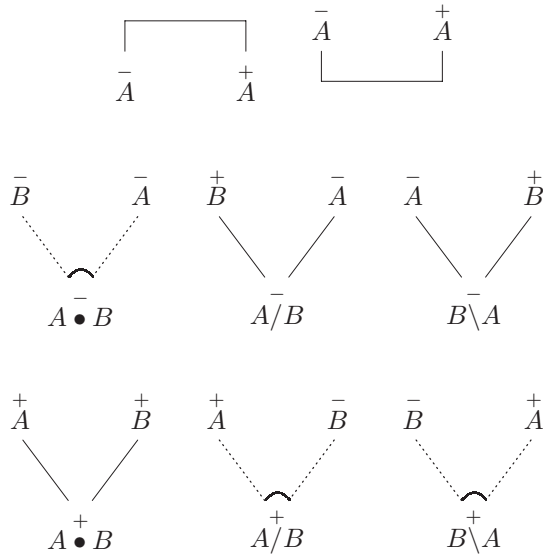
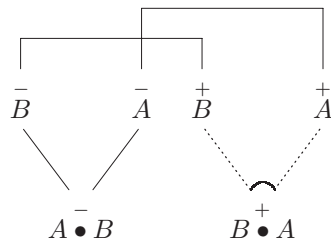


Table 4.4: Links for L_ϵ

$$\begin{aligned} \|A \bullet B\| &= \|A\| \otimes \|B\| \\ \|A/B\| &= \|A\| \wp (\|B\|)^\perp \\ \|B \setminus A\| &= (\|B\|)^\perp \wp \|A\| \end{aligned}$$

When we look at the non-commutative, intuitionistic fragment of linear logic, we extend the non-commutative links above with polarity marking, just as we did for multiplicative intuitionistic linear logic in the previous section. The links for L_ϵ are shown in Table 4.4.

Example 4.17 *One characteristic non-theorem of L is $A \bullet B \not\vdash B \bullet A$. There is only one proof structure for this sequent, which is shown below and which is not planar.*



Theorem 4.18 (Roorda (1991)) *A proof net is valid in L_ϵ iff*

- all its correction graphs are acyclic and connected,
- all its axiom links are planar.

See also Lamarche & Retoré (1996) for a good overview of proof nets for the Lambek calculus. Abrusci & Ruet (1999) present a combined commutative/non-commutative version of multiplicative linear logic, essentially a classical version of the intuitionistic calculus of de Groote (1996). They give a proof net calculus for their logic with a correctness criterion based on Girard's (1987) long trip condition. However, it is unclear if we can extend this methodology to non-associative or multimodal logics.

4.8 Conclusions

We have seen how proof nets function as a sort of parallel proof theory for **MLL** and how proof nets for **MILL** can be obtained as a natural fragment of this calculus. We have also seen a natural way of dealing with non-commutativity. In the next chapters we will adapt these proof nets to several different settings. In Chapter 5, we will add the first order quantifiers to the proof net calculus and use them to encode string positions and locality domains. In Chapter 6, we will use *labeling* as a way of enforcing word order and structural constraints. Finally, in Chapter 7 we will look at a version of the contraction criterion discussed in Section 4.5 which is sound and complete for the multimodal Lambek calculus.

CHAPTER 5

PROOF NETS FOR FIRST ORDER LINEAR LOGIC

IN this chapter we will discuss the first order multiplicative intuitionistic fragment of linear logic, **MILL1**, and its applications to linguistics. The first order multiplicative fragment is attractive because its computational complexity is NP complete, just like the propositional multiplicative fragment and because there is a proof net calculus with a correctness criterion which is a natural extension of the ‘switching’ criterion of Theorem 4.9.

We give an embedding translation from formulas in the Lambek calculus to formulas in **MILL1** and show this translation is sound and complete. We then exploit the extra power of the first order fragment to give an account of a number of linguistic phenomena which have no satisfactory treatment in the Lambek calculus.

This chapter is based on joint work with Mario Piazza (Moot & Piazza 2001).

5.1 Proof Theory

Definition 5.1 (Language) *The language $\mathcal{L}(\mathbf{MILL1})$ is defined as follows.*

[Alphabet] *The alphabet consists of the following symbols: countably many individual variables x_0, x_1, \dots , countably many individual constants c_0, c_1, \dots , symbols for functors of different arities, the binary connectives ‘ \otimes ’ and ‘ $-\circ$ ’, the quantifiers ‘ \forall ’ and ‘ \exists ’, the sequent symbol ‘ \vdash ’ and the auxiliary symbols ‘ $;$ ’, ‘ $;$ ’, ‘ $($ and ‘ $)$ ’.*

[Terms] *If f is a function symbol of arity n , and e_1, \dots, e_n are (not necessarily distinct) variables and constants, then $f(e_1, \dots, e_n)$ is a term.*

If f has arity 0 we will call its term a proposition.

$$\begin{array}{c}
\frac{}{A \vdash A} [Ax] \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash C}{\Gamma, \Delta \vdash C} [Cut] \\
\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} [L\otimes] \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} [R\otimes] \\
\frac{\Delta \vdash A \quad \Gamma, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} [L\multimap] \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} [R\multimap] \\
\frac{\Gamma, A \vdash C}{\Gamma, \exists x.A \vdash C} [L\exists] \qquad \frac{\Gamma \vdash A[x := e]}{\Gamma \vdash \exists x.A} [R\exists] \\
\frac{\Gamma, A[x := e] \vdash C}{\Gamma, \forall x.A \vdash C} [L\forall] \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} [R\forall]
\end{array}$$

Table 5.1: The sequent calculus MILL1

[Formulas] Given the set \mathcal{T} of terms as defined above and the set \mathcal{V} of variables, the formulas in our language are the following

$$\mathcal{F} ::= \mathcal{T} \mid \mathcal{F} \otimes \mathcal{F} \mid \mathcal{F} \multimap \mathcal{F} \mid \exists \mathcal{V}.\mathcal{F} \mid \forall \mathcal{V}.\mathcal{F}$$

[Sequents] If Γ is a multiset of formulas separated by ‘;’ and C is a formula then $\Gamma \vdash C$ is a sequent. By taking Γ as a multiset we will implicitly assume that the sequent comma ‘;’ is associative and commutative. We will call Γ the antecedent of the sequent and C the succedent.

5.1.1 Sequent Calculus

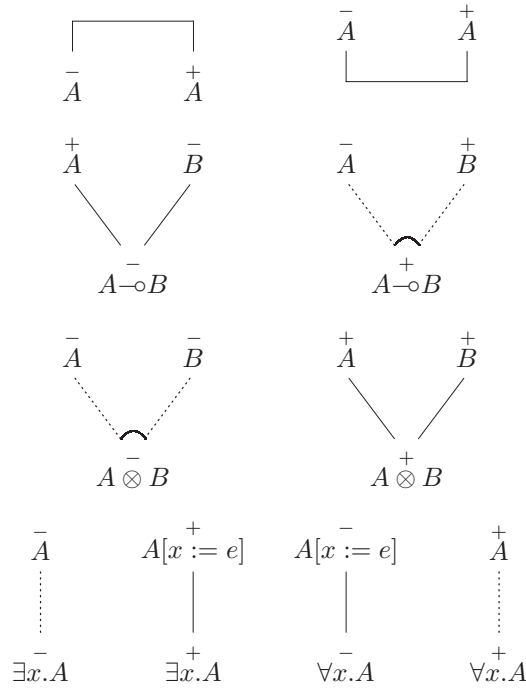
The sequent calculus for MILL1 is given by the rules shown in Table 5.1.

The variable x in the quantifier rules is called the *eigenvariable* of the rule. The rules $[R\forall]$ and $[L\exists]$ have the condition that this variable does not occur freely in Γ or C .

The notation $A[x := e]$ in the rules $[L\forall]$ and $[R\exists]$ signifies the formula A with a variable or constant (of our choice) e substituted for all occurrences of the variable x . When we read these rules from conclusion to premiss, as in backward chaining proof search, we only have to consider substituting a finite number of variables and constants for x , namely either those actually occurring in Γ or C or a fresh variable or constant. In practice we want to delay making a choice for e until we reach the axioms.

We will adopt the following conventions on variables.

- Each quantifier will have a different eigenvariable. We can accommodate for this condition by selecting a different variable name for each application of a quantifier rule.
- The end-sequent of a proof does not contain occurrences of free variables. We can replace these free variables by constants not occurring elsewhere in the proof.

Table 5.2: Links for **MILL1**

The calculus enjoys elimination of the *Cut* rule and the cut-free formulation of **MILL1** can be used as an algorithm for proof search. Interestingly, as noted by Lincoln (1995), adding quantifiers to the multiplicative fragment of linear logic does not change the complexity of deciding whether a sequent is provable. A nondeterministic machine can try all possibilities for the rules $[L\forall]$ and $[R\exists]$ and find a solution in polynomial time. The NP hardness of the propositional fragment then gives us an NP completeness result for first order multiplicative linear logic.

5.1.2 Proof Structures

Definition 5.2 A proof structure $\langle S, \mathcal{L} \rangle$ for first order multiplicative intuitionistic linear logic consists of a set S of formulas and a set *call* of links in S of the forms shown in Table 5.2, subject to the following conditions.

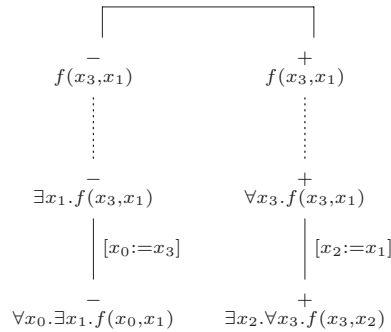
- (i) Each formula is the conclusion of exactly one link.
- (ii) Each formula is the premiss of at most one link.
- (iii) Each quantifier link uses a distinct eigenvariable. If a variable occurs freely in some formula of the structure then this variable is the eigenvariable of a link.

- (iv) *Those formulas which are not the premiss of any link are the conclusions of the proof structure. Conclusions must be closed.*

5.1.3 Proof Nets

While we can show by induction on the length of the proof that we can associate a proof structure with every sequent proof, it should be obvious that not all proof structures correspond to sequent proofs.

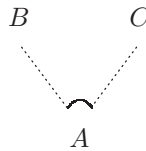
Example 5.3 *Though the sequent $\forall x_0.\exists x_1.f(x_0, x_1) \vdash \exists x_2.\forall x_3.f(x_3, x_2)$ is undrivable, it corresponds to the following proof structure.*



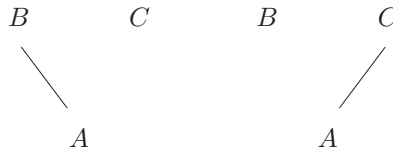
A subset of proof structures, which we will call *proof nets*, does correspond to sequent proofs. We can distinguish proof nets from other proof structures by looking only at properties of the underlying graphs of proof structures.

Definition 5.4 *From a proof structure we obtain a correction graph by*

- (i) *replacing all links*



by one of the following links.



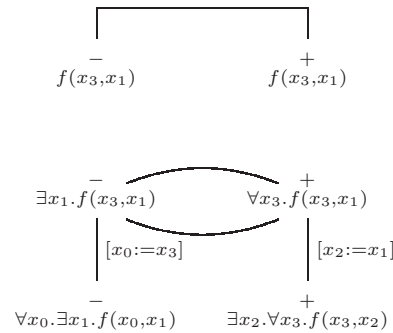
(ii) replacing all links



with eigenvariable x by a link from A to any formula in which x occurs freely or by a link from A to B .

Theorem 5.5 (Girard (1991)) A proof structure is a proof net if and only if all its correction graphs are acyclic and connected.

Example 5.6 Returning to our previous example, we should be able to find a correction graph of that proof structure which violates the proof net condition. Of the 9 correction graphs we can associate with that proof structure, the following



is both disconnected and cyclic.

5.1.4 Embedding the Lambek Calculus

As a prelude to our linguistic applications we will first show a translation of formulas and sequents in the Lambek calculus into formulas of MILL1.

Lambek Calculus

Definition 5.7 (Language) The language $\mathcal{L}(\mathbf{L})$ is the following

[Alphabet] The alphabet consists of the following symbols: a finite set of atomic formulas \mathcal{A} , the binary connectives $'/'$, $'\bullet'$ and $'\setminus'$, the sequent arrow $'\vdash'$ and the auxiliary symbol $'\cdot'$.

[Formulas] The formulas of \mathbf{L} are the following

$$\mathcal{F} ::= \mathcal{A} \mid \mathcal{F} / \mathcal{F} \mid \mathcal{F} \bullet \mathcal{F} \mid \mathcal{F} \setminus \mathcal{F}$$

[Sequents] If Γ is a list of formulas separated by ‘,’ and C is a formula then $\Gamma \vdash C$ is a sequent. By taking Γ as a list instead of a multiset we will assume the sequent comma ‘,’ to be associative and non-commutative.

$$\frac{}{A \vdash A} [Ax] \quad \frac{\Delta \vdash A \quad \Gamma, A, \Gamma' \vdash C}{\Gamma, \Delta, \Gamma' \vdash C} [Cut]$$

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \bullet B, \Delta \vdash C} [L\bullet] \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \bullet B} [R\bullet]$$

$$\frac{\Delta \vdash B \quad \Gamma, A, \Gamma' \vdash C}{\Gamma, A/B, \Delta, \Gamma' \vdash C} [L/] \quad \frac{\Gamma, B \vdash A}{\Gamma \vdash A/B} [R/]$$

$$\frac{\Delta \vdash B \quad \Gamma, A, \Gamma' \vdash C}{\Gamma, \Delta, B \setminus A, \Gamma' \vdash C} [L\setminus] \quad \frac{B, \Gamma \vdash A}{\Gamma \vdash B \setminus A} [R\setminus]$$

For linguistic reasons, the Lambek calculus is usually defined with the additional condition that all antecedents are non-empty. In the same way, we can reformulate **MILL1** with a restriction to non-empty antecedents and impose a similar restriction on proof nets.

We can also *allow* empty antecedents and disallow them by means of an extra argument whenever empty antecedent derivations would be undesirable, using the underivability of the following sequent.

$$\not\vdash \exists x_0. a(\dots, x_0) \multimap \forall x_1. a(\dots, x_1)$$

This strategy is actually an instance of the locality domains we will introduce in more detail in Section 5.2.3. Allowing empty antecedent derivations will be useful in Section 5.2.2, where we give an account of pied piping.

Translation

For the translation, we replace (propositional) atomic formulas in **L** by binary terms in **MILL1**. The two extra arguments will encode the (abstract) list positions the formula occupies. This is closely related to the standard Logic Programming practice of representing a partial list by a pair of arguments, often called a *difference list* (see, for example, Pereira & Shieber (1987)). Our translation can also be seen as an extension of the translation proposed by Pareschi (1988), who proposes a translation of product-free Lambek calculus formulas into (extended) definite clauses.

A *sequent* in the Lambek calculus $A_1, \dots, A_n \vdash B$ will be translated as $\|A_1\|^{(c_0, c_1)}, \dots, \|A_n\|^{(c_{n-1}, c_n)} \vdash \|B\|^{(c_0, c_n)}$.

Definition 5.8 The embedding translation $\|\cdot\|^{(e_i, e_j)}$ from $\mathcal{F}(\mathbf{L})$ to $\mathcal{F}(\mathbf{MILL1})$ is defined as follows.

$$\begin{aligned}
\|a\|^{\langle e_i, e_j \rangle} &= a(e_i, e_j) \\
\|A/B\|^{\langle e_i, e_j \rangle} &= \forall x_k. \|B\|^{\langle e_j, x_k \rangle} \multimap \|A\|^{\langle e_i, x_k \rangle} \\
\|B \setminus A\|^{\langle e_i, e_j \rangle} &= \forall x_k. \|B\|^{\langle x_k, e_i \rangle} \multimap \|A\|^{\langle x_k, e_j \rangle} \\
\|A \bullet B\|^{\langle e_i, e_j \rangle} &= \exists x_k. \|A\|^{\langle e_i, x_k \rangle} \otimes \|B\|^{\langle x_k, e_j \rangle}
\end{aligned}$$

Note that this definition guarantees that the translation of any non-atomic formula is of the form $Qx.A\#B$, where Q is a quantifier and $\#$ a binary connective.

Lemma 5.9 *If an L sequent is translated into a sequent provable in MILL1, then there is also a MILL1 proof of the same translated sequent where every conclusion of a rule with main formula $A\#B$ is the premiss of a rule with main formula $Qx.A\#B$.*

Proof We show only the case for formulas $\exists x.A \otimes B$. The cases for the implications are similar.

Suppose we have a subproof of the form

$$\begin{array}{c}
\frac{\Delta \vdash A \quad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B} [R\otimes] \\
\vdots \Pi \\
\frac{\Gamma \vdash A \otimes B}{\Gamma \vdash \exists x.A \otimes B} [R\exists]
\end{array}$$

we can move the application of the $[R\exists]$ rule up as follows

$$\begin{array}{c}
\frac{\frac{\Delta \vdash A \quad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B} [R\otimes]}{\Delta, \Delta' \vdash \exists x.A \otimes B} [R\exists] \\
\vdots \Pi \\
\Gamma \vdash \exists x.A \otimes B
\end{array}$$

as applying the rules in Π with $\exists x.A \otimes B$ instead of $A \otimes B$ is unproblematic because we have one free variable *less*.

For the other case, suppose we have a subproof of the form

$$\begin{array}{c}
\frac{\Delta, A, B \vdash D}{\Delta, A \otimes B \vdash D} [L\otimes] \\
\vdots \Pi \\
\frac{\Gamma, A \otimes B \vdash C}{\Gamma, \exists x.A \otimes B \vdash C} [L\exists]
\end{array}$$

we can move the application of the $[L\otimes]$ rule down as follows

$$\begin{array}{c}
\Delta, A, B \vdash D \\
\vdots \Pi \\
\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} [L\otimes] \\
\frac{\Gamma, A \otimes B \vdash C}{\Gamma, \exists x.A \otimes B \vdash C} [L\exists]
\end{array}$$

which again gives us a valid proof. \square

Theorem 5.10 *A sequent $A_1, \dots, A_n \vdash B$ is derivable in **L** if and only if the sequent $\|A_1\|^{(c_0, c_1)}, \dots, \|A_n\|^{(c_{n-1}, c_n)} \vdash \|B\|^{(c_0, c_n)}$ is derivable in **MILL1**.*

Proof Because of the reordering result of Lemma 5.9, this proof is a simple induction consisting of replacing a single rule like

$$\frac{\Gamma, B \vdash A}{\Gamma \vdash A/B} [R/]$$

by a pair of rules

$$\frac{\frac{\|\Gamma\|^{(c_0, c_n)}, \|B\|^{(c_n, x_0)} \vdash \|A\|^{(c_0, x_0)}}{\|\Gamma\|^{(c_0, c_n)} \vdash \|B\|^{(c_n, x_0)} \multimap \|A\|^{(c_0, x_0)}} [R-\multimap]}{\|\Gamma\|^{(c_0, c_n)} \vdash \forall x_0. \|B\|^{(c_n, x_0)} \multimap \|A\|^{(c_0, x_0)}} [R\forall]}$$

and vice versa¹. \square

Readers who are familiar with the relational completeness result of Kurtonina (1995, chapter 4) will note that the formula translation is a variant of the one proposed there. Because all theorems of **MILL1** are also theorems of classical first order logic, this essentially made proving the completeness direction of Theorem 5.10 unnecessary. As the emphasis of this article is on proof theory, we chose to give a syntactic proof of this result.

The Lambek Calculus With Permutation

When we add the following Permutation rule to the Lambek calculus

$$\frac{\Gamma, B, A, \Delta \vdash C}{\Gamma, A, B, \Delta \vdash C} [P]$$

the resulting logic **LP** is just a syntactic variant of **MILL**, if we interpret both ‘/’ and ‘\’ as ‘ \multimap ’ and ‘ \bullet ’ as ‘ \otimes ’. We can also use the following variation on the translation from **L** into **MILL1**

$$\begin{aligned} \|a\|_{\mathbf{LP}}^{(e_i, e_j)} &= a(e_i, e_j) \\ \|A/B\|_{\mathbf{LP}}^{(e_i, e_j)} &= \forall x_k. \forall x_l. \forall x_m. \forall x_n. \|B\|^{(x_k, x_l)} \multimap \|A\|^{(x_m, x_n)} \\ \|B \backslash A\|_{\mathbf{LP}}^{(e_i, e_j)} &= \forall x_k. \forall x_l. \forall x_m. \forall x_n. \|B\|^{(x_k, x_l)} \multimap \|A\|^{(x_m, x_n)} \\ \|A \bullet B\|_{\mathbf{LP}}^{(e_i, e_j)} &= \exists x_k. \exists x_l. \exists x_m. \exists x_n. \|A\|^{(x_k, x_l)} \otimes \|B\|^{(x_m, x_n)} \end{aligned}$$

where we quantify over all positions of subformulas.

¹There is a small technicality involved with constants versus free variables. The induction hypothesis gives us a proof with c_{n+1} instead of x_0 , but it is always possible to replace a constant by a variable which does not occur elsewhere in the proof. As a consequence of our translation all occurrences of c_{n+1} will be in $\|A\|$ and $\|B\|$, so the end-sequent is again closed.

The two translations give us a logic where translated formulas of **LP** and **L** coexist and interact in non-trivial ways. A consequence of the above translation is, for example, that $\|A/B\|_{\text{LP}} \vdash \|A/B\|_{\text{L}}$ is a theorem but the converse is not.

The Non-associative Lambek Calculus

There are linguistic reasons to restrict even the structural rule of associativity (which is implicit in the sequent formulation above) from the Lambek calculus. The non-associative Lambek calculus **NL**, introduced in (Lambek 1961), is obtained by using *trees* of formulas as antecedents.

Sequent calculus for **NL** is the following, where the notation $\Gamma[\Delta]$ indicates an antecedent tree Γ with a distinguished subtree occurrence Δ .

$$\begin{array}{c} \frac{}{A \vdash A} [Ax] \quad \frac{\Delta \vdash A \quad \Gamma[A] \vdash C}{\Gamma[\Delta] \vdash C} [Cut] \\ \\ \frac{\Gamma[(A, B)] \vdash C}{\Gamma[A \bullet B] \vdash C} [L\bullet] \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{(\Gamma, \Delta) \vdash A \bullet B} [R\bullet] \\ \\ \frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[(A/B, \Delta)] \vdash C} [L/] \quad \frac{(\Gamma, B) \vdash A}{\Gamma \vdash A/B} [R/] \\ \\ \frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[(\Delta, B \setminus A)] \vdash C} [L\setminus] \quad \frac{(B, \Gamma) \vdash A}{\Gamma \vdash B \setminus A} [R\setminus] \end{array}$$

We could obtain a formulation of **L** equivalent to the one given in Section 5.1.4 by adding the following structural rules for associativity.

$$\frac{\Gamma[(\Delta_1, (\Delta_2, \Delta_3))] \vdash C}{\Gamma[(\Delta_1, \Delta_2), \Delta_3] \vdash C} [Ass1] \quad \frac{\Gamma[(\Delta_1, \Delta_2), \Delta_3] \vdash C}{\Gamma[(\Delta_1, (\Delta_2, \Delta_3))] \vdash C} [Ass2]$$

To recover control over associativity, we give our atomic formulas an extra argument encoding its *depth*. Every depth level has its own constant.

$$\begin{array}{l} \|a\|_{\text{NL}}^{\langle e_i, e_j, d_i \rangle} = a(e_i, e_j, d_i) \\ \|A/B\|_{\text{NL}}^{\langle e_i, e_j, d_i \rangle} = \forall x_k. \|B\|_{\text{NL}}^{\langle e_j, x_k, d_i \rangle} \multimap \|A\|_{\text{NL}}^{\langle e_i, x_k, d_{i-1} \rangle} \\ \|B \setminus A\|_{\text{NL}}^{\langle e_i, e_j, d_i \rangle} = \forall x_k. \|B\|_{\text{NL}}^{\langle x_k, e_i, d_i \rangle} \multimap \|A\|_{\text{NL}}^{\langle x_k, e_j, d_{i-1} \rangle} \\ \|A \bullet B\|_{\text{NL}}^{\langle e_i, e_j, d_i \rangle} = \exists x_k. \|A\|_{\text{NL}}^{\langle e_i, x_k, d_{i+1} \rangle} \otimes \|B\|_{\text{NL}}^{\langle x_k, e_j, d_{i+1} \rangle} \end{array}$$

We translate the sequents themselves as follows:

$$\|\Gamma \vdash C\|_{\text{NL}}^{\langle c_i, c_j, d_i \rangle} = \|\Gamma\|_{\text{NL}}^{\langle c_i, c_j, d_i \rangle} \vdash \|C\|_{\text{NL}}^{\langle c_i, c_j, d_i \rangle}$$

For the antecedent term, we use the following translation function, making sure that c_k on the right hand side is an unused constant.

$$\|(\Gamma, \Delta)\|_{\text{NL}}^{\langle c_i, c_j, d_i \rangle} = \|\Gamma\|_{\text{NL}}^{\langle c_i, c_k, d_{i+1} \rangle}, \|\Delta\|_{\text{NL}}^{\langle c_k, c_j, d_{i+1} \rangle}$$

It is perhaps not obvious that this encoding gives us enough information to reconstruct an antecedent tree at any step, so we first prove the following lemma.

Lemma 5.11 *For any binary tree there is a bijection between the tree and the sequence of the depth of its leaves.*

Proof Surjectivity is trivial, given that we can read off the depths of the leaves of a tree from left to right. What remains to be shown is that any sequence obtained by listing the depth of the leaves of a binary tree from left to right determines a unique binary tree.

A depth sequence for a binary tree is inductively given by a rewrite system with start symbol d_k for some integer k and one rewrite rule

$$d_i \rightarrow d_{i+1}d_{i+1}$$

A depth sequence d_1, \dots, d_n generated from root k with respective depth indices $\text{index}(d_1), \dots, \text{index}(d_n)$ and deepest leaf at depth index m has the following property

$$\sum_{1 \leq i \leq n} 2^{m-\text{index}(d_i)} = 2^{m-k}$$

which we can show by simple induction.

From a depth sequence, we can construct a tree by induction on its length n as follows.

If $n = 1$ our tree consists of a single leaf.

If $n > 1$ our tree has been generated from two subtrees with root d_{k+1} , so we know there exists an x such that

$$\sum_{1 \leq i < x} 2^{m-\text{index}(d_i)} = \sum_{x \leq i \leq n} 2^{m-\text{index}(d_i)} = 2^{m-(k+1)}$$

Suppose now there is a $y \neq x$ which has this same property. This would mean there is a non-empty sequence between x and y such that the sum of 2^{m-i} for all i between x and y is 0. But as any sum of powers of two is strictly greater than 0 this leads to a contradiction. Ergo, there is a unique way to split the sequence into two strictly smaller subsequences which, by the induction hypothesis, correspond to unique binary trees, guaranteeing the uniqueness of the resulting tree. \square

Theorem 5.12 *An NL sequent is derivable if and only if its translation obtained by the function $\|\cdot\|_{\text{NL}}$ is.*

Proof Given that we can reorder our **MILL1** proofs by Lemma 5.9 and that we can construct a unique binary tree from the depths of the formulas by Lemma 5.11, this proof is only a minor modification of Theorem 5.10.

The proof proceeds by induction on the length of the sequent proof as before. We show only one case and only the depth argument. An **NL** inference

$$\frac{\Delta \vdash B \quad \Gamma[A] \vdash C}{\Gamma[(A/B, \Delta)] \vdash C} [L/]$$

would be translated as

$$\frac{\|\Delta\|^{d_k} \vdash \|B\|^{d_k} \quad \|\Gamma[\|A\|^{d_{k-1}}]\|^{d_0} \vdash \|C\|^{d_0}}{\|\Gamma[\|B\|^{d_k} \multimap \|A\|^{d_{k-1}}, \|\Delta\|^{d_k}]\|^{d_0} \vdash \|C\|^{d_0}} [L-\circ]$$

and vice versa. \square

The Non-associative Lambek Calculus With Permutation

The non-associative Lambek calculus with Permutation **NLP** is the logic we get from **NL** by adding an additional rule of permutation. After the embedding results of the previous sections, perhaps the natural question to ask is how this logic would compare to the one we would get by using only the depth argument from the **NL** translation, call it **NLP'**. It turns out that **NLP** is not the same as **NLP'**.

As **NLP** has little linguistic relevance, the fact that the encoding is not easily extensible to handle this logic is not a big restriction on possible applications of our system. It appears, however, that **NLP'** is **NLP** with the following additional structural rule

$$\frac{\Gamma[(\Delta_1, \Delta_3), (\Delta_2, \Delta_4)] \vdash C}{\Gamma[(\Delta_1, \Delta_2), (\Delta_3, \Delta_4)] \vdash C} [Mix]$$

which, in combination with commutativity, allows us to rearrange the multiple binary subtrees in any depth-preserving way.

Unary Connectives

In Section 3.4 we introduced the unary operators ' \diamond ' and ' \square^\downarrow ' which can license or restrain the various structural rules. As our result for **NL** depends on the trees being binary and the unary connectives would introduce unary branches, it is unclear to us if it is possible to give a direct extension of our proposed translation which includes the unary connectives. However, Versmissen (1996) proposes a translation of the unary connectives into binary connectives with the help of two atomic formulas not used elsewhere in the grammar: m , which intuitively corresponds to the bracket ' \langle ' and n , which intuitively corresponds to the bracket ' \rangle '. The translation, from **L** \diamond to **L** is then defined as follows.

$$\begin{aligned} \|a\| &= a \\ \|A/B\| &= \|A\| / \|B\| \\ \|B \setminus A\| &= \|B\| \setminus \|A\| \\ \|A \bullet B\| &= \|A\| \bullet \|B\| \\ \|\square^\downarrow A\| &= (m \setminus \|A\|) / n \\ \|\diamond A\| &= (m \bullet \|A\|) \bullet n \end{aligned}$$

We can combine this translation with one of the previous translations to obtain a system with unary connectives. In Section 5.2.2, we will give an example of how to mimic the way unary connectives give access to different structural rules.

5.2 Linguistic Applications

Because of the embedding results of Section 5.1.4, we can take the lexicon from a Lambek grammar fragment and translate it into an equivalent lexicon for our calculus.

For example, given the lexicon l' for the Lambek calculus

$$\begin{aligned}
l'(\text{Lambek}) &= np \\
l'(\text{Gödel}) &= np \\
l'(\text{Russell}) &= np \\
l'(\text{student}) &= n \\
l'(\text{article}) &= n \\
l'(\text{proof}) &= n \\
l'(\text{the}) &= np/n \\
l'(\text{sleeps}) &= np \backslash s \\
l'(\text{likes}) &= (np \backslash s) / np \\
l'(\text{wrote}) &= (np \backslash s) / np \\
l'(\text{influenced}) &= (np \backslash s) / np \\
l'(\text{sent}) &= ((np \backslash s) / np) / np \\
l'(\text{believes}) &= (np \backslash s) / s \\
l'(\text{yesterday}) &= s \backslash s
\end{aligned}$$

we can translate it into the lexicon l for **MILL1** which is parameterized for two position constants

$$\begin{aligned}
l(\text{Lambek}, c_i, c_j) &= np(c_i, c_j) \\
l(\text{Gödel}, c_i, c_j) &= np(c_i, c_j) \\
l(\text{Russell}, c_i, c_j) &= np(c_i, c_j) \\
l(\text{student}, c_i, c_j) &= n(c_i, c_j) \\
l(\text{article}, c_i, c_j) &= n(c_i, c_j) \\
l(\text{proof}, c_i, c_j) &= n(c_i, c_j) \\
l(\text{the}, c_i, c_j) &= \forall x_0. n(c_j, x_0) \multimap np(c_i, x_0) \\
l(\text{sleeps}, c_i, c_j) &= \forall x_0. np(x_0, c_i) \multimap s(x_0, c_j) \\
l(\text{likes}, c_i, c_j) &= \forall x_1. np(c_j, x_1) \multimap \forall x_0. (np(x_0, c_i) \multimap s(x_0, x_1)) \\
l(\text{wrote}, c_i, c_j) &= \forall x_1. np(c_j, x_1) \multimap \forall x_0. (np(x_0, c_i) \multimap s(x_0, x_1)) \\
l(\text{influenced}, c_i, c_j) &= \forall x_1. np(c_j, x_1) \multimap \forall x_0. (np(x_0, c_i) \multimap s(x_0, x_1)) \\
l(\text{sent}, c_i, c_j) &= \forall x_2. np(c_j, x_2) \multimap \forall x_1. (np(x_2, x_1) \multimap \\
&\quad \forall x_0. (np(x_0, c_i) \multimap s(x_0, x_1))) \\
l(\text{believes}, c_i, c_j) &= \forall x_1. s(c_j, x_1) \multimap \forall x_0. (np(x_0, c_i) \multimap s(x_0, x_1)) \\
l(\text{yesterday}, c_i, c_j) &= \forall x_0. s(x_0, c_i) \multimap s(x_0, c_j)
\end{aligned}$$

with which we can derive, for example, ‘Russell wrote the article’ as a formula of type s as shown in Figure 5.1 on the next page.

For reasons of space, we have written only the main connective at each node of the proof structure and indicated the substitutions next to the unary links.

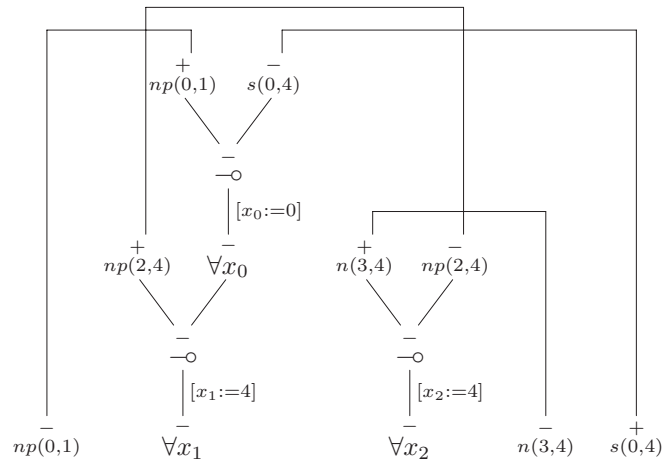


Figure 5.1: Proof net for ‘Russell wrote the article’

In the remainder of this section we will discuss a number of linguistic phenomena which have no satisfactory treatment in **L**, and sketch how they can be treated in the more expressive framework we are proposing here.

5.2.1 Quantifier Scope

It has been noted since Montague (1974) that quantifiers like ‘someone’ and ‘everyone’, though they occupy *np* positions in a sentence, may take scope at the sentence level.

In the Lambek calculus, we can account for some of the consequences of this fact by giving a quantifier two assignments in the lexicon: one when it occurs as an subject, and one for when it occurs as a (direct) object.

$$\begin{aligned} l'(\textit{someone}) &= s/(np \setminus s) \\ &= (s/np) \setminus s \\ l'(\textit{everyone}) &= s/(np \setminus s) \\ &= (s/np) \setminus s \end{aligned}$$

These assignments allow us to derive classic sentences like

(5.1) Someone sleeps.

(5.2) Someone wrote the article.

(5.3) Everyone likes someone.

but not

(5.4) Russell sent everyone the article.

Another problem arises with the following example. In Montague semantics, a sentence

(5.5) Gödel believes someone sleeps

will have two readings, one in which there exists a specific person whom Gödel believes to be sleeping (*de re* reading) and one in which an unknown person is believed to be sleeping (*de dicto* reading).

In the standard Lambek calculus, only the second reading can be derived with the lexical assignments above. This means we have to add a new lexical entry for the quantifiers, and another if we want quantifiers to appear in indirect object positions. Oehrle (1994) gives a good discussion of these problems and proposes a solution in the form of string labeling.

Moortgat proposes the q operator for quantification

$$\frac{\Delta, A, \Delta' \vdash B \quad \Gamma, C, \Gamma' \vdash D}{\Gamma, \Delta, q(A, B, C), \Delta', \Gamma' \vdash D} [Lq]$$

A problem with this operator is that it only allows a left rule to be formulated, which makes it hard to view it as a logical connective. Decompositions of the q connective into proper logical connectives, combined with an appropriate package of structural rules, have been proposed in (Morrill 1994) and (Moortgat 1996a).

We can extend the translation function $\|\cdot\|^{(e_i, e_j)}$ to translate $q(A, B, C)$ into **MILL1** as follows.

$$\|q(A, B, C)\|^{(e_i, e_j)} = \forall x_0. \forall x_1. (\|A\|^{(e_i, e_j)} \multimap \|B\|^{(x_0, x_1)}) \multimap \|C\|^{(x_0, x_1)}$$

After which the $[Lq]$ rule becomes a derived rule in the following way.

$$\frac{\frac{\frac{\|\Delta\|^{(i,j)}, \|A\|^{(j,j+1)}, \|\Delta'\|^{(j+1,k)} \vdash \|B\|^{(i,k)}}{\|\Delta\|^{(i,j)}, \|\Delta'\|^{(j+1,k)} \vdash \|A\|^{(j,j+1)} \multimap \|B\|^{(i,k)}} [R\multimap]}{\|\Gamma\|^{(1,i)}, \|\Delta\|^{(i,j)}, (\|A\|^{(j,j+1)} \multimap \|B\|^{(i,k)}) \multimap \|C\|^{(i,k)}, \|\Delta'\|^{(j+1,k)}, \|\Gamma'\|^{(k,l)} \vdash \|D\|^{(1,l)}} [L\multimap]}{\|\Gamma\|^{(1,i)}, \|\Delta\|^{(i,j)}, \forall x_0. \forall x_1. (\|A\|^{(j,j+1)} \multimap \|B\|^{(x_0, x_1)}) \multimap \|C\|^{(x_0, x_1)}, \|\Delta'\|^{(j+1,k)}, \|\Gamma'\|^{(k,l)} \vdash \|D\|^{(1,l)}} [L\forall]} [L\forall]$$

In **MILL1** we can assign generalized quantifiers the following lexical formula

$$\begin{aligned} l(\text{someone}, c_i, c_j) &= \forall x_0 \forall x_1 (np(c_i, c_j) \multimap s(x_0, x_1)) \multimap s(x_0, x_1) \\ l(\text{everyone}, c_i, c_j) &= \forall x_0 \forall x_1 (np(c_i, c_j) \multimap s(x_0, x_1)) \multimap s(x_0, x_1) \end{aligned}$$

which, in the spirit of Montague's 'quantifying in' rule, lets a quantifier take as its input a sentence s (regardless of the position labeling) which is incomplete for a noun phrase np at the position of the quantifier, where the output will be a sentence spanning the same positions as the incomplete sentence.

This single lexical assignment allows us to derive the two readings for ‘Gödel believes someone sleeps’. The *de dicto* reading, where the succedent s is linked to the antecedent s of *believes*, as shown in Figure 5.2, and the *de re* reading, where the succedent s is linked to the antecedent s of *someone*, as shown in Figure 5.3.

5.2.2 Relative Pronouns

In the Lambek calculus we can give a relative pronoun like ‘which’ the following two assignments

$$\begin{aligned} l'(which) &= (n \setminus n) / (np \setminus s) \\ &= (n \setminus n) / (s / np) \end{aligned}$$

This allows us to derive all cases where the incomplete sentence, which the relative pronoun expects to its right, is incomplete for an np at the right or left peripheral position, like the following

(5.6) Article which [[] $_{np}$ influenced Gödel] $_s$.

(5.7) Article which [Lambek likes [] $_{np}$] $_s$.

but not the cases where the np occurs elsewhere, as in the following examples

(5.8) Article which [Lambek wrote [] $_{np}$ yesterday] $_s$.

(5.9) Article which [Gödel sent [] $_{np}$ Russell] $_s$.

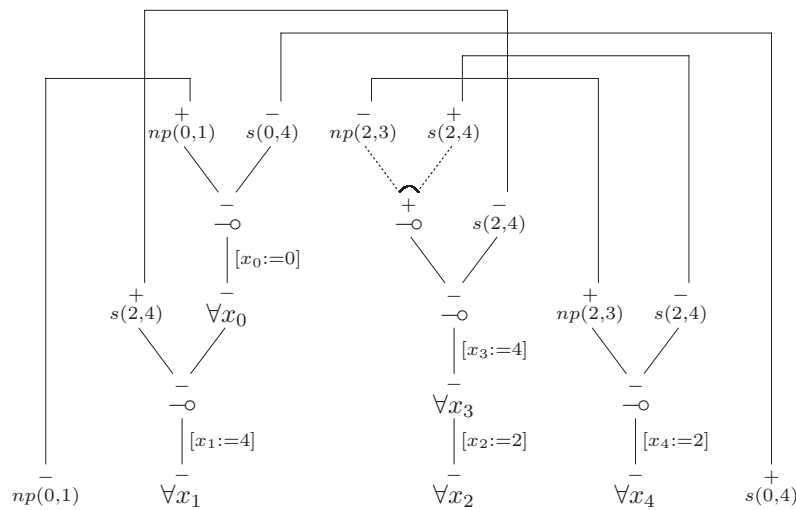


Figure 5.2: *De dicto* reading for ‘Gödel believes someone sleeps’

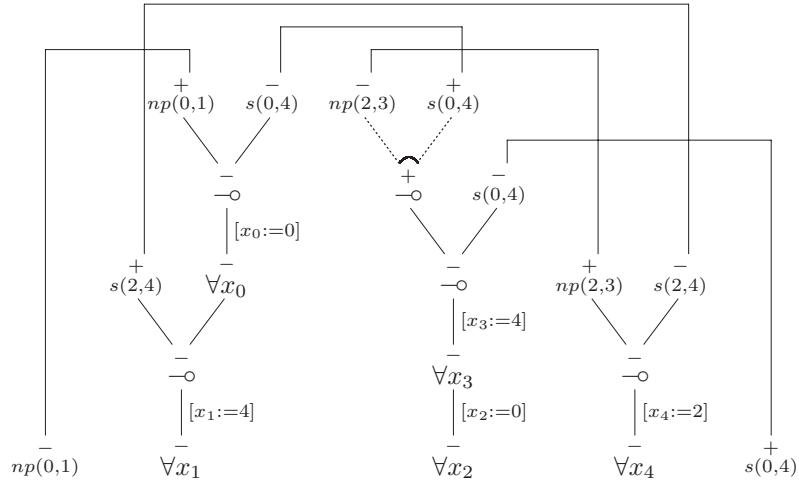
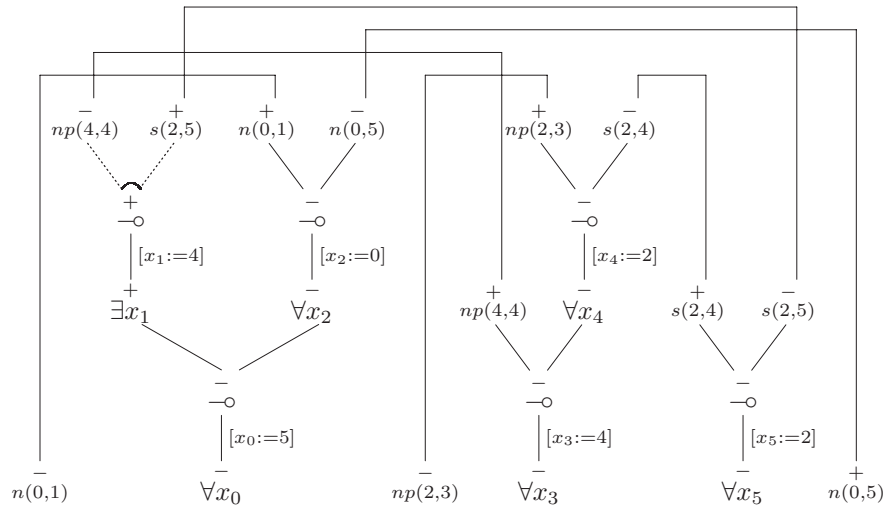
Figure 5.3: *De re* reading for ‘Gödel believes someone sleeps’

Figure 5.4: Proof net for ‘article which Lambek wrote yesterday’

In **MILL1** we can formulate the fact that the np may occur anywhere in the incomplete sentence by quantifying over its position arguments and using the same variable for both its left and right position, giving it a license to insert itself at any position. The resulting lexical entry

$$l(\text{which}, c_i, c_j) = \forall x_0. (\exists x_1. np(x_1, x_1) \multimap s(c_j, x_0)) \multimap (\forall x_2. n(x_2, c_i) \multimap n(x_2, x_0))$$

allows us to construct a proof net (Figure 5.4) for ‘article which Lambek wrote yesterday’ as an expression of category n .

Our treatment can be compared to the one proposed by Hodas (1992) for the linear logic programming language Lolli and it appears to suffer from the same problem observed by Hodas, namely that of blocking cases of extraction of an np from a complex subject, as in

(5.10) *Article which [[the proof in [] np] np is long] s .

without blocking subject extraction altogether. If we choose to make a subject its own domain, as we could using our approach to locality from Section 5.2.3, we would block the derivation of Example 5.6 in addition to the derivation of 5.10. If not, we fail to reject Example 5.10.

Pied Piping

Morrill (1994) discusses relatives like ‘whom’ and ‘which’ which allow pied piping, i.e. they have the possibility of fronting material from the extraction site. This is perhaps best illustrated by comparing the extraction case in Example 5.11 below to the nominal and prepositional pied piping cases in Examples 5.12 and 5.13 respectively. Morrill analyses these relatives as selecting for a pp or an np which is incomplete for an np , as indicated by the bracketing below.

(5.11) (a student) [[] np] np which [Gödel liked the article of [] np] s .

(5.12) (a student) [the article of [] np] np which [Gödel liked [] np] s .

(5.13) (a student) [of [] np] pp which [Gödel liked the article [] pp] s .

The assignment to ‘which’ of the previous section allows us to derive only the extraction case of Example 5.11, where no material is fronted. With the following two lexical assignments we can derive both the nominal pied piping case of Example 5.12 and the prepositional pied piping case of Example 5.13.

$$\begin{aligned} l(\text{which}, c_i, c_j) &= \forall x_1. \forall x_3. (\exists x_0. np(x_0, x_0) \multimap np(x_1, c_i)) \multimap \\ &\quad ((\exists x_2. np(x_2, x_2) \multimap s(c_j, x_3)) \multimap \\ &\quad \forall x_4. (n(x_4, x_1) \multimap n(x_4, x_3))) \\ &= \forall x_1. \forall x_3. (\exists x_0. np(x_0, x_0) \multimap pp(x_1, c_i)) \multimap \\ &\quad ((\exists x_2. pp(x_2, x_2) \multimap s(c_j, x_3)) \multimap \\ &\quad \forall x_4. (n(x_4, x_1) \multimap n(x_4, x_3))) \end{aligned}$$

If we allow for empty antecedent derivations, the extraction assignment is a special case of the nominal pied piping assignment. This is because there is an empty antecedent derivation of

$$\vdash \forall x_1. (\exists x_0. np(x_0, x_0) \multimap np(x_1, c_j))$$

which instantiates both x_1 and x_0 to c_j , after which we have a syntactic variant of the previous lexical assignment to ‘which’.

5.2.3 Locality

The Lambek calculus also has problems of overgeneration, caused by the associativity rule. As already suggested by our embedding translation for non-associative calculi in Section 5.1.4, we can block associativity when necessary.

The basic idea is to add a ‘domain variable’ as first argument of all atomic formulas. For ordinary lexical items and the succedent formula we universally quantify over this variable, but some lexical items may introduce new domains.

$$\begin{aligned}
l(\mathit{Lambek}, c_i, c_j) &= \forall x_0. np(x_0, c_i, c_j) \\
l(\mathit{Gödel}, c_i, c_j) &= \forall x_0. np(x_0, c_i, c_j) \\
l(\mathit{admires}, c_i, c_j) &= \forall x_1. \forall x_2. np(x_2, c_j, x_1) \multimap \\
&\quad \forall x_0. (np(x_2, x_0, c_i) \multimap s(x_2, x_0, x_1)) \\
l(\mathit{himself}, c_i, c_j) &= \forall x_0. \forall x_1. \forall x_2. \forall x_3. \forall x_4. (np(x_0, c_i, c_j) \multimap \\
&\quad (np(x_0, x_1, x_2) \multimap s(x_0, x_3, x_4))) \multimap \\
&\quad (np(x_0, x_1, x_2) \multimap s(x_0, x_3, x_4)) \\
l(\mathit{believes}, c_i, c_j) &= \forall x_1. \forall x_2. (\forall x_3. s(x_3, c_j, x_1) \multimap \\
&\quad \forall x_0. (np(x_2, x_0, c_i) \multimap s(x_2, x_0, x_1)))
\end{aligned}$$

The key lexical item here is *believes*: it introduces a new locality domain x_3 for the embedded s , which because of our conditions on quantifiers will be different from the domain assigned to the whole sentence.

The assignment to the reflexive *himself* indicates it has a transitive verb as its input and an intransitive verb as its output, but, because there is only a single locality variable x_0 , both will be in the same locality domain.

This allows us to account for

$$(5.14) \text{ Lambek}_{x_6} \text{ believes Gödel}_{x_1} \text{ admires himself}_{x_1}.$$

$$(5.15) * \text{Lambek}_{x_6} \text{ believes Gödel}_{x_1} \text{ admires himself}_{x_6}.$$

In Figure 5.5 on the facing page we show how, abstracting over the position labeling for the sake of simplicity, we can derive sentence 5.14.

Figure 5.6 on the next page shows that it is not possible to bind *himself* in the domain of the main sentence x_6 . When we choose the substitution $x_5 := x_6$, we can only link the positive $s(x_1)$ as shown, which will force us to substitute x_1 for x_4 . But after performing this link there are three negative $np(x_6)$ formulas and only two positive $np(x_6)$, so continuing will never result in a proof net.

5.3 Conclusions

We have shown that we can extend the multiplicative fragment of linear logic to the first order fragment without sacrificing either the elegant proof theory (proof nets with a natural soundness criterion) or the computational complexity.

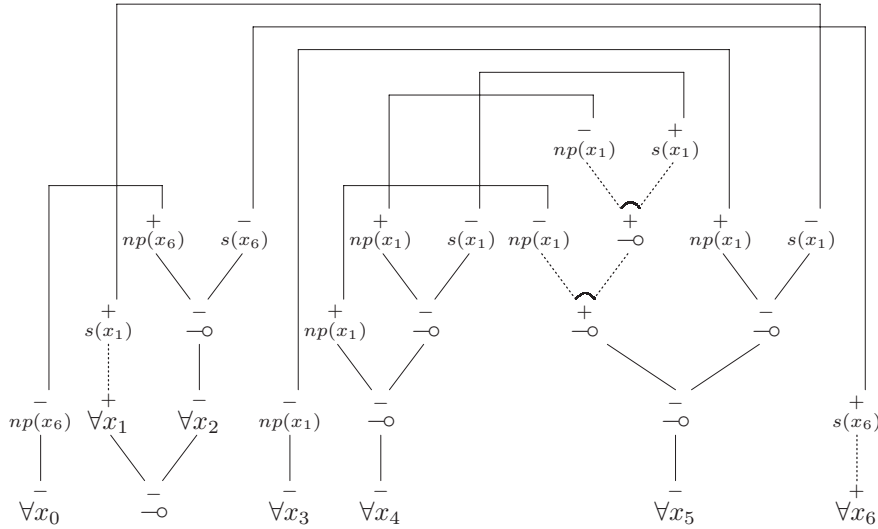


Figure 5.5: ‘Lambek_{x₆} believes Gödel_{x₁} admires himself_{x₁}’

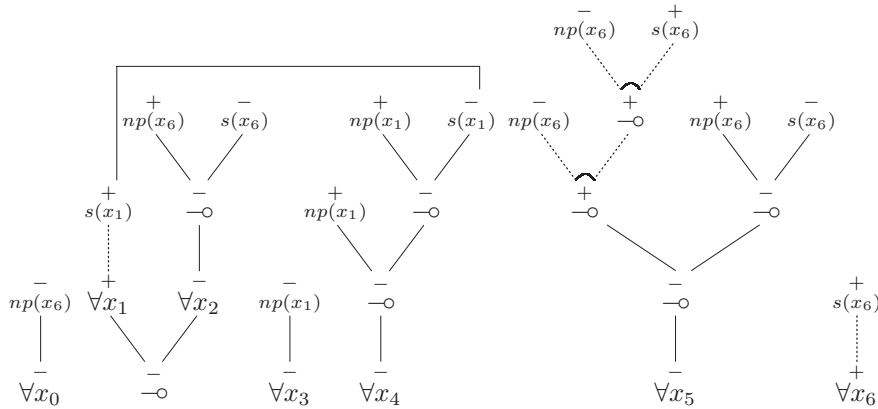


Figure 5.6: ‘*Lambek_{x₆} believes Gödel_{x₁} admires himself_{x₆}’

We have given an embedding translation of sequents of both **L** and **NL** to sequents in the first order fragment and shown how several linguistic phenomena which could not be adequately handled in the Lambek calculus could be treated in **MILL1**.

Though we have not tried to be comprehensive in our discussion of linguistics, we hope to have given the reader an idea of a number of possible applications of **MILL1**.

CHAPTER 6

ALGEBRAIC CRITERIA

THE proof net criteria we've seen so far, for the multiplicative fragment and the first order multiplicative fragment of linear logic, have been of a graph theoretic nature; consisting of either switching or contractions on the graph representation of proof structures. In this chapter we will look at a different approach to checking the correctness of proof structures, which are called the algebraic or labeling approaches. In an algebraic correctness criterion, the nodes of the graph are labeled with terms of an algebra and equations or rewrite rules on these terms will decide whether the proof structure is correct.

The algebraic criteria discussed in this chapter are also used in the Grail automated theorem prover, which is described in Appendix A.

Though the proof net approach described in Chapter 4 gives us both a nice proof theory, and a simple, transparent algorithm for proof search, it does so only for **LP**. While **LP** is the language of choice for *semantic* composition, via the Curry Howard interpretation of proofs, as a language of *structural* composition it is only of limited interest. We will develop a solution to this problem in the line of Gabbay's (1996) labeled deduction; instead of using formulas A as our basic declarative unit, we use *labeled* formulas $x : A$. The label x represents a piece of structural information. The rules will be adapted to operate on both the formulas and the labels.

The agenda for this chapter is the following.

- Keep the proof theory of multiplicative intuitionistic linear logic.
- Introduce a new level of description in the form of structural labeling.
- Apply these labels as *constraints* which give us the required discriminatory power to embed arbitrary multimodal Lambek calculi with linear structural rules in the proof net architecture.

Identity

$$\frac{}{\mathbf{x} : A \vdash \mathbf{x} : A} [Ax] \quad \frac{\Gamma, \mathbf{y} : B \vdash Z[\mathbf{y}] : C \quad \Delta \vdash Y : B}{\Gamma, \Delta \vdash Z[Y] : C} [Cut]$$

Binary Connectives

$$\frac{\Gamma, \mathbf{x} : A, \mathbf{y} : B \vdash Z[\mathbf{x} \circ_i \mathbf{y}] : C}{\Gamma, \mathbf{z} : A \bullet_i B \vdash Z[\mathbf{z}] : C} [L\bullet_i] \quad \frac{\Gamma \vdash X : A \quad \Delta \vdash Y : B}{\Gamma, \Delta \vdash X \circ_i Y : A \bullet_i B} [R\bullet_i]$$

$$\frac{\Delta \vdash Y : B \quad \Gamma, \mathbf{x} : A \vdash Z[\mathbf{x}] : C}{\Gamma, \Delta, \mathbf{y} : A/_i B \vdash Z[\mathbf{y} \circ_i Y] : C} [L/_i] \quad \frac{\Gamma, \mathbf{y} : B \vdash X \circ_i \mathbf{y} : A}{\Gamma \vdash X : A/_i B} [R/_i]$$

$$\frac{\Delta \vdash Y : B \quad \Gamma, \mathbf{x} : A \vdash Z[\mathbf{x}] : C}{\Gamma, \Delta, \mathbf{y} : B \setminus_i A \vdash Z[Y \circ_i \mathbf{y}] : C} [L \setminus_i] \quad \frac{\Gamma, \mathbf{y} : B \vdash \mathbf{y} \circ_i X : A}{\Gamma \vdash X : B \setminus_i A} [R \setminus_i]$$

Unary Connectives

$$\frac{\Gamma, \mathbf{x} : A \vdash Z[\langle \mathbf{x} \rangle^i] : C}{\Gamma, \mathbf{y} : \diamond_i A \vdash Z[\mathbf{y}] : C} [L\diamond_i] \quad \frac{\Gamma \vdash Z : C}{\Gamma \vdash \langle Z \rangle^i : \diamond_i C} [R\diamond_i]$$

$$\frac{\Gamma, \mathbf{x} : A \vdash Z[\mathbf{x}] : C}{\Gamma, \mathbf{y} : \square_i A \vdash Z[\langle \mathbf{y} \rangle^i] : C} [L\square_i^\perp] \quad \frac{\Gamma \vdash \langle Z \rangle^i : C}{\Gamma \vdash Z : \square_i C} [R\square_i^\perp]$$

Structural Rules

$$\frac{\Gamma \vdash Z[\exists'[X_1, \dots, X_n]] : C}{\Gamma \vdash Z[\exists[X_{\pi_1}, \dots, X_{\pi_n}]] : C} [SR]$$

Table 6.1: The labeled sequent calculus $\mathbf{NL}\diamond_{\mathcal{R}}$

We will first follow this approach for the labeled sequent calculus and labeled natural deduction, where adding labeling and proving soundness and completeness is a relatively simple step.

When we add labels to the proof net calculus we will need some auxiliary label constructors in order to allow us to make the same distinctions as the labeled sequent calculus. In the next chapter we will see how these labels correspond to two sided proof nets and prove soundness and completeness of that approach.

6.1 Labeled Sequent Calculus

A labeled deductive version of the sequent calculus is presented in Table 6.1. This consists of replacing the antecedent by a multiset, and formulas by labeled formulas. Labels are defined as follows.

Definition 6.1 *Over a countably infinite set $\mathbf{x}, \mathbf{y}, \mathbf{z}, \dots$ of structural variables \mathcal{V} ,*

Hypothesis

$$\mathbf{x} : A$$

Binary Connectives

$$\frac{[x : A]^n \quad [y : B]^n \quad \vdots}{\frac{X : A \bullet_i B \quad Z[x \circ_i y] : C}{Z[X] : C} [\bullet E]^n \quad \frac{X : A \quad Y : B}{X \circ_i Y : A \bullet_i B} [\bullet I]^n} [\bullet E]^n$$

$$\frac{X : A /_i B \quad Y : B}{X \circ_i Y : A} [/E] \quad \frac{[x : B]^n \quad \vdots}{\frac{X \circ_i x : A}{X : A /_i B} [/I]^n} [/E]$$

$$\frac{Y : B \quad X : B \setminus_i A}{Y \circ_i X : A} [\setminus E] \quad \frac{[x : B]^n \quad \vdots}{\frac{x \circ_i X : A}{X : B \setminus_i A} [\setminus I]^n} [\setminus E]$$

Unary Connectives

$$\frac{[x : A]^n \quad \vdots}{\frac{X : \diamond_i A \quad Z[\langle x \rangle^i] : C}{Z[X] : C} [\diamond E]^n \quad \frac{X : A}{\langle X \rangle^i : \diamond_i A} [\diamond I]^n} [\diamond E]^n$$

$$\frac{X : \square_i^\perp A}{\langle X \rangle^i : X} [\square^\perp E] \quad \frac{\langle X \rangle^i : A}{X : \square_i^\perp A} [\square^\perp I]$$

Structural Rules

$$\frac{\Gamma \vdash Z[\Xi'[X_1, \dots, X_n]] : C}{\Gamma \vdash Z[\Xi[X_{\pi_1}, \dots, X_{\pi_n}]] : C} [SR]$$

Table 6.2: The labeled natural deduction calculus $\mathbf{NL} \diamond_{\mathcal{R}}$

natural deduction calculus for $\mathbf{NL} \diamond_{\mathcal{R}}$. In this calculus we can suppress the antecedent formulas which are still explicitly present in the unlabeled calculus of Table 3.9 on page 43, because the label provides enough information to fully reconstruct the antecedent.

The following translation function, which is a simple modification of the one in the previous section, shows how we can generate the antecedent for-

$$\begin{array}{c}
\frac{\frac{\text{wil} : \Box_0^\perp((np \setminus_1 s) /_0 \text{inf})}{\langle \text{wil} \rangle^0 : (np \setminus_1 s) /_0 \text{inf}} \quad [\Box_0^\perp E] \quad \frac{\text{redde}n : \Box_0^\perp np \setminus_1 \text{inf}}{\langle \text{redde}n \rangle^0 : np \setminus_1 \text{inf}} \quad [\Box_0^\perp E]}{\frac{\text{n} : np \quad \langle \text{redde}n \rangle^0 : np \setminus_1 \text{inf}}{\text{n} \circ_1 \langle \text{redde}n \rangle^0 : \text{inf}} \quad [/_0 E]} \quad [\setminus_1 E]} \\
\frac{\text{r} : np \quad \langle \text{wil} \rangle^0 \circ_0 (\text{n} \circ_1 \langle \text{redde}n \rangle^0) : np \setminus_1 s}{\text{r} \circ_1 (\langle \text{wil} \rangle^0 \circ_0 (\text{n} \circ_1 \langle \text{redde}n \rangle^0)) : s} \quad [\setminus_1 E]}{\text{r} \circ_1 (\text{n} \circ_1 (\langle \text{wil} \rangle^0 \circ_0 \langle \text{redde}n \rangle^0)) : s} \quad [MC]} \\
\frac{\text{r} \circ_1 (\text{n} \circ_1 (\langle \text{wil} \rangle^0 \circ_0 \langle \text{redde}n \rangle^0)) : s}{\text{r} \circ_1 (\text{n} \circ_1 (\text{wil} \circ_1 \text{redde}n))^0 : s} \quad [K]}{\text{r} \circ_1 (\text{n} \circ_1 (\text{wil} \circ_1 \text{redde}n))^1 : s} \quad [I]} \\
\frac{\text{r} \circ_1 (\text{n} \circ_1 (\text{wil} \circ_1 \text{redde}n))^1 : s}{\text{r} \circ_1 (\text{n} \circ_1 (\text{wil} \circ_1 \text{redde}n))^1 : s} \quad [K2]}{\text{r} \circ_1 (\text{n} \circ_1 (\text{wil} \circ_1 \text{redde}n))^1 : s} \quad [K2]} \\
\frac{\text{r} \circ_1 (\text{n} \circ_1 (\text{wil} \circ_1 \text{redde}n))^1 : s}{\text{r} \circ_1 (\text{n} \circ_1 (\text{wil} \circ_1 \text{redde}n)) : \Box_1^\perp s} \quad [\Box_1^\perp I]}{\text{r} \circ_1 (\text{n} \circ_1 (\text{wil} \circ_1 \text{redde}n)) : \Box_1^\perp s} \quad [\Box_1^\perp I]}
\end{array}$$

Figure 6.2: Natural deduction derivation of ‘(dat) Ripley Newt wil redde’n’

mulas from the label and vice versa.

$$\begin{aligned}
\|x\| &= A && \text{iff } x : A \text{ is a hypothesis of the proof} \\
\|\langle X \rangle^i\| &= \langle \|X\| \rangle^i \\
\|X \circ_i Y\| &= \|X\| \circ_i \|Y\|
\end{aligned}$$

Proposition 6.4 *If $\|X\| = \Gamma$ then $X : C$ has a labeled natural deduction proof iff $\Gamma \vdash C$.*

Proof Induction. □

Because we can suppress the antecedent completely without losing information and because we don’t have to introduce a new structural variables for every subformula used in the proof as in the labeled sequent calculus, the labeled natural deduction calculus gives an especially pleasant way of portraying proofs in $\mathbf{NL} \diamond_{\mathcal{R}}$.

Example 6.5 *Figure 6.2 presents a labeled natural deduction version of the derivation shown in Figure 3.4 on page 42.*

6.3 Labeled Proof Nets

Moortgat (1997) proposes to add structural labeling to proof nets in the following way. His proposal should be contrasted with earlier labeling proposals like Moortgat (1990) and Morrill (1995) which were incomplete for the product formulas.

Definition 6.6 *Over a countably infinite set \mathcal{V} of structural variables, we define the set of structural labels \mathcal{L} as follows*

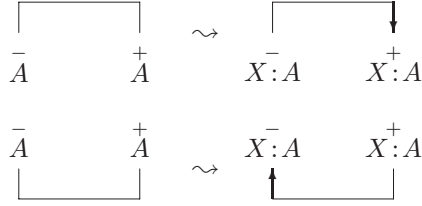


Table 6.3: Dynamic graphs for identity links

$$\begin{array}{l}
 \mathcal{L} ::= \mathcal{V} \\
 | \langle \mathcal{L} \rangle^i \quad (\text{Structural counterpart of } \diamond_i) \\
 | \mathcal{L} \circ_i \mathcal{L} \quad (\text{Structural counterpart of } \bullet_i) \\
 | [\mathcal{L}]^i \quad (\text{Auxiliary constructor for } \square_i^\perp) \\
 | \lfloor \mathcal{L} \rfloor^i \quad (\text{Auxiliary constructor for } \diamond_i) \\
 | \mathcal{V} \setminus_i \mathcal{L} \quad (\text{Auxiliary constructor for } \setminus_i) \\
 | \mathcal{L} /_i \mathcal{V} \quad (\text{Auxiliary constructor for } /_i) \\
 | \mathcal{L}^{\triangleleft_i} \quad (\text{Auxiliary constructor for } \bullet_i) \\
 | \mathcal{L}^{\triangleright_i} \quad (\text{Auxiliary constructor for } \bullet_i)
 \end{array}$$

Definition 6.7 The set of normal labels \mathcal{N} is a subset of \mathcal{L} defined as follows

$$\begin{array}{l}
 \mathcal{N} ::= \mathcal{V} \\
 | \langle \mathcal{N} \rangle^i \\
 | \mathcal{N} \circ_i \mathcal{N}
 \end{array}$$

As can be seen from the definition above, there are two kinds of label constructors: *structural* and *auxiliary*. The structural connectives are those we used to generate antecedent terms for multimodal sequents. In addition, we have an auxiliary constructor for each of the connectives in our formula language. The purpose of these constructors will be to check the sublinear constraints on derivability for that specific connective.

The negative formulas are initially assigned distinct structural variables x, y, \dots , the positive formula a metavariable Z . We apply the following links, where all newly occurring structural or metavariables are fresh, until we reach atomic formulas.

From a proof structure we can construct the underlying dynamic graph, or *essential net* as Lamarche (1994) calls them, as shown in Tables 6.3, 6.4 and 6.5 on this page and the following pages. A dynamic graph is a directed graph, where every link in the proof structure induces one or two edges. The direction of the edges in the graph indicate the flow of information in the labels: for all negative formulas the information flow is upwards, whereas for all positive formulas the information flows downwards. Note that we follow de Groote (1999a) in reversing the arrows of Lamarche (1994).

Definition 6.8 An dynamic graph is correct if the following properties hold.

- (i) it is acyclic,

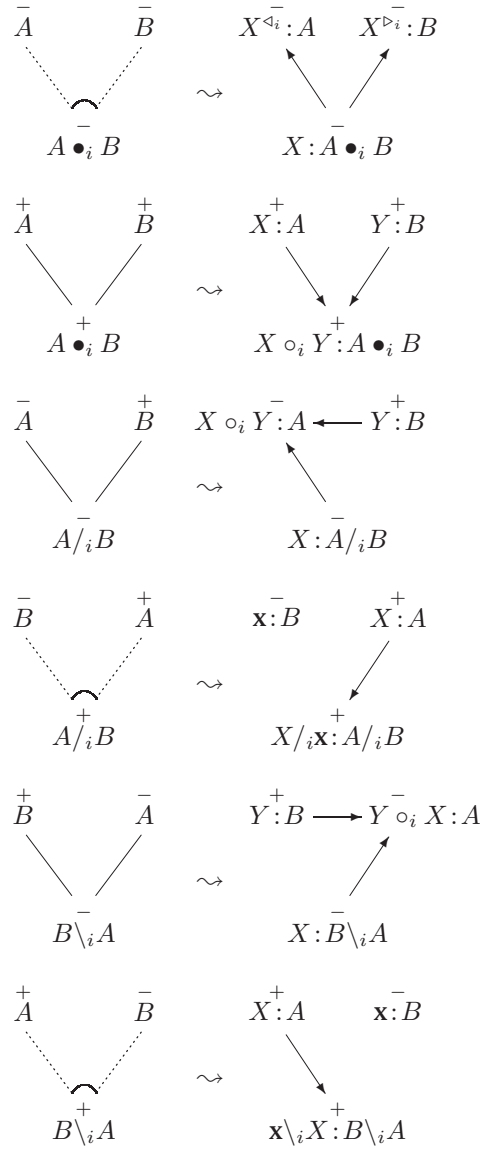


Table 6.4: Dynamic graphs for binary links

- (ii) every path from the negative premiss of a positive $/_i$ or \backslash_i link passes through the conclusion of this link,
- (iii) every path from the negative inputs of the graph passes through the positive output of the graph.

Theorem 6.9 (Lamarche (1994)) A sequent $\Gamma \vdash C$ is provable in MILL iff its dynamic graph is correct.

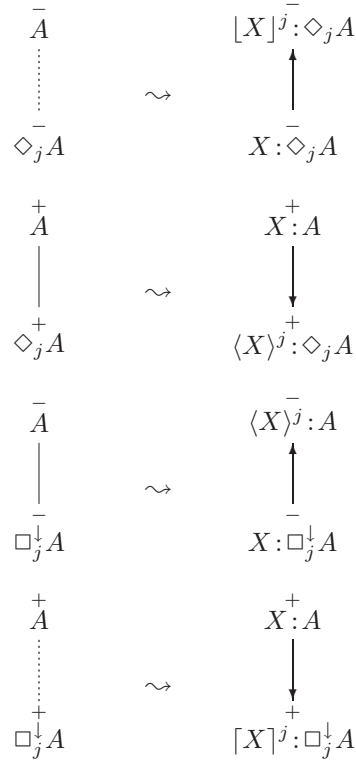


Table 6.5: Dynamic graphs for unary links

Corollary 6.10 *Every correct MILL proof net can be assigned a label to its (unique) positive conclusion based on its underlying dynamic graph.*

Proof Immediate from Theorem 6.9 and the fact that the labeling follows the information flow of the dynamic graph. \square

We now define a set of *conversions* on this succedent label, which will check the sublinear constraints on derivability. We have one logical conversion for each connective, as shown in Table 6.6.

We call the label on the right hand side of the conversion a *redex* and the label on the left hand side its *contractum*. Converting a label X to a label Y ($X \rightarrow Y$) consists of replacing all occurrences of a redex by its contractum. We will write \rightarrow (reduces to) for the transitive, reflexive closure of \rightarrow .

In addition to these logical label conversions, we can have structural label conversions corresponding to the set of structural rules \mathcal{R} . We again require these conversions to be *linear* according to Definition 3.4, i.e. all metavariables in the conversion occur exactly once on the left hand side of the conversion and exactly once on the right hand side of the conversion. Because our logical component is fixed and language independent a label conversion cannot refer to auxiliary constructors as this would allow us to change the

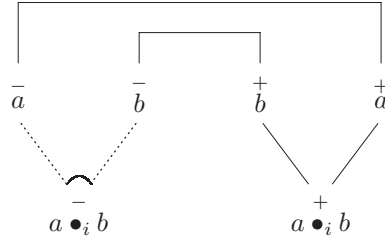
| | |
|--|---------------------|
| $X^{\triangleleft_i} \circ_i X^{\triangleright_i} \rightarrow X$ | $[\bullet_i]$ |
| $(X \circ_i Y) /_i Y \rightarrow X$ | $[/_i]$ |
| $Y \setminus_i (Y \circ_i X) \rightarrow X$ | $[\setminus_i]$ |
| $\langle [X]^i \rangle^i \rightarrow X$ | $[\diamond_i]$ |
| $\lceil [X]^i \rceil^i \rightarrow X$ | $[\square_i^\perp]$ |

Table 6.6: Logical conversions

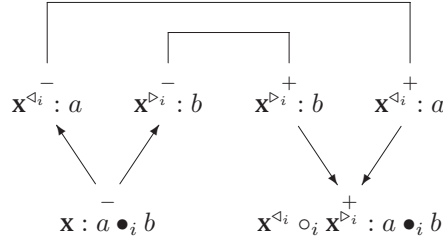
basic meaning of the connectives.

Definition 6.11 We will call a label \mathcal{L} reducible if and only if there is a normal label \mathcal{N} such that $\mathcal{L} \rightarrow \mathcal{N}$.

Example 6.12 A standard well-behavedness property for a sequent or proof net calculus is that we can restrict ourselves to atomic instances of the axiom rule. We have seen in Lemma 4.10 of Section 4.4 that we can derive all non atomic instances of this rule at the logical level. The label reductions should therefore also allow us to produce a normal label for those. For the product formula, the proof net below



produces the following dynamic graph.



Using a non atomic axiom link would produce a label x for the succedent type immediately. We can now use the logical conversion for the product formula on the output label $x^{\triangleleft_i} \circ_i x^{\triangleright_i}$ to get $x^{\triangleleft_i} \circ_i x^{\triangleright_i} \rightarrow x$.

Definition 6.13 A labeled proof structure is a proof net if and only if it confirms to the following conditions

- (i) its underlying dynamic graph is correct.
- (ii) all negative conclusions of the proof structure are assigned a structural variable.
- (iii) the label assigned to the positive conclusion of the proof structure is reducible.

6.4 Conclusion

We have seen how a two level approach of proof nets and labeling. What is still lacking is a proof of the soundness and correctness of this algebraic criterion. In the next chapter, I will present an alternative, graph theoretic approach to proof nets for the multimodal Lambek calculus, which turns out to be closely related to the algebraic approach. That system will have only a single level of representation for which we will prove the basic soundness, completeness and cut elimination results.

CHAPTER 7

CONTRACTION CRITERIA

IN the previous chapter, we have seen how we could use algebraic correctness criteria for $NL \diamond_{\mathcal{R}}$. In this chapter, I will present a graph theoretic correctness criterion, in the line of Danos' contraction criterion for Multiplicative Linear Logic, discussed in Section 4.5.

This chapter is based on joint research with Quintijn Puite, and parts of it have appeared before in (Puite & Moot 1999), (Moot & Puite 1999) and (Moot & Puite 2001).

7.1 Two Sided Proof Nets

Proof nets, as we have seen them in the previous chapters, having their roots in the one sided sequent calculus, have a certain asymmetry in that they are allowed to have conclusions, i.e. formulas which are not the premiss of a link, but not hypotheses, i.e. formulas which are not the conclusion of a link. Puite (1998) proposes a proof net calculus based on the two sided sequent calculus, where a proof net can have both hypotheses and conclusions.

In this calculus every link has a dual link. So, in addition to a normal right tensor link, we have a left tensor link as shown in Table 7.1 on the next page, which is by symmetry a par link and which has $A \otimes B$ as a premiss and A and B as conclusions.

In the two sided calculus, this new link is a primitive link, but we can see it as a *defined* link in the one sided calculus as shown in Figure 7.1 on the following page.

If we compare the $[L \otimes]$ link with the defined link above, it is easy to verify that for any proof structure \mathcal{S} with the primitive $[L \otimes]$ link we can generate a proof structure \mathcal{S}' where the primitive link is replaced by the defined link

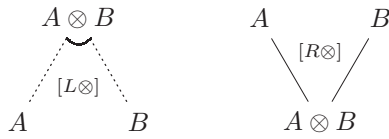


Table 7.1: Two sided tensor links

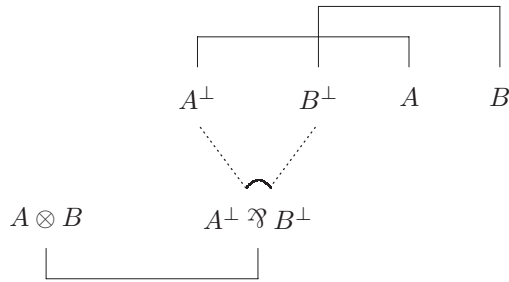


Figure 7.1: One sided representation of the $[L\otimes]$ link

above. Now for every switching ω the correction graph ωS will be acyclic and connected if and only if $\omega S'$ is acyclic and connected.

We also have a left and right par link, which look as shown in Table 7.2.

One of the advantages of a two sided calculus is that we can now have explicit negation links as shown in Table 7.3 on the facing page, a left negation link, which looks very much like an axiom link, and a right negation link, which looks very much like a cut link.

The elimination of double negation and the De Morgan equalities, which were ‘compiled away’ in the one sided calculus, now become explicitly representable in the two sided calculus.

Example 7.1 *As an example, we construct the proof net for the De Morgan equivalence below.*

$$a \otimes b \vdash (a^\perp \wp b^\perp)^\perp$$

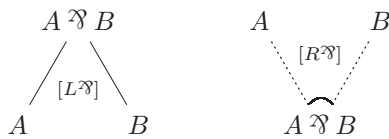


Table 7.2: Two sided par links

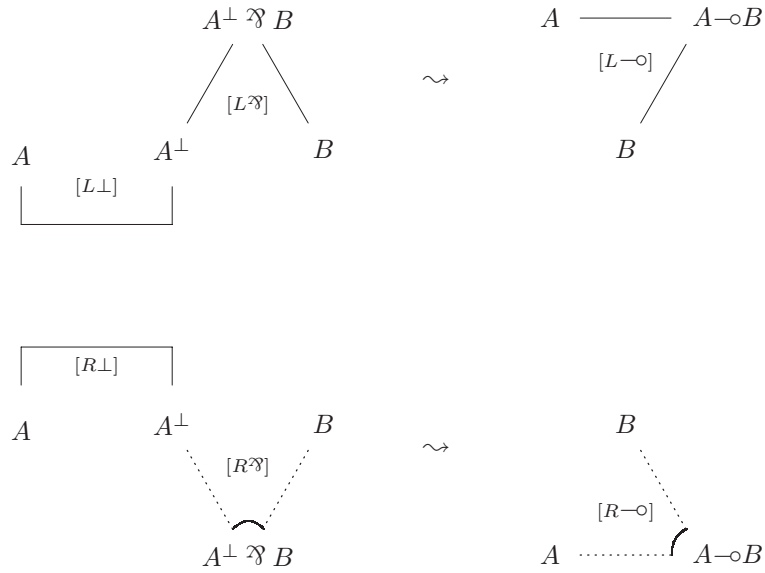


Table 7.4: Links for implication, defined and primitive

Now, given that linear implication $A \multimap B$ is defined as $A^\perp \wp B$, we can abbreviate a combined par and negation link as shown in Table 7.4.

Example 7.2 *The sequent below*

$$a \multimap b, b \multimap c \vdash a \multimap c$$

generates the proof structure shown in Figure 7.4 on the next page after connecting the b and c atomic formulas.

Note that we now want to connect the a atomic formulas, but it is difficult to portray this on a flat plane. When we imagine the plane we draw on is cylindrical, i.e. if we move up far enough we reenter the plane from below, we can see how moving upward from the top a formula we would reach the bottom a formula. However, we will choose to portray these situations by using curved connections, as shown in Figure 7.5 on the facing page.

Observe that the above representation is very close to the corresponding natural deduction proof, shown in Figure 7.6 on the next page, with the difference that instead of coindexing the $[R\multimap]$ rule with the discharged hypothesis a , we represent the discharge by a link which removes the hypothesis a from the hypotheses of the proof structure.

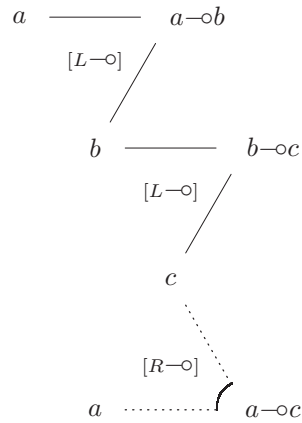


Figure 7.4: Proof structure for $a, a \multimap b, b \multimap c \vdash a, a \multimap c$

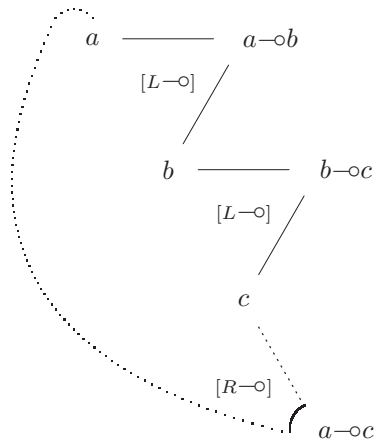


Figure 7.5: Proof net for $a \multimap b, b \multimap c \vdash a \multimap c$

7.2 Sequent Calculus

Though we have seen in the previous section how to extend proof nets for multiplicative linear logic to a two sided calculus, unless we modify our cor-

$$\frac{\frac{[a \vdash a]^1 \quad \frac{b \vdash b \quad c \vdash c}{b, b \multimap c \vdash c} [L \multimap]}{a, a \multimap b, b \multimap c \vdash c} [L \multimap]}{a \multimap b, b \multimap c \vdash a \multimap c} [R \multimap]^1$$

Figure 7.6: Natural deduction proof of $a \multimap b, b \multimap c \vdash a \multimap c$

rectness criterion, we will still be stuck in the multiplicative fragment of linear logic, albeit in the two sided formulation of it.

The rest of this chapter will be devoted to giving a correctness criterion for $\mathbf{NL}\diamond_{\mathcal{R}}$ by stating a contraction criterion which is a special case of Danos' contraction criterion as discussed in Section 4.5. Moreover, this criterion will be modular in that it generates a correctness criterion from the structural rule component \mathcal{R} of specific grammars, as long as they are linear according to Definition 3.4. In Section 7.8, we will give a specialized contraction criterion for the original Lambek calculus \mathbf{L} .

For ease of reference, we repeat the language and the sequent calculus of $\mathbf{NL}\diamond_{\mathcal{R}}$ here.

Definition 7.3 *The language $\mathcal{L}(\mathbf{NL}\diamond_{\mathcal{R}})$ consists of the following.*

[Formulas] *Given a set of atomic formulas \mathcal{A} , a set I of binary indices and a set J of unary indices, the set of formulas is defined inductively as follows*

$$\mathcal{F} ::= \mathcal{A} \mid \diamond_j \mathcal{F} \mid \square_j^\perp \mathcal{F} \mid \mathcal{F}/_i \mathcal{F} \mid \mathcal{F} \bullet_i \mathcal{F} \mid \mathcal{F} \setminus_i \mathcal{F}$$

We will use A, B, C, \dots to denote arbitrary formulas.

[Antecedent Terms] *Over the set of formulas \mathcal{F} , we define the set of antecedent terms \mathcal{T} as follows*

$$\mathcal{T} ::= \mathcal{F} \mid \langle \mathcal{T} \rangle^j \mid \mathcal{T} \circ_i \mathcal{T}$$

We will use $\Gamma, \Delta, \Xi, \dots$ to denote arbitrary antecedent terms. When we want to refer to a specific subtree occurrence Δ of an antecedent term Γ we will write this as $\Gamma[\Delta]$

[Sequents] *A sequent is written as $\Gamma \vdash C$, where Γ is an antecedent term we will call the antecedent of the sequent and C is a formula we will call the succedent of the sequent.*

The sequent calculus for $\mathbf{NL}\diamond$ is given by the rules in Table 7.5 on the facing page. The set of structural rules \mathcal{R} is an additional set of sequent rules, which is dependent on the application of the system. Each rule of \mathcal{R} is schematically of the form shown on the last line of Figure 7.5 on the next page, where Ξ and Ξ' are fixed trees built from the structural operators $(- \circ_i -)$ and $\langle - \rangle^j$ with n distinct structural variables as leaves, and where π is a permutation of these leaves. As a consequence, each structural variable occurs once in the premiss and once in the conclusion of a structural rule.

This restriction guarantees the structural rules of contraction and weakening will never be derivable in our logic and as a consequence all our connectives are multiplicatives in the sense of Danos & Regnier (1989).

Illustration: wh extraction in English

As our running example throughout this chapter we will look at what is often called *wh* extraction.

$$\begin{array}{c}
\frac{}{A \vdash A} [Ax] \qquad \frac{\Gamma[A] \vdash C \quad \Delta \vdash A}{\Gamma[\Delta] \vdash C} [Cut] \\
\\
\frac{\Gamma[\langle A \rangle^j] \vdash C}{\Gamma[\diamond_j A] \vdash C} [L\diamond_j] \qquad \frac{\Gamma \vdash C}{\langle \Gamma \rangle^j \vdash \diamond_j C} [R\diamond_j] \\
\\
\frac{\Gamma[A] \vdash C}{\Gamma[\langle \square_j^\perp A \rangle^j] \vdash C} [L\square_j^\perp] \qquad \frac{\langle \Gamma \rangle^j \vdash C}{\Gamma \vdash \square_j^\perp C} [R\square_j^\perp] \\
\\
\frac{\Gamma[(A \circ_i B)] \vdash C}{\Gamma[A \bullet_i B] \vdash C} [L\bullet_i] \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{(\Gamma \circ_i \Delta) \vdash A \bullet_i B} [R\bullet_i] \\
\\
\frac{\Gamma[A] \vdash C \quad \Delta \vdash B}{\Gamma[(A/_i B \circ_i \Delta)] \vdash C} [L/_i] \qquad \frac{(\Gamma \circ_i B) \vdash A}{\Gamma \vdash A/_i B} [R/_i] \\
\\
\frac{\Gamma[A] \vdash C \quad \Delta \vdash B}{\Gamma[(\Delta \circ_i B \setminus_i A)] \vdash C} [L\setminus_i] \qquad \frac{(B \circ_i \Gamma) \vdash A}{\Gamma \vdash B \setminus_i A} [R\setminus_i] \\
\\
\frac{\Gamma[\exists'[\Delta_1, \dots, \Delta_n]] \vdash C}{\Gamma[\exists[\Delta_{\pi_1}, \dots, \Delta_{\pi_n}]] \vdash C} [SR]
\end{array}$$

Table 7.5: The sequent calculus $NL_{\diamond\mathcal{R}}$

To keep the current discussion simple we will only look at the *wh* word ‘whom’, which we analyze as a noun modifier selecting a sentence from which a noun phrase is missing. The restriction ‘whom’ imposes is that this missing noun phrase cannot occur in subject position, as indicated by the following examples.

(7.1) * agent whom [[]_{np} interrogated Neo]_s

(7.2) agent whom [Trinity escaped []_{np}]_s

(7.3) agent whom [Morpheus considered []_{np} dangerous]_s

To model this behavior, we give a very simple grammar fragment with only one binary and one unary mode. An extracted *np* is marked as $\diamond_0 \square_0^\perp np$. As $\diamond_j \square_j^\perp A \vdash A$ is a theorem of the base logic for all j and A , this allows these constituents to function as an *np*. Crucial for this application is that the $[L\square_j^\perp]$ rule, read top down, introduces unary brackets, which produces the proper configuration for the structural rules shown in Table 7.6.

It should be noted, however, that these rules allow a $\langle \Delta \rangle^0$ constituent to move only from a right branch of a structure to another right branch. As a subject would appear on a left branch, this prevents subject extraction as desired.

7.3 Proof Structures

We will now present proof structures for $\mathbf{NL}\diamond_{\mathcal{R}}$.

Definition 7.5 A link L is a tuple $\langle \tau, \nu, P, Q, p, q \rangle$, where τ is either ' \otimes ' or ' \wp ', ν is a label indicating the name of the link, P is a sequence of formulas which we call the premisses of L , Q is a sequence of formulas which we call the conclusions of L , p is a subsequence of P and q is a subsequence of Q such that $\text{length}(p) + \text{length}(q) \leq 1$.

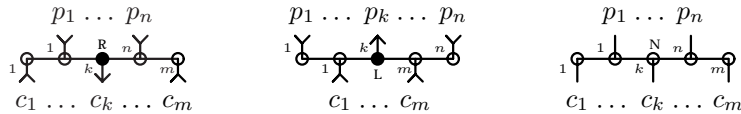
If $\tau = '\otimes'$, we will call the link a tensor link. If $\tau = '\wp'$, we will call the link a par link.

If $\text{length}(p) = 1$, we will call the link a left link, the element A of p the output or main formula and the other elements of P and the elements of Q the input or active formulas.

If $\text{length}(q) = 1$, we will call the link a right link, the element B of q the output or main formula and the elements of P and the other elements of Q the input or active formulas.

If $\text{length}(p) = \text{length}(q) = 0$ we will call the link a neutral link.

We will display our links a bit different from the way we did in Section 7.1 and portray them, following Puite (1998), as shown below, with the premisses above the horizontal line and the conclusions below it, both numbered according to their linear order (when no confusion is possible we will often omit the numbering and order the premisses and conclusions from left to right). For left and right links, we indicate the main formula of the link by an arrow moving *to* that formula, while the active formulas have an arrow moving *from* them. Neutral links are displayed without arrows. We attach the label ν to the horizontal line.



We distinguish tensor links and par links graphically by drawing a solid horizontal line for a tensor link and a dashed horizontal line for a par link.

Definition 7.6 A proof structure $\langle S, \mathcal{L} \rangle$ consists of a finite set S of formulas together with a set \mathcal{L} of links in S of the forms shown in Table 7.8, for each binary mode i and unary mode j .

such that the following holds.

- every formula of S is at most once a conclusion of a link,
- every formula of S is at most once a premiss of a link.

Formulas which are not the conclusion of any link are the *hypotheses* H of the proof structure, while those that are not the premiss of any link are its *conclusions* Q .

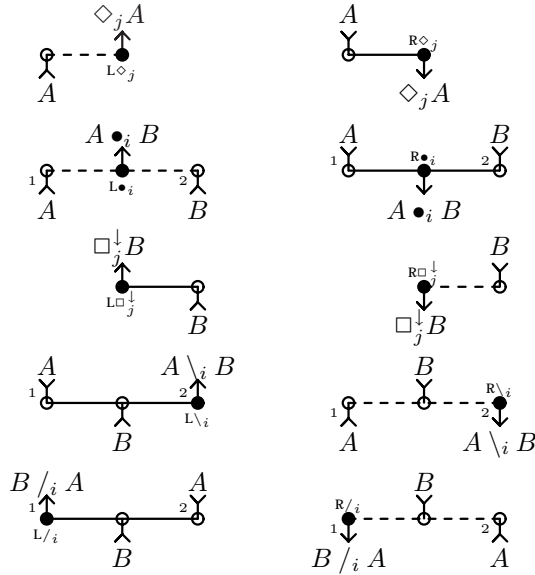


Table 7.8: Links for $NL\Diamond\mathcal{R}$

Note that there are no links corresponding to the axiom or cut rule in the sequent calculus. Instead we will have axiomatic and cut *formulas*. An axiomatic formula is a formula which is not the main formula of any link, whereas a cut formula is a formula which is the main formula of two links. We will call all formulas which are neither cut nor axiomatic formulas flow formulas.

We can determine which formulas are axiomatic and cut formulas in the graphical representation of a proof structure as follows. An axiomatic formula has no arrows pointing to it; depending on whether it is a hypothesis or a conclusion of the proof structure, there are four possibilities. A cut formula has two arrows pointing to it.

| | | | | |
|------------|---|---|---|---|
| A | $\begin{matrix} A \\ \Upsilon \end{matrix}$ | $\begin{matrix} \downarrow \\ A \end{matrix}$ | $\begin{matrix} \downarrow \\ A \\ \Upsilon \end{matrix}$ | $\begin{matrix} \downarrow \\ A \\ \uparrow \end{matrix}$ |
| (Axiom) | (Axiom) | (Axiom) | (Axiom) | (Cut) |
| hypothesis | hypothesis | –hypothesis | –hypothesis | |
| conclusion | –conclusion | conclusion | –conclusion | |

Example 7.7 The proof structure corresponding to the sequent proof of Figure 7.7 is shown in Figure 7.8 on the facing page.

There are five axiomatic formulas in this proof structure: both n formulas, both np formulas and the s formula, each corresponding to one instance of the axiom rule in the sequent proof.

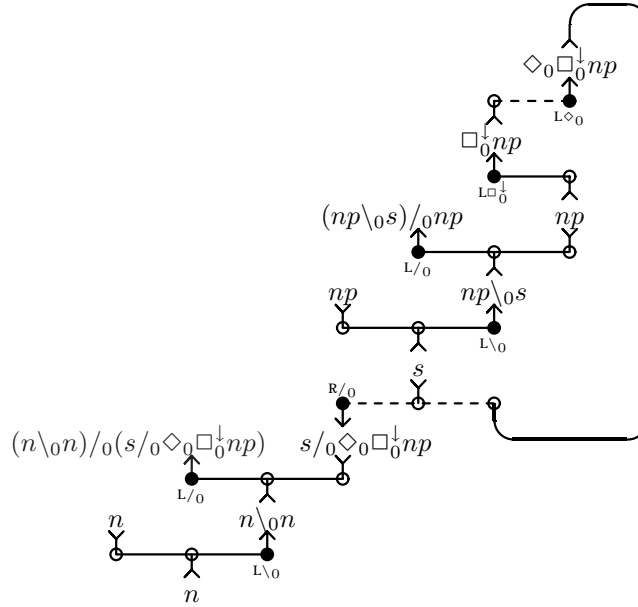


Figure 7.8: Proof structure corresponding to the derivation of Figure 7.7

Note that the antecedent formulas of the end-sequent correspond to the hypotheses of the proof structure and the succedent formula to its conclusion, and that every logical rule of the sequent proof corresponds to a link of the same name in the proof structure.

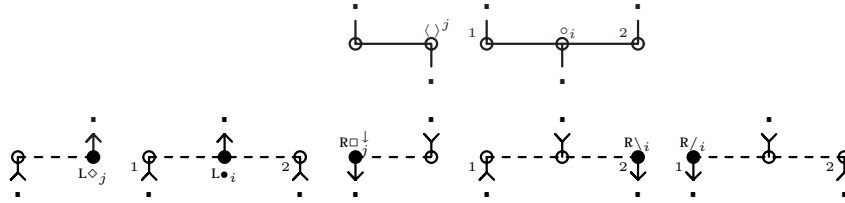
We can show by simple induction on the length of the sequent proof that for any given sequent proof we can construct a proof structure having the properties mentioned in the example above.

However, the converse does not hold: there are proof structures which do not correspond to any derivable sequent. A correctness criterion allows us to distinguish proof structures which *do* correspond to derivable sequents, *proof nets*, from other proof structures.

7.4 Abstract Proof Structures

To formulate our correctness criterion we need to convert our proof structures into slightly more abstract graphs. We will call these graphs *abstract proof structures* (aps's).

Definition 7.8 An abstract proof structure $\langle V, \mathcal{L} \rangle$ consists of a finite set V of vertices, where each vertex is assigned a sequence of premisses and a sequence of conclusions, together with a set \mathcal{L} of links in V of the following forms.



such that the following holds.

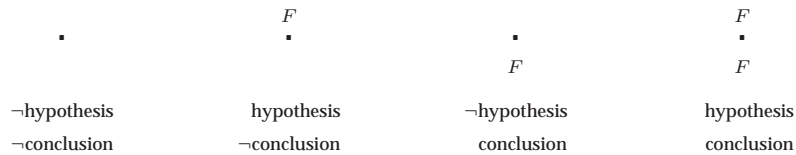
- every vertex of V is at most once a conclusion of a link,
- every vertex of V is at most once a premiss of a link.

Furthermore, we assign to each vertex a sequence of premisses and a sequence of conclusions, where we require that each vertex which is a hypothesis (resp. conclusion) of the structure has a single formula in its sequence of premisses (resp. conclusions) and for all other vertices this sequence is empty.

From a proof structure S we obtain an abstract proof structure \mathcal{A} by replacing all $[L/_i]$, $[L \setminus_i]$ and $[R \bullet_i]$ links by $[\circ_i]$ links and all $[L \square_j^{\perp}]$ and $[R \diamond_j]$ links by $[\langle \rangle^j]$ links. That is, we forget about the inputs and outputs of all tensor links.

In addition, we replace every formula F of S by a vertex v , which is assigned $[F]$ as its sequence of premisses if F is a hypothesis of S and $[\]$ otherwise, and which is assigned $[F]$ as its sequence of conclusions if F is a conclusion of S and $[\]$ otherwise.

Graphically, we will display premisses above and conclusions below their vertex, as shown below. The premisses and conclusion play no active role in our correctness criterion, they merely allow us to keep track of which formula occurrences are assigned to the hypotheses and conclusions of the proof structure.



We will write $S \mapsto \mathcal{A}$ to indicate that the aps \mathcal{A} is obtained from the proof structure S in this fashion. We will often write \widehat{S} for the abstract proof structure \mathcal{A} obtained from S .

Definition 7.9 A hypothesis tree is an acyclic, connected abstract proof structure containing only $[\circ_i]$ and $[\langle \rangle^j]$ links.

A hypothesis tree \mathcal{A} with hypotheses A_1, \dots, A_n and conclusion C corresponds to a sequent with antecedent formulas A_1, \dots, A_n and succedent formula C in the obvious way.

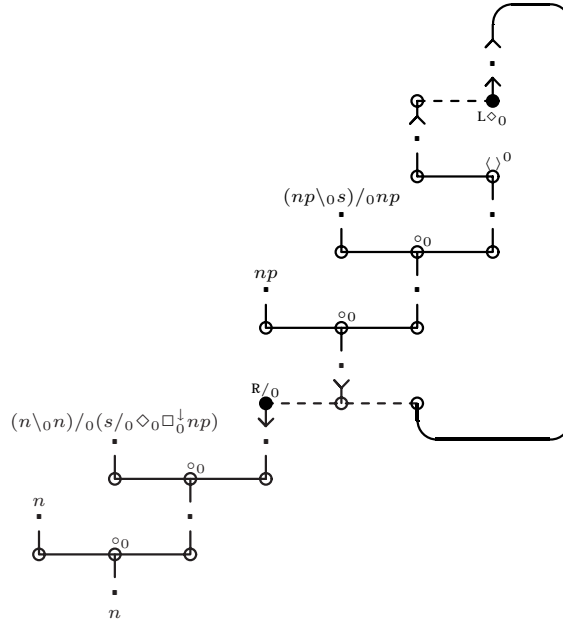


Figure 7.9: Abstract proof structure corresponding to Figure 7.8

We will write Γ_C for the hypothesis tree corresponding to the sequent $\Gamma \vdash C$.

Example 7.10 *The abstract proof structure computed for the proof structure of Figure 7.8 from Example 7.7 is shown in Figure 7.9.*

The abstract proof structure of the example above is not a hypothesis tree. This does not mean the original proof structure is not a *proof net*, of course. We will define a number of conversions on abstract proof structures: contractions, which are valid in the base logic, and structural conversions, which correspond to the structural rules in the sequent calculus.

By a *contraction* we will mean the replacement of one of the pairs of links shown in Table 7.9 on the following page by a single node. Contractions will be named after the par link. We require that all vertices shown in the redices of Table 7.9 are distinct. σ_H and σ_Q represent the sequence of hypotheses and the sequence of conclusions of the displayed vertex respectively.

Note that all contractions are a variation on the same theme: in every redex the ‘active vertices’ of a par link are connected to a single, neutral tensor link in a way which respects the left to right ordering and the reduct is a single node.

In addition to these contraction steps a grammar fragment can have a set \mathcal{R} of *structural conversions*. These conversions operate on trees of neutral tensor links only, with the condition that both trees in the conversion have the same set of leaves. This is a reflection of the same restriction on structural rules in the sequent calculus.

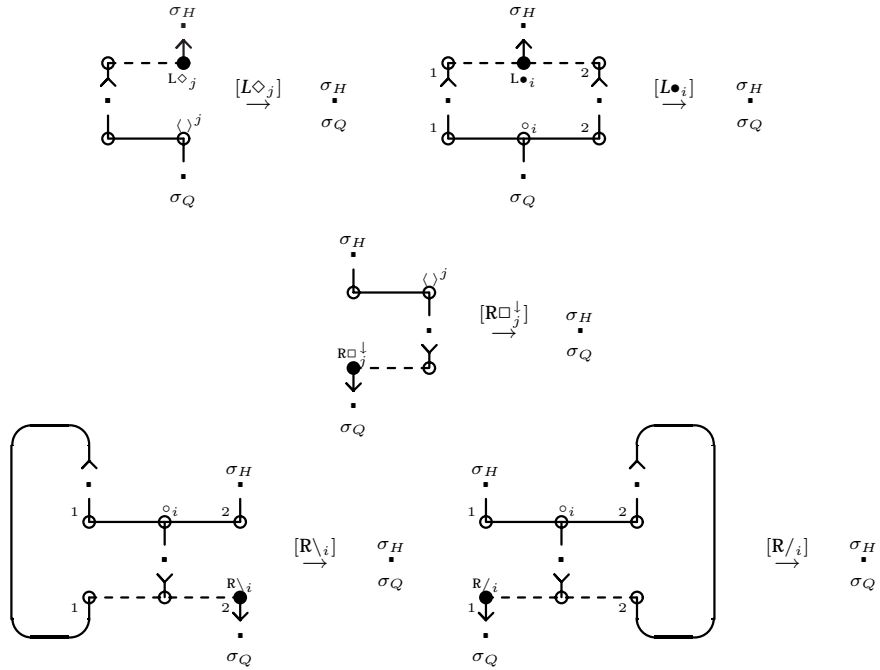
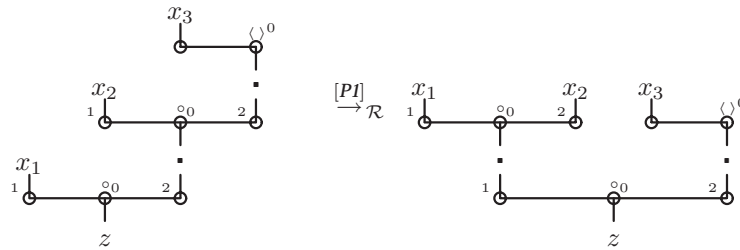


Table 7.9: The contractions for $NL\Diamond$

Example 7.11 *The following structural conversion corresponds to sequent rule [P1] of our examples.*

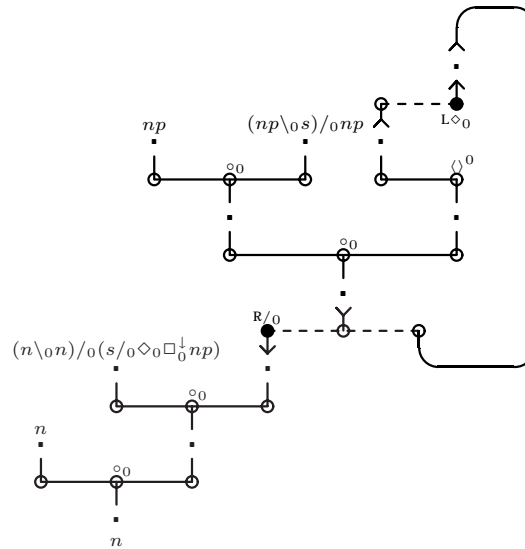


7.5 Proof Nets

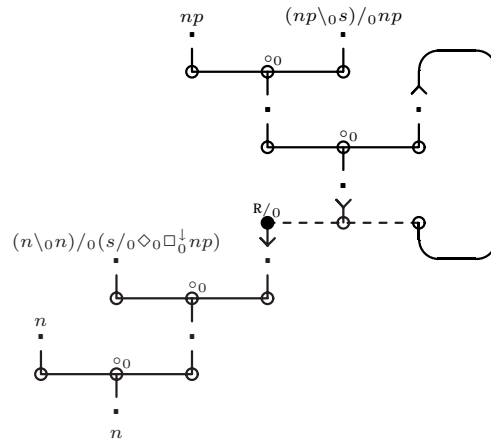
Definition 7.12 *A proof structure is a proof net if and only if its abstract proof structure converts to a hypothesis tree.*

Example 7.13 *The abstract proof structure of Example 7.10 converts to a tree given the structural conversion [P1], making this proof structure a proof net for any grammar fragment with this structural rule.*

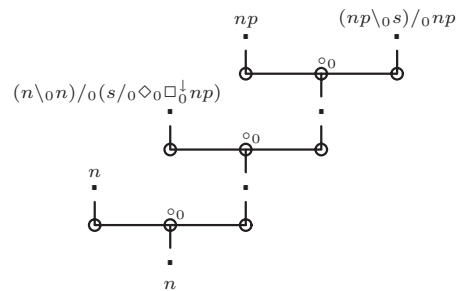
After applying the [P1] conversion, the abstract proof structure looks as follows.



Now we can apply the $[L \diamond_0]$ contraction, which results in the following abstract proof structure.



Finally, we can apply the $[R /_0]$ contraction and the result will be the following hypothesis tree



which corresponds to the end-sequent

$$n \circ_0 ((n \setminus_0 n) /_0 (s /_0 \diamond_0 \square_0^\perp np)) \circ_0 (np \circ_0 (np \setminus_0 s) /_0 np) \vdash n$$

of Example 7.4.

Note that we have computed the structure of the antecedent instead of assuming it as given and that we have performed the conversions in the exact same order as the corresponding rules in the sequent proof when read from axioms to end-sequent.

Before we prove our main theorem, we will first prove the following short lemma.

Lemma 7.14 *If S is a non-trivial proof structure such that the underlying abstract proof structure \widehat{S} is actually a hypothesis tree, then at least one of the leaves (conclusion and hypotheses) of S is the main formula of its link.*

Proof To prove: if every hypothesis is an active formula of its link, then the conclusion is the main formula of its link. We proceed by induction on the hypothesis tree Γ .

The trivial case $\Gamma = \cdot^A$ cannot occur.

In case $\Gamma = \Gamma_1 \circ_i \Gamma_2$, assume every hypothesis is an active formula of its link. We write L for the final \circ_i link, connecting Γ_1 and Γ_2 . If Γ_1 is trivial, the assumption entails that the corresponding formula in S is an active formula of L . If Γ_1 is non-trivial, by induction hypothesis we know that its conclusion is the main formula of the link above, whence of the form $\diamond_j A$ or $A \bullet_i B$. This implies that it is not the main formula of L , which would be $\square_j^\perp B$, $A \setminus_i B$ or $B /_i A$. Hence it is an active formula of L . The same holds for the second premiss of L . As both premisses are active, the conclusion of L must be main, as desired.

The case $\Gamma = \langle \Gamma_1 \rangle^j$ is proved analogously. \square

Theorem 7.15 *A sequent $\Gamma \vdash C$ is derivable in $\text{NL} \diamond_{\mathcal{R}}$ if and only if there is a proof structure S which converts to the hypothesis tree Γ_C , using only the contractions and the structural conversions in \mathcal{R} .*

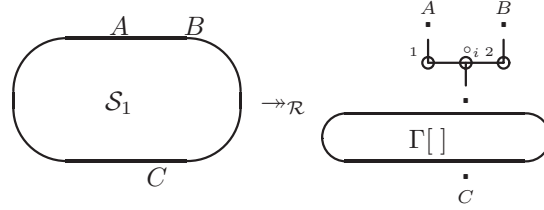
Proof

[\Rightarrow] From sequents to proof nets, we proceed by induction on the length of the sequent proof. We extend the conversion sequence with a contraction or a structural conversion whenever we encounter the corresponding rule in the sequent proof. By ‘applying a conversion step to a proof structure’ we will mean ‘applying a conversion step to its underlying abstract proof structure’.

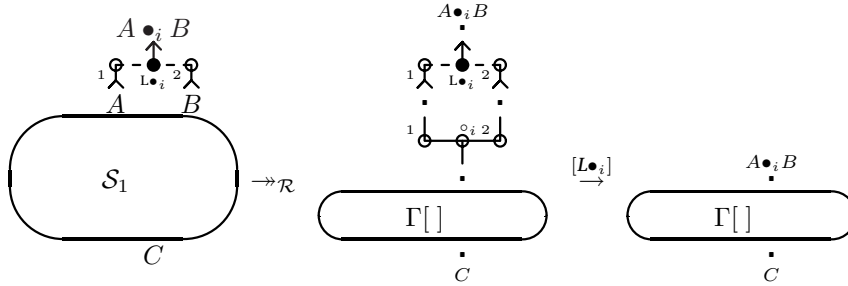
For a $[L \bullet_i]$ rule

$$\frac{\Gamma[(A \circ_i B)] \vdash C}{\Gamma[A \bullet_i B] \vdash C} [L \bullet_i]$$

we know the following by induction hypothesis.



We can keep the original conversion sequence, where we attach a $[L_{\bullet_i}]$ link to the abstract proof structure and we extend the conversion sequence with a $[L_{\bullet_i}]$ contraction as follows.



$[\Leftarrow]$ The sequentialization part of the proof proceeds in a way analogous to the ‘splitting par’ sequentialization proof of Danos (1990). We proceed by induction on the length l of the conversion sequence.

$[l = 0]$ If there are no conversions in the sequence, our proof structure corresponds to an abstract proof structure which is already a hypothesis tree. We proceed by induction on the number of tensor links in the proof structure.

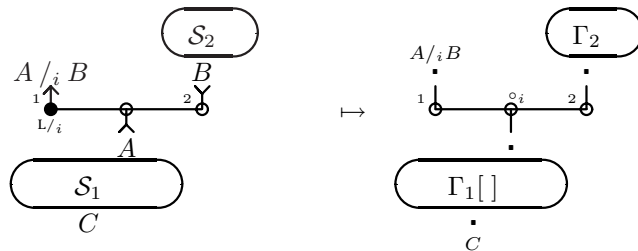
If there are no tensor links, our proof structure looks as follows

$$A \mapsto \begin{array}{c} A \\ \bullet \\ A \end{array}$$

and the corresponding sequent proof is $\frac{}{A \vdash A} [Ax]$.

If there are tensor links, then by Lemma 7.14 we know the proof structure has at least one main leaf, call it D .

In the sub-case where D is the main formula of a $[L/_i]$ link, D is of the form $A/_i B$ and must be the first premiss of this link. Now the proof structure S and the underlying hypothesis tree Γ_C are of the form



By induction hypothesis there are derivations \mathcal{D}_2 of $\Gamma_2 \vdash B$ and \mathcal{D}_1 of $\Gamma_1[A] \vdash C$, which may be combined into

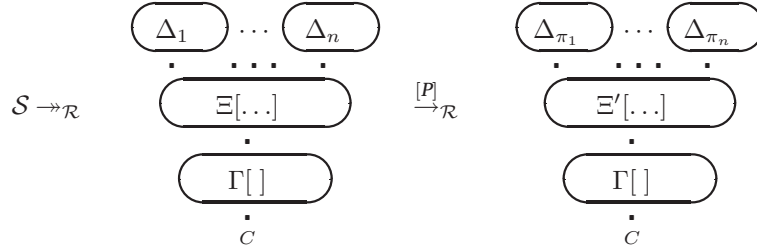
$$\frac{\begin{array}{c} \vdots \mathcal{D}_2 \\ \Gamma_2 \vdash B \end{array} \quad \begin{array}{c} \vdots \mathcal{D}_1 \\ \Gamma_1[A] \vdash C \end{array}}{\Gamma_1[(A/iB \circ_i \Gamma_2)] \vdash C} [L/i]$$

which is a derivation of $\Gamma \vdash C$.

The remaining sub-cases, where D is the main formula of a $[R\Diamond_j]$, $[R\bullet_i]$, $[L\Box^j]$ or $[L\setminus_i]$ link, are proved similarly.

$[l > 0]$ We look at the last conversion in the sequence.

If it is a structural conversion $\Xi[x_1, \dots, x_n] \xrightarrow{[P]}_{\mathcal{R}} \Xi'[x_{\pi_1}, \dots, x_{\pi_n}]$, we are in the following situation.



The induction hypothesis gives us a derivation \mathcal{D}_1 of

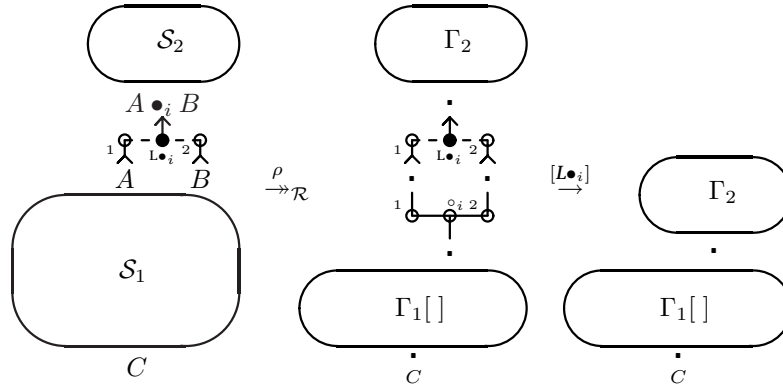
$$\Gamma[\Xi[\Delta_1, \dots, \Delta_n]] \vdash C$$

which we can extend as follows

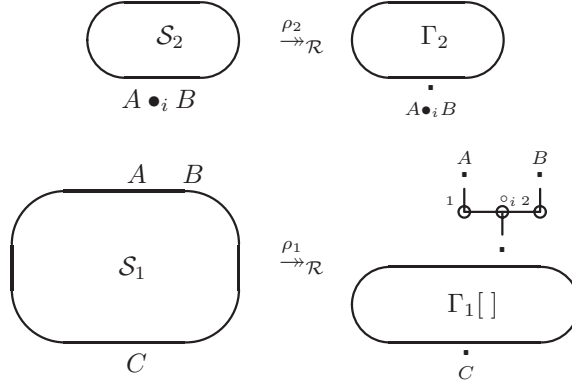
$$\frac{\begin{array}{c} \vdots \mathcal{D}_1 \\ \Gamma[\Xi[\Delta_1, \dots, \Delta_n]] \vdash C \end{array}}{\Gamma[\Xi'[\Delta_{\pi_1}, \dots, \Delta_{\pi_n}]] \vdash C} [P]$$

to give us a derivation of $\Gamma \vdash C$.

If the last conversion is a $[L\bullet_i]$ contraction we are schematically in the following situation.



Reasoning backwards from the hypothesis tree, we can see that the $[L\bullet_i]$ link serves as a boundary and that every conversion in the sequence is either applied strictly above or strictly below it. So we can split our initial conversion sequence ρ into a conversion sequence ρ_1 which converts S_1 to a hypothesis tree and a conversion sequence ρ_2 which converts S_2 into a hypothesis tree as follows



As both conversion sequences are strictly smaller than our initial conversion sequence, the induction hypothesis gives us a derivation \mathcal{D}_2 ending in $\Gamma_2 \vdash A \bullet_i B$ and a derivation \mathcal{D}_1 ending in $\Gamma_1[(A \circ_i B)] \vdash C$. We can combine these derivations as shown below

$$\frac{\frac{\frac{\vdots \mathcal{D}_2}{\Gamma_2 \vdash A \bullet_i B} \quad \frac{\frac{\vdots \mathcal{D}_1}{\Gamma_1[(A \circ_i B)] \vdash C} \quad [L\bullet_i]}{\Gamma_1[A \bullet_i B] \vdash C} [Cut]}{\Gamma_1[\Gamma_2] \vdash C} [Cut]}$$

The other contractions are similar. \square

The sequentialization part of our proof has the somewhat inelegant property that it produces sequent proofs which use the $[Cut]$ rule even for proof nets without cut formulas. However, we can refine the proof of Theorem 7.15 in such a way that the sequent proofs we produce have exactly one $[Cut]$ rule application for each cut formula. For this purpose we first state the following lemma.

Lemma 7.16 (Substitution) *Let \mathcal{D}_1 be a derivation of $\Gamma_1 \vdash C_1$ and \mathcal{D}_2 be a derivation of $\Gamma_2[C_1] \vdash C_2$.*

- (i) *If $C_1 \vdash C_1$ is an axiom of \mathcal{D}_1 , the succedent formula of which coincides with the succedent formula of $\Gamma_1 \vdash C_1$, then we can substitute \mathcal{D}_2 into \mathcal{D}_1 in order to get a derivation $\mathcal{D}_1[\mathcal{D}_2]$ of $\Gamma_2[\Gamma_1] \vdash C_2$.*

$$\frac{\begin{array}{c} C_1 \vdash C_1 \\ \vdots D_1 \\ \Gamma_1 \vdash C_1 \end{array} \quad \begin{array}{c} \vdots D_2 \\ \Gamma_2[C_1] \vdash C_2 \end{array}}{\Gamma_2[\Gamma_1] \vdash C_2} [Cut] \quad \text{becomes} \quad \begin{array}{c} \vdots D_2 \\ \Gamma_2[C_1] \vdash C_2 \\ \vdots D_1 \\ \Gamma_2[\Gamma_1] \vdash C_2 \end{array}$$

- (ii) If $C_1 \vdash C_1$ is an axiom of \mathcal{D}_2 , the antecedent formula of which coincides with the occurrence in $\Gamma_2[C_1] \vdash C_2$, then we can substitute \mathcal{D}_1 into \mathcal{D}_2 in order to get a derivation $\mathcal{D}_2[\mathcal{D}_1]$ of $\Gamma_2[\Gamma_1] \vdash C_2$.

$$\frac{\begin{array}{c} C_1 \vdash C_1 \\ \vdots D_1 \\ \Gamma_1 \vdash C_1 \end{array} \quad \begin{array}{c} \vdots D_2 \\ \Gamma_2[C_1] \vdash C_2 \end{array}}{\Gamma_2[\Gamma_1] \vdash C_2} [Cut] \quad \text{becomes} \quad \begin{array}{c} \vdots D_1 \\ \Gamma_1 \vdash C_1 \\ \vdots D_2 \\ \Gamma_2[\Gamma_1] \vdash C_2 \end{array}$$

Proof In general every leaf of a tree determines a path to the root. In particular every axiom rule of a derivation determines a path of sequents from that axiom to the conclusion of the derivation. Let $\Gamma \vdash \Delta$ and $\Gamma' \vdash \Delta'$ be two successive sequents in a certain path β , i.e. $\Gamma' \vdash \Delta'$ is the conclusion of an inference rule with $\Gamma \vdash \Delta$ among its hypotheses.

For a binary inference rule R we say that β passes R via the left (right) hypothesis if $\Gamma \vdash \Delta$ is the first (second) hypothesis of R with respect to the formulation of Figure 7.5.

- (i) As the occurrence C_1 is preserved along the path β in \mathcal{D}_1 between $C_1 \vdash C_1$ and $\Gamma_1 \vdash C_1$, the possible inference rules β passes are $[Cut]$ (via the left hypothesis), the left logical rules (if binary, then via the left hypothesis), or a structural rule. Each of these rules has the property that if

$$\frac{\Gamma_1 \vdash C_1 \quad (\Gamma_0 \vdash C_0)}{\Gamma_3 \vdash C_1}$$

is an instance, then so is

$$\frac{\Gamma_2[\Gamma_1] \vdash C_2 \quad (\Gamma_0 \vdash C_0)}{\Gamma_2[\Gamma_3] \vdash C_2}$$

- (ii) As the occurrence C_1 is preserved along the path β in \mathcal{D}_2 between $C_1 \vdash C_1$ and $\Gamma_2[C_1] \vdash C_2$, it will never be an active formula in any inference rule β passes. Hence if

$$\frac{\Gamma_2[C_1] \vdash C_2 \quad (\Gamma_0 \vdash C_0)}{\Gamma_3[C_1] \vdash C_3}$$

is an instance of a rule, then so is

$$\frac{\Gamma_2[\Gamma_1] \vdash C_2 \quad (\Gamma_0 \vdash C_0)}{\Gamma_3[\Gamma_1] \vdash C_3}$$

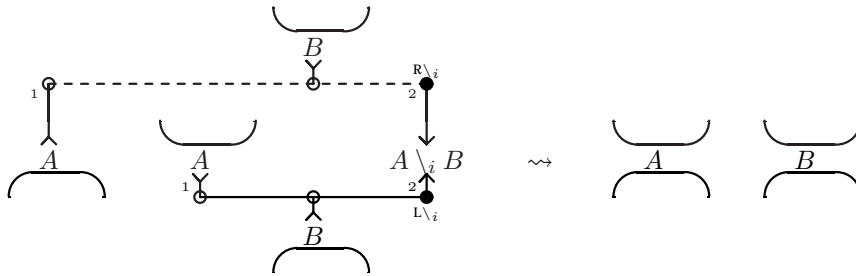
□

Now extend the proof of Theorem 7.15 by simultaneously showing that every axiomatic formula corresponds to an [Ax] rule, and moreover that every axiomatic conclusion corresponds to an axiom as in Lemma 7.16.1 and every axiomatic hypothesis corresponds to an axiom as in Lemma 7.16.2. We adapt the proof in the case that $l > 0$ and the last conversion step is a contraction: if the main formula of L (say: D) is a cut formula of S we proceed as described earlier. However, if D is not a cut formula, then D is an axiomatic leaf of one of the two substructures, whence we can apply the par rule followed by the appropriate substitution.

7.6 Cut Elimination

One important property of proof net calculi and logics in general is cut elimination. Given a proof net with a number of cut formulas, we want to find a proof net without cut formulas which converts to the same hypothesis tree. In investigating cut elimination, we will give our notion of conversion sequence slightly more structure, touching upon reordering the conversions in such a way that the sequence satisfies certain properties necessary for cut elimination.

Recall that a cut formula is a formula which is the main formula of two dual links. A cut reduction step, $S \rightsquigarrow S'$, is defined as deleting these links and the cut formula, while pairwise identifying the active formulas in case they are different (as occurrence of the same formula), or deleting them if they are identical.



Let D be a cut formula and L the corresponding par link. We will show that if (S, ρ) is a conversion sequence ending in hypothesis tree Γ_C , then so is (S', ρ') , where $S \rightsquigarrow S'$, and ρ' consists of the same set of conversion steps as ρ , except the contraction α of L , in a sense to be made precise shortly.

Before proving the cut elimination theorem, we first introduce the auxiliary notations of component and block and make some observations about their properties.

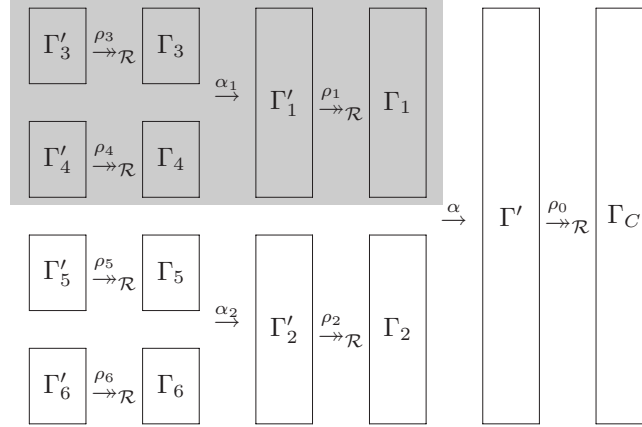


Figure 7.10: The components of a certain conversion sequence with three contractions

Definition 7.17 Let $\mathcal{A} = \langle V, \mathcal{L} \rangle$ be an abstract proof structure. $\mathcal{A}' = \langle V', \mathcal{L}' \rangle$ is a component of \mathcal{A} if \mathcal{A}' is a substructure of \mathcal{A} , where \mathcal{L}' contains only tensor links and \mathcal{A}' is maximally connected with respect to the tensor links of \mathcal{L} .

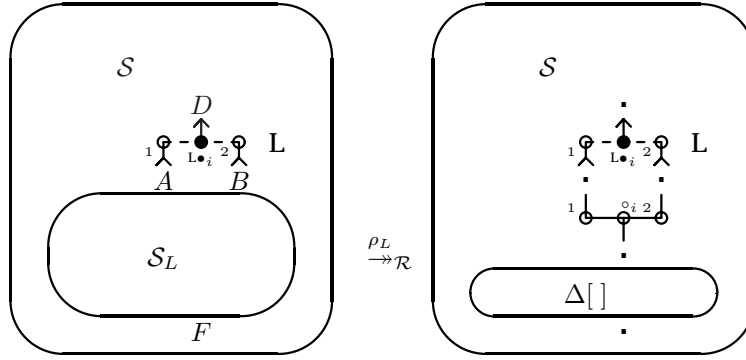
For a proof structure \mathcal{S} , a substructure \mathcal{S}' is a component iff $\widehat{\mathcal{S}'}$ is a component of $\widehat{\mathcal{S}}$.

When we delete all p par links (but not their vertices) from an abstract proof structure \mathcal{A} the notion of component as defined above coincides with the definition of component from graph theory. Observe that a component may consist of one vertex only.

For a *proof net*, its components are $p + 1$ hypothesis trees. This even holds for all intermediate abstract proof structures between \mathcal{S} and Γ_C : reasoning backwards from the final hypothesis tree, we start with one component ($p = 0$). After a number of structural conversions, a contraction α splits this component Γ' into two parts and replaces one node by a redex. The par link L of this redex now serves as a boundary between the two new components Γ_1 , which is attached to the active formulas A and B of L , and Γ_2 , which is attached to the main formula D of L . At this moment Γ_1 is a *nice* hypothesis tree w.r.t. L , i.e. attaching L enables its contraction α .

All next structural conversions take place completely within one of the two components, and the next contraction takes place in exactly one of the two components as well. In this way every par contraction replaces one component by two new components, yielding $p + 1$ components in each abstract proof structure.

Figure 7.10 gives an illustration of how we can factor the different conversions of ρ into components. Another way of seeing this is that ρ determines a rooted, ordered *tree* of hypothesis trees, with the initial components as the leaves and the final hypothesis tree as the root, where, reasoning backwards

Figure 7.11: The block of a $[L\bullet_i]$ link L

from the final hypothesis tree, every structural conversion produces a unary branch and every contraction a binary branch.

Definition 7.18 (Block) Let (S, ρ) be a conversion sequence ending in Γ_C . Let L be a par link of S and α be the contraction in ρ corresponding to L .

The conversion sequence ρ looks as follows.

$$S \xrightarrow{\rho_1} \mathcal{S}_1 \xrightarrow{\alpha} \mathcal{S}_2 \xrightarrow{\rho_2} \Gamma_C$$

We define a subnet (S_L, ρ_L) called the block of L by induction on the length of ρ_1 as follows. As before, when we talk about applying a conversion to a proof structure we will mean applying the conversion to the underlying abstract proof structure.

If $\|\rho_1\| = 0$ then S_L is the component of the active formulas of L and ρ_L is empty.

If $\|\rho_1\| > 0$ then we are in the following situation.

$$S \xrightarrow{\delta_0} S' \xrightarrow{\rho'_1} \mathcal{S}_1$$

Induction hypothesis gives us (S'_L, ρ'_L) , which is the block for the shorter conversion sequence starting with S' . If the reduct of δ_0 is not in S'_L then $(S_L, \rho_L) = (S'_L, \rho'_L)$. If the reduct of δ_0 is in S'_L then ρ_L is ρ'_L with the conversion δ_0 prefixed and S_L is S'_L with the reduct of δ_0 replaced by its redex.

Figure 7.11 gives an illustration what the block of the $[L\bullet_i]$ looks like schematically with respect to the full proof net S . By construction, we can replace the conversion sequence ρ for S by the following, where $\tilde{\rho}_L$ is the conversion sequence which contains all conversions of ρ not in ρ_L with the exception of α .

$$\begin{array}{ccccccc} S & \xrightarrow{\rho_L} & \mathcal{A} & \xrightarrow{\alpha} & \mathcal{B} & \xrightarrow{\tilde{\rho}_L} & \Gamma_C \\ \cup & & \cup & & & & \\ S_L & \xrightarrow{\rho_L} & \Delta_F & & & & \end{array}$$

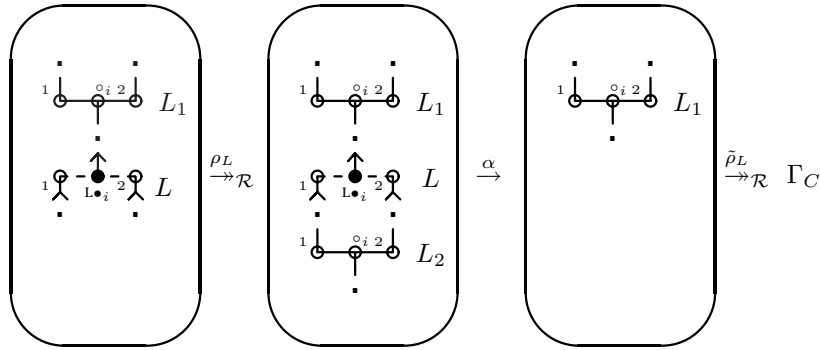
Example 7.19 In Figure 7.10, assuming Γ_1 is the component containing the active formulas of L , the block of L is (S_L, ρ_L) , where S_L consists of the components Γ'_3 and Γ'_4 connected by the par link contracted in step α_1 and where the sequence ρ_L consists of the conversions in the gray area of the figure.

Theorem 7.20 (Cut elimination) If S is a proof net converting to Γ_C , and $S \rightsquigarrow S'$ by a cut reduction step, then S' is a proof net converting to Γ_C as well.

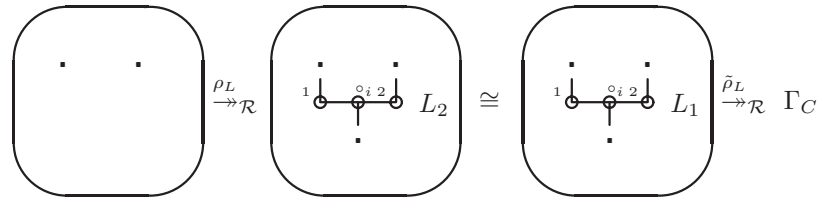
Proof Let α be the contraction corresponding to the par link L which is removed by the cut reduction step. As observed above, we can replace the conversion sequence ρ by

$$\widehat{S} \xrightarrow{\rho_L} \mathcal{A} \xrightarrow{\alpha} \mathcal{B} \xrightarrow{\tilde{\rho}_L} \Gamma_C$$

that is, we are schematically in the following situation, where L_1 remains untouched during ρ_L .



Executing a cut reduction step yields the situation pictured below.



which proves the result. □

7.7 Abstract Proof Structures and Labels

We will now briefly sketch the relationship between the structural labels of the previous chapter and the abstract proof structures of the current chapter.

The basic idea is that for a given structural label every sublabel corresponds to a vertex in the corresponding abstract proof structure and every

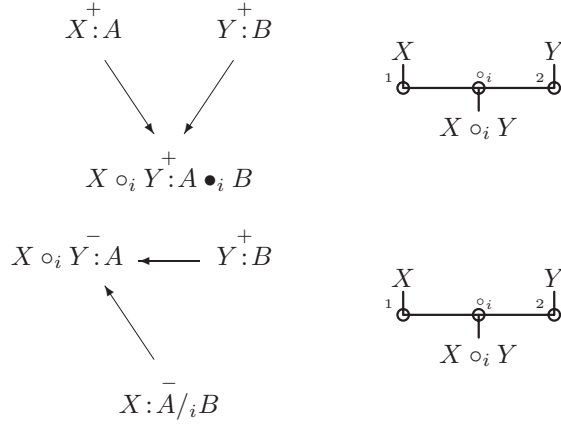


Table 7.10: Structural labels and abstract proof structures: tensor cases

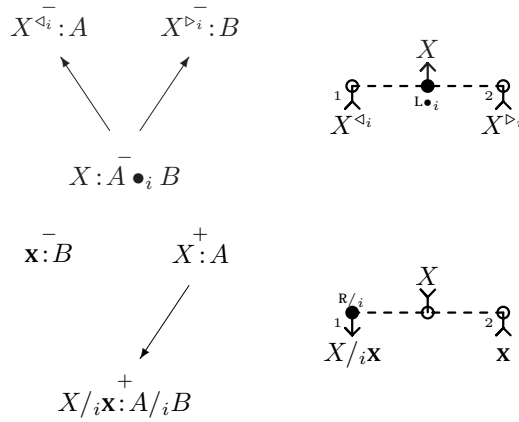


Table 7.11: Structural labels and abstract proof structures: par cases

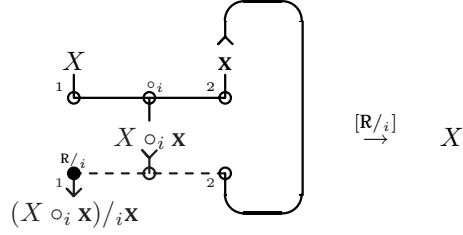
auxiliary constructor corresponds to a par link in the abstract proof structure and the other constructors correspond to tensor links.

First, note that the dynamic graphs for the axiom and the cut link, shown in Table 6.3 on page 94 have no reflex of this link in the structural label. Similarly, axiom and cut do not correspond to a link in the abstract proof structures.

Now compare the dynamic graphs for the tensor links of ‘•’ and ‘/’ to the corresponding abstract proof structures. Table 7.10 shows this relation. To make the relation more clear we label the vertices of the aps with the corresponding term label. The formula which is labeled with the complex label is always the conclusion of the link in the abstract proof structure.

Finally, compare the labels for the par links for ‘•’ and ‘/’ with the corresponding abstract proof structure. Table 7.11 shows them next to each other.

As shown in the table, the single $[L \bullet_i]$ link corresponds two auxiliary constructors: the $X^{<i}$ and the $X^{>i}$ constructor. Because the structural labels are

Figure 7.12: Contraction for a $[R/i]$ link with labeling

$$\begin{array}{c}
 \frac{}{A \vdash A} [Ax] \qquad \frac{\Gamma, A, \Gamma' \vdash C \quad \Delta \vdash A}{\Gamma, \Delta, \Gamma' \vdash C} [Cut] \\
 \\
 \frac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, A \bullet B, \Gamma' \vdash C} [L\bullet] \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \bullet B} [R\bullet] \\
 \\
 \frac{\Gamma, A, \Gamma' \vdash C \quad \Delta \vdash B}{\Gamma, A/B, \Delta, \Gamma' \vdash C} [L/] \qquad \frac{C, \Gamma, B \vdash A}{C, \Gamma \vdash A/B} [R/] \\
 \\
 \frac{\Gamma, A, \Gamma' \vdash C \quad \Delta \vdash B}{\Gamma, \Delta, B \setminus A, \Gamma' \vdash C} [L\setminus] \qquad \frac{B, \Gamma, C \vdash A}{\Gamma, C \vdash B \setminus A} [R\setminus]
 \end{array}$$

Table 7.12: The sequent calculus **L**

essentially trees, the par link for $[L\bullet_i]$ cannot be expressed by a single constructor. In the abstract proof structure, we treat the two occurrences of the X label as the same vertex. For the $[R/i]$ link the x label will also occur twice in the structural label and we will treat both occurrences as the same vertex in the abstract proof structure.

When we compare the label conversions of Table 6.6 on page 97 to the graph contractions of Table 7.9 on page 112 we note that they express the same restrictions. Figure 7.12 shows the $[R/i]$ contraction on an abstract proof structure, again with the structural labels on the vertices to make the correspondence more clear.

7.8 Lambek Calculus

In this section a contraction criterion for the Lambek calculus will be formulated and proved. This criterion is a combination of Danos' contraction criterion for one sided **MLL** (Danos 1990) and Lafont's criterion for parsing boxes (Lafont 1995). The contraction relation is terminating, though not confluent. However, we achieve confluence on a restricted domain, leading us to the main contraction theorem, Theorem 7.28. Our contraction criterion has the special property that a priori there is no order on the leaves of the

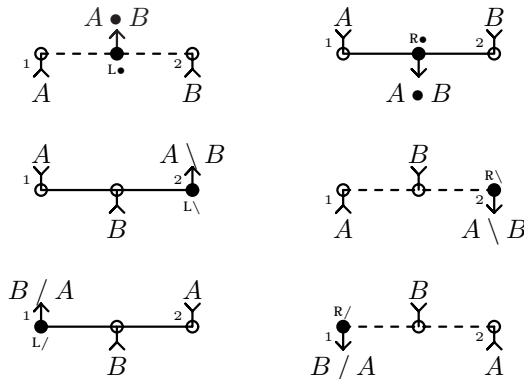
proof structure; if the proof structure is *correct* (in the sense that it contracts properly), our criterion a posteriori provides the unique order of the leaves.

The Lambek calculus (**L**), as introduced by Lambek (1958), is defined by the inference rules of Table 7.12 on the facing page, where the antecedent part of each sequent is a non-empty sequence rather than a structure tree. In our formulation of the calculus, preservation of non-empty antecedent parts during applications of the rules $[R/]$ and $[R\setminus]$ is forced by the presence of the extra formula C .

The Lambek calculus is equivalent to the special case of $NL\Diamond_{\mathcal{R}}$ with zero unary modes, one binary mode and no structural rules but associativity. The latter mimics the fact that each sequent has a sequence instead of a structure tree as antecedent part. In this way Theorem 7.15 provides us with a correctness criterion for **L**, since **L** derives $C_1, \dots, C_n \vdash C$ precisely if this special instance of $NL\Diamond_{\mathcal{R}}$ derives $C_1 \circ (C_2 \circ (\dots (C_{n-1} \circ C_n) \dots)) \vdash C$.

However, we can obtain a more attractive correctness criterion when by adapting our theory in such a way that the structural rules become part of the theory and are not present explicitly anymore. This is done by a generalization of the links in the definition of abstract proof structure.

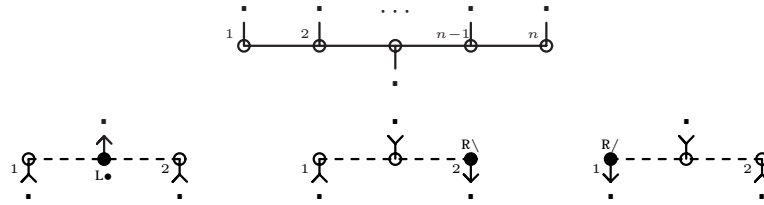
Definition 7.21 An **L**-proof structure $\langle S, \mathcal{L} \rangle$ consists of a finite set S of (\diamond - and \square^\perp -free and unimodal) formulas together with a set \mathcal{L} of links in S of the following forms.



such that the following holds.

- every formula of S is at most once a conclusion of a link,
- every formula of S is at most once a premiss of a link.

Definition 7.22 An abstract **L**-proof structure $\langle V, \mathcal{L} \rangle$ consists of a finite set V of nodes together with a set \mathcal{L} of links in V of the following forms (where $n \geq 2$).



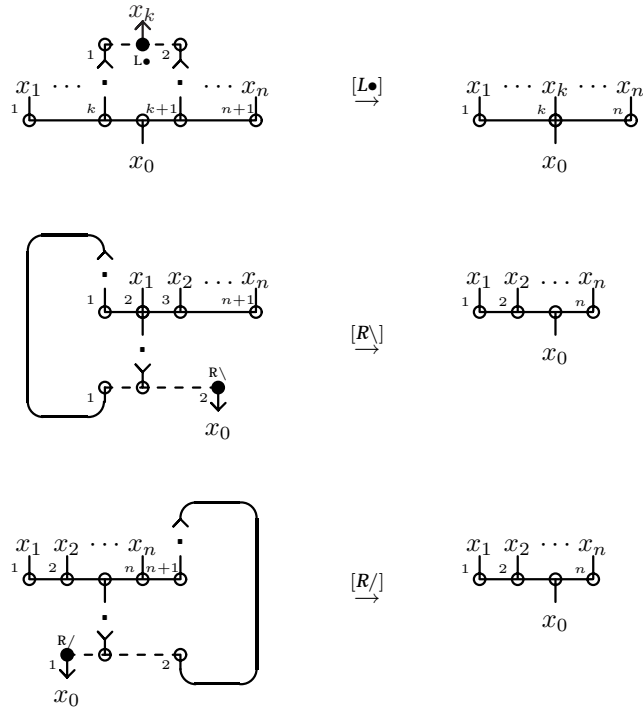
such that the following holds.

- every node of V is at most once a conclusion of a link,
- every node of V is at most once a premiss of a link.

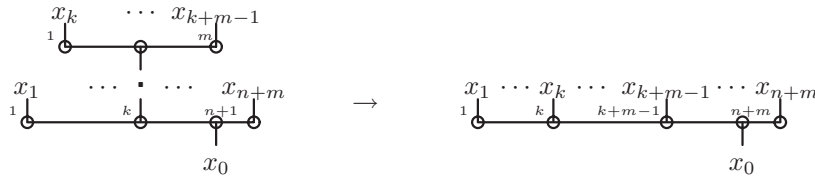
Furthermore, we assign to each node a sequence of premisses and a sequence of conclusions, as in Definition 7.8.

The generalized tensor link will be called an n -comb ($n \geq 2$). For practical reasons, we define a 1-comb to be a single node; notice that thus a 1-comb is not a link, contrary to n -combs with $n \geq 2$.

The redex of a contraction consists of a par link and an $(n + 1)$ -comb ($n + 1 \geq 2$), as depicted below (where we require — as usual — all nodes to be distinct). Observe that in every case the par link is attached to two successive formulas of the $(n + 1)$ -comb, when we order them in a cyclic way. It is replaced by an n -comb (which is a single node if $n = 1$), and all nodes keep their labels. The contraction will be named after the par link ($[L\bullet]$, $[R\backslash]$, $[R/]$).



By a *structural conversion* we mean the following composition of combs ($n + 1, m \geq 2$).



Now, starting with a proof structure \mathcal{S} , we can form the underlying abstract proof structure $\widehat{\mathcal{S}}$ in the usual way (which — besides nodes — consists of par links and 2-combs only).

For any non-empty sequence Γ and formula C , let $\|\Gamma\|$ be the multiset of elements in Γ ; let Γ_C be the obvious abstract proof structure (consisting of one n -comb, $n \geq 1$) with conclusion node (lower) labeled by C . Any abstract proof structure of this form will be called a *hypothesis comb*. Let \rightarrow be the transitive, reflexive closure of \rightarrow , by which we mean the contractions as well as the structural conversions.

It is easy to see that this conversion relation is terminating; in each conversion step at least one link disappears.

Theorem 7.23 $\Gamma \vdash C$ is derivable in \mathbf{L} if and only if there is a proof structure \mathcal{S} such that $\widehat{\mathcal{S}} \rightarrow \Gamma_C$.

Proof The proof is similar to that of Theorem 7.15: it can be shown that for any derivation \mathcal{D} of $\Gamma \vdash C$ the corresponding proof structure converts to the hypothesis comb Γ_C .

The other way around, we can prove that a proof structure \mathcal{S} that converts to a hypothesis comb Γ_C is actually the proof structure of a derivation \mathcal{D} of $\Gamma \vdash C$. \square

Given a proof structure \mathcal{S} with p par links, we define a switching ω for \mathcal{S} to be a choice, for each par link L , of one of the active ends of L . The *correction graph* $\omega\mathcal{S}$ of \mathcal{S} under the switching ω is obtained by replacing each par link by the chosen active end. Let \mathcal{PS}' denote the collection of those elements \mathcal{S} of \mathcal{PS} , the set of all proof structures, for which all 2^p correction graphs $\omega\mathcal{S}$ are trees.

Lemma 7.24 Let $\mathcal{S}_1, \mathcal{S}_2 \in \mathcal{PS}$ and suppose $\mathcal{S}_1 \rightarrow \mathcal{S}_2$. Then $\mathcal{S}_1 \in \mathcal{PS}'$ if and only if $\mathcal{S}_2 \in \mathcal{PS}'$.

In particular, the conversion steps are well defined on \mathcal{PS}' (i.e. they do yield an element of \mathcal{PS}' when applied on an element of \mathcal{PS}').

Since hypothesis combs belong to \mathcal{PS}' , we immediately obtain the next result.

Corollary 7.25 *If a proof structure S converts to a hypothesis comb Γ_C , then $S \in \mathcal{PS}'$.*

So proof nets (the proof structures that convert to a hypothesis comb) will only be found in \mathcal{PS}' . Now confluence of this conversion relation on \mathcal{PS}' is easily proved. This is a consequence of firstly the absence of cycles in the correction graphs, and secondly the absence of the unary connectives which already destroy confluence on general $\mathbf{NL}\diamond_{\mathcal{R}}$.

Lemma 7.26 *If $S \in \mathcal{PS}'$ converts in one step to S_1 and S_2 , then both S_1 and S_2 convert in at most one step to a common $S_3 \in \mathcal{PS}'$.*

By means of Lemma 7.26 and termination, we can sharpen Theorem 7.23 into the following.

Theorem 7.27 *Let $\Gamma \vdash C$ be a sequent. Then the following are equivalent.*

- (i) $\Gamma \vdash C$ is derivable in \mathbf{L} ;
- (ii) *There is a proof structure S such that all conversion sequences $S \twoheadrightarrow S'$ (where S' is a hypothesis comb) satisfy $S' = \Gamma_C$.*

Theorem 7.28 *Let S be a proof structure and $S \twoheadrightarrow S'$ be an arbitrary conversion sequence to a normal form. Then S is the proof structure of a derivation if and only if S' is a hypothesis comb.*

Again the following holds.

Theorem 7.29 (Cut elimination) *If S is a proof net converting to Γ_C , and $S \rightsquigarrow S'$ by a cut reduction step, then S' is a proof net converting to Γ_C as well.*

7.9 Discussion

We have presented a proof net calculus for the multimodal Lambek calculus which is new, elegant and very general. By giving a correctness criterion for \mathbf{L} , we have also shown how our correctness criterion can function as a sort of meta correctness criterion which can be used to produce a correctness criterion for special instances of $\mathbf{NL}\diamond_{\mathcal{R}}$.

The formalism we have presented here is related to a number of other proposals, notably to Danos's (1990) graph contractions, of which our contractions are a special case. As a result, acyclicity and connectedness of the underlying correction graphs are a consequence of our correctness criterion.

We have also sketched the relation between abstract proof structures and the structural labels of the labeled proof nets of Moortgat (1997) we discussed in Chapter 6. Advantages of our formalism are that we have a very direct correspondence between proof structures and abstract proof structures and

that cyclic or disconnected proof structures are unproblematically disqualified by our correctness criterion. The algebraic correctness criterion will fail to compute a meaningful label for cyclic or disconnected proof structures.

It is possible to overcome the formal difference between proof structures and abstract proof structures. Puite (2001) introduces the notion of *link graph* for this purpose. Link graphs comprise both proof structures and hypothesis trees, which also play a role as sequents for the calculus. By means of this new notion Puite proves a correctness criterion for **CNL**, the classical non-associative Lambek calculus (de Groote & Lamarche 2001), along the lines of the proof of Theorem 7.15.

PART III

RELATIONS AND COMPUTATIONS

One of the attractive aspects of proof nets as discussed in the previous chapter is that they lend themselves well to automated proof search. First of all, in Section 7.6 we saw that we could eliminate cut formulas from proof nets, making it unnecessary to consider cut formulas in our proof search. Secondly, we can restrict ourselves to proof nets where all our axiomatic formulas are atomic, as indicated by the following lemma.

Lemma 8.1 *Given a proof structure S we can construct a proof structure S' with the same hypotheses and conclusions where all axiomatic formulas are atomic and where $\widehat{S}' \rightarrow_{\emptyset} \widehat{S}$. We will call such a proof structure eta expanded.*

Proof By induction on the total complexity of the axiomatic formulas.

If there are no complex axiomatic formulas in the proof structure, we take $S' = S$ and an empty conversion sequence.

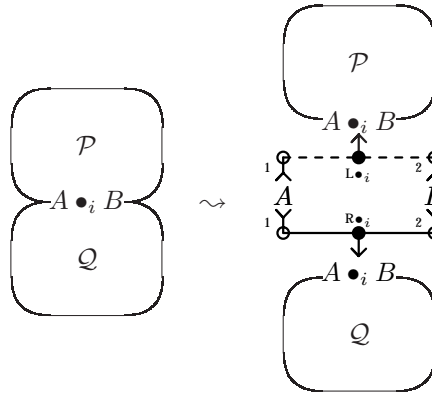
If we have a proof structure S_0 where the axiomatic formulas have $n + 1$ total connectives, we can expand a complex axiomatic $A \bullet_i B$ formula as shown in Figure 8.1. The other connectives are treated similarly. The resulting proof structure S_1 will have two new axiomatic formulas and the total number of connectives of axiomatic formulas will be n .

By induction hypothesis we know that $\widehat{S}'_1 \rightarrow_{\emptyset} \widehat{S}_1$, so we can suffix a $[L_{\bullet_i}]$ contraction producing the following conversion sequence.

$$\widehat{S}'_1 \rightarrow_{\emptyset} \widehat{S}_1 \xrightarrow{[L_{\bullet_i}]} \widehat{S}_0$$

As we use only contractions, the theorem holds regardless of the structural rules. \square

The following corollary is an immediate consequence of Theorem 7.20 and Lemma 8.1.

Figure 8.1: Eta expansion step for a $A \bullet_i B$ formula

Corollary 8.2 *For every proof net \mathcal{P} of $\Gamma \vdash C$ there exists a proof net \mathcal{P}' , also of $\Gamma \vdash C$, which is cut free and eta expanded.*

So we can, without loss of generality, restrict ourselves to proof structures where all complex formulas are neither axiomatic nor cut formulas. A simple algorithm for the enumeration of cut free, eta expanded proof nets is shown in Table 8.1 on the next page.

We assume computation is nondeterministic, i.e. the steps of our algorithm can produce a number of solutions: the lexicon can produce different formulas for each word, there can be many different ways of identifying the atomic formulas and we might be able to convert our abstract proof structure to many different hypothesis trees. When one step in our algorithm fails to produce a solution, we backtrack to a previous step and try the next solution there until we have found all solutions.

The set of parameters on the final hypothesis tree can restrict the output of the algorithm in any of the following ways.

- (i) left to right traversal of the hypothesis tree yields the formulas in the order indicated by the input sequence.
- (ii) only binary modes from $I' \subseteq I$ and unary modes $J' \subseteq J$ can occur in the hypothesis tree.
- (iii) return only the shortest conversion sequence(s).

When we use our algorithm for parsing a sentence, we typically want to satisfy condition (i). However, it can be useful to see *all* different hypothesis trees for the current proof structure because this might reveal ungrammatical sentences which are derivable with the current lexicon and structural rules.

Sometimes it makes sense to disallow certain modes from appearing in the final hypothesis tree, as indicated by condition (ii). We call a mode $i \in I'$ or $j \in J'$ *external* and a mode $i \in I \setminus I'$ or $j \in J \setminus J'$ an *internal* mode.

Input – sequence w_1, \dots, w_n of words
 – lexicon l , which assigns formulas to words
 – set of goal formulas Q
 – set \mathcal{R} of structural rules
 – set P of parameters restricting the shape of the final hypothesis tree

Output set of cut free, eta expanded proof nets with hypotheses $l(w_1), \dots, l(w_n)$ and conclusion $q \in Q$

- (1) For each of the words w_i in the input sequence, select one of the formulas assigned to this word from the lexicon and select a $q \in Q$ as the conclusion.
- (2) Decompose the formulas according to the links of Table 7.8 on page 108 until we reach the atomic subformulas. The disjoint union of these proof structures is itself a proof structure, though it will have several hypotheses in addition to those from the lexicon and several conclusions in addition to the goal formula.
- (3) Identify each atomic premiss with an atomic conclusion to produce a proof structure with hypotheses $l(w_1), \dots, l(w_n)$ and conclusion q .
- (4) Convert the abstract proof structure corresponding to this proof structure to a hypothesis tree using only the structural conversions of \mathcal{R} and the contractions.
- (5) Check if this hypothesis tree conforms to our parameters P .

Table 8.1: Proof search algorithm for $\mathbf{NL} \diamond_{\mathcal{R}}$

Finally, parameter (iii) states that we are sometimes only interested in the shortest conversion sequence to a hypothesis tree. This should not be taken as a constraint on the derivability relation in the sense of ‘shortest move’ constraints proposed in minimalist frameworks (Chomsky 1995), but as a way of preferring conversion sequences without *redundant* structural conversions.

Throughout the next sections we will present some improvements over the initial, naive algorithm of Table 8.1. These improvements can be categorized as follows.

[Compilation] This is a standard programming technique where predictable computation steps are done in advance and the results stored or where we collapse several simple steps into a single derived step. In the context of declarative programming languages compilation is sometimes called *partial execution* (Pereira & Shieber 1987). We will apply partial execution to the current problem in Section 8.2, where we will store abstract proof structures in the lexicon, eliminating step 2 from the algorithm and in Section 8.6 where we compile multiple par

contractions into a single, derived contraction.

[Divide and Conquer] This refers to the basic technique of solving a problem by dividing it in several simpler problems in such a way that a solution to all these simple problems is a solution to the complete problem as well. We will use this strategy in Sections 8.5 and 8.9 where we will use components as a natural way of restricting structural rule applications.

[Early Failure] In constraint programming (Dechter 2000), it is often possible to get good performance on computationally intractable problems. This is done by strategies for realizing, as early as possible, that the current choices we have made will never lead us to a solution. While early failure may take the form of simple, deterministic tests, it can sometimes also consist of doing computations as early as possible. We will see examples of this in Sections 8.1, 8.3, 8.4 and 8.7.

[Parallel Computation] We will develop a way of performing the structural conversions in parallel in Section 8.8.

A general trade-off we will see is that we can sacrifice generality or completeness for efficiency. Some of the most powerful heuristics mentioned in this chapter function only for restricted fragments of $\text{NL}\diamond_{\mathcal{R}}$ and in the next chapter we will see that only quite restricted fragments of $\text{NL}\diamond_{\mathcal{R}}$ are decidable in polynomial time.

8.1 Invariants

As connecting the atomic formulas and the structural conversions are computationally expensive, it is desirable to do some static tests on the set of proof structures we get from the lexical formulas after the unfolding stage of the algorithm to make sure we at least have a chance of ultimately converting to a hypothesis tree. The following are two simple tests to reject proof structures which can never satisfy our correctness criterion.

First, by our definition of hypotheses and conclusions of proof structures, all atomic formulas other than lexical formulas or the conclusion must be both a premiss and a conclusion of some link in a proof structure with hypotheses $l(w_1), \dots, l(w_n)$ and conclusion q . So we can count if each of these atomic formulas occurs as many times as a conclusion as it occurs as a premiss. This is sometimes called the *count check* (van Benthem 1986).

Secondly, the following lemma, suggested to me by Quintijn Puite, gives us a condition on the number of binary links occurring in a proof net.

Lemma 8.3 *Suppose we have a proof structure S with h hypotheses, t binary tensor links, p binary par links and a single conclusion. Then the following holds if S is a proof net.*

$$t + 1 = p + h$$

Proof Reasoning backwards from the hypothesis tree to the initial hypothesis structure we see that it holds for the hypothesis tree (with $p = 0$), that the structural conversions and the unary contractions preserve t , p and h and that the contractions for the binary links increase t and p simultaneously. \square

We cannot use the same reasoning for unary connectives; though the contractions for the unary connectives remove one unary tensor and one unary par link, even in the case without structural rules we can state only that $p_1 \leq t_1$, where p_1 is the number of unary par links and t_1 the number of unary tensor links. This is because there can be an arbitrary number of unary tensor links in the final hypothesis tree. In the presence of structural rules, which possibly increase the number of unary tensor links, there is little we can tell simply from counting the unary links.

8.2 Compiling the Lexicon

Instead of have a lexicon which consists of formulas, we can compile the formulas of the lexicon to proof structures, which we can further compile to abstract proof structures, where we keep track of the output formulas of every abstract proof structure. We can denote this by using square brackets, for example. A formula between square brackets is then a ‘true’ hypothesis or conclusion of the abstract proof structure, we will call it a *bound* formula. The other formulas are atoms which will disappear after they are used for axiom connections, so they are ‘temporary’ hypotheses and conclusions, we will call them *free* formulas.

This is only necessary so we can distinguish between an atomic formula a used as a hypothesis and an atomic formula a used as a conclusion, the former looking like shown below on the left, the latter looking like shown below on the right.

$$\begin{array}{c} [a] \\ \cdot \\ a \end{array} \qquad \begin{array}{c} a \\ \cdot \\ [a] \end{array}$$

Alternatively, we can say that the hypotheses and conclusions of a proof structure or an abstract proof structure are the conclusion of ‘hypothesis’ links or the premiss of ‘conclusion’ links respectively.



In the depiction of the proof structures, this will have the advantage that all axiomatic formulas are now the active formula of two links, by symmetry with the cut formulas, which are the main formula of two links. It is often of mnemonic value to use the word w to which this lexical proof structure is

assigned instead of ‘Hyp’ as the label of of a hypothesis link. We’ll give an example of this in Section 8.4.

Identifying two vertices by means of an axiom connection is possible in an abstract proof structure if one has the formula F as a free premiss and the other has the formula F as a free conclusion. Both formulas will disappear after the axiom connection.

8.3 Acyclicity and Connectedness

If we translate formulas and antecedent terms of $\mathbf{NL}\diamond_{\mathcal{R}}$ to formulas and antecedent terms of multiplicative intuitionistic linear logic as follows

$$\begin{aligned} \|a\| &= a \\ \|\diamond_j A\| &= \|A\| \\ \|\square_j A\| &= \|A\| \\ \|A/_i B\| &= \|B\| \multimap \|A\| \\ \|B \setminus_i A\| &= \|B\| \multimap \|A\| \\ \|A \bullet_i B\| &= \|A\| \otimes \|B\| \\ \\ \|\langle \Gamma \rangle^j\| &= \|\Gamma\| \\ \|\Gamma \circ_i \Delta\| &= \|\Gamma\|, \|\Delta\| \end{aligned}$$

then every derivable sequent of $\mathbf{NL}\diamond_{\mathcal{R}}$ corresponds to a derivable sequent of **MILL**.

We define switchings and correction graphs for abstract proof structures of $\mathbf{NL}\diamond_{\mathcal{R}}$ analogous to the way we did for **L**-proof structures in Section 7.8. As we already noted in Section 4.5, we have linear time algorithms for checking whether all correction graphs of a proof structure are acyclic and connected. Because of this, it seems prudent to make an acyclicity and connectedness test before trying to convert the abstract proof structure to a tree.

In many cases, we can already see during the stage where we are connecting the axioms that, no matter how we continue, we will never produce an acyclic and connected abstract proof structure.

If an abstract proof structure \mathcal{A} has a substructure with a cyclic correction graph, then \mathcal{A} will have a cyclic correction graph too.

Similarly, if an abstract proof structure \mathcal{A} has a correction graph with disconnected substructures $\mathcal{A}_1, \dots, \mathcal{A}_n$ then every disconnected substructure must have a free formula at at least one of its vertices, otherwise it will be impossible to produce a connected correction graph for \mathcal{A} even after we perform further axiomatic connections.

8.4 Axiomatic Connections

The algorithm, as shown in Table 8.1 does not specify anything about the order in which we perform the axiomatic connections in step 3. From a logical point of view the order in which we connect the axioms does not matter, but

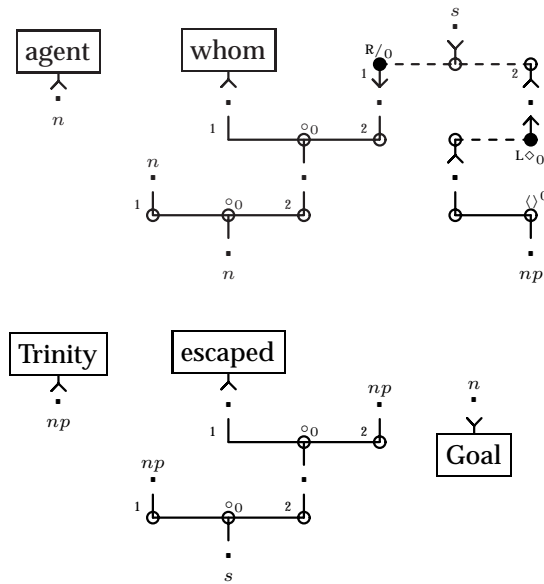


Figure 8.2: Lexical aps's for 'agent whom Trinity escaped'

from a computational point of view it is useful to keep the principles of early failure in mind and always connect the axiom which is the most restricted in its possibilities. This will make the information in the proof structure more explicit, which in turn can trigger other early failure mechanisms.

Let's look at an example. Figure 8.2 shows the lexical abstract proof structures for 'agent whom Trinity escaped' according to the lexicon of Table 7.7 on page 106.

We see one s conclusion and one s premiss, two np conclusions and two np premisses, and two n conclusions and two n premisses. In this case, there is only one possible way of connecting the s formulas, so this is the preferred connection, resulting in the abstract proof structure shown in Figure 8.3.

After this connection, some information which was implicit in Figure 8.2 has become explicit, for example that, unless some structural conversion operates on this abstract proof structure, the word 'whom' will precede the word 'escaped' in the final hypothesis tree. We'll see in Section 8.7 how to exploit this kind of word order information.

At the current stage, superficially, it doesn't matter if we decide to link the np 's or the n 's, because in both case we have to consider two possibilities. However, should we choose to link the n conclusion attached to 'agent' to the n premiss of the goal formula, we would produce a disconnected abstract proof structure. So, if we take the acyclicity and connectedness criterion discussed in Section 8.3 into account, we have only one way of connecting the four n formulas, namely as shown in Figure 8.4.

Finally, we connect the np formulas. We have two possibilities here, depending on where we connect the np conclusion corresponding to 'Trinity'

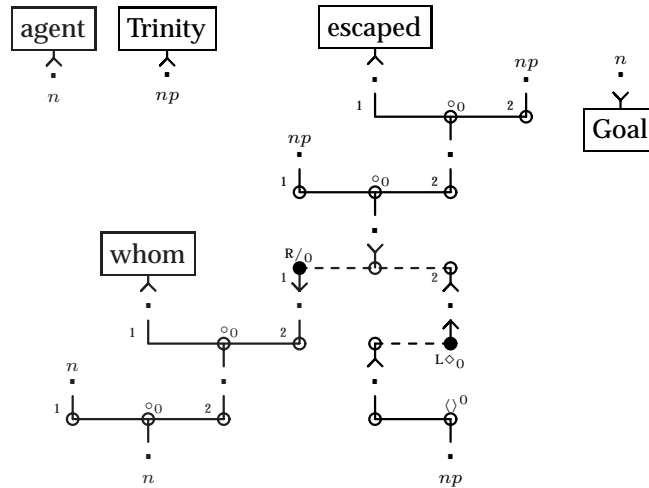


Figure 8.3: Abstract proof structure after the *s* connection

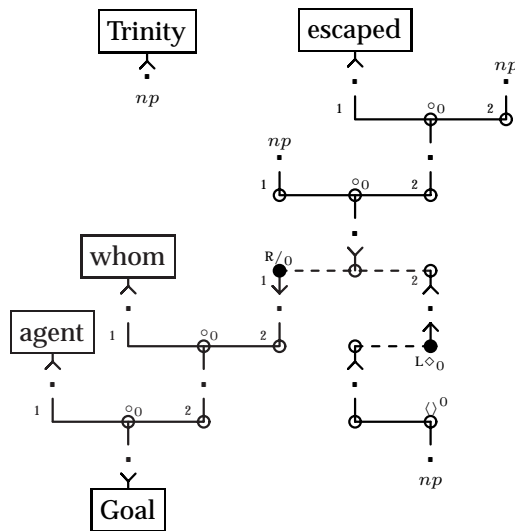


Figure 8.4: Abstract proof structure after the *n* connections

to the left or right premiss. In this case, only the first possibility allows the resulting abstract proof structure to convert to a hypothesis tree, as shown in Example 7.13

In the worst case, with *n* premisses and *n* conclusions, we might have to consider all *n!* different connections, but the strategy of first connecting the atomic formula with the smallest number of possible candidates for connection appears to be quite powerful.

Another strategy to perform the axiom links would be to perform them

incrementally from left to right, starting with the first word of the sentence and trying to make as many connections as possible after each new word. This has been independently proposed by Johnson (1998) and by Morrill (1998) (2000). Both authors present evidence that the number of unconnected atomic formulas in a proof structure corresponds to the relative difficulty a human would have when processing the sentence, giving some surprising psycholinguistic support for the use of proof nets in linguistic analysis. I want to avoid making any claims about the psychological reality of the current, opportunistic literal selection strategy. Connecting the operation of an automated theorem prover to psychological processes is, in my opinion, neither necessary nor desirable.

Finally, the axiom connections are related to strategies for resolution-based theorem provers. Eisinger & Ohlbach (1993) give a good overview of different literal selection strategies.

8.5 Components

Components, which we introduced in Section 7.6 to prove cut elimination, have some good properties we can use for automated deduction.

Recall from Definition 7.17 that a component of an abstract proof structure \mathcal{A} is a maximally connected substructure with respect to the tensor links of \mathcal{A} . Structural conversions operate on one component only, and leave all other components unchanged. Contractions operate on one component, in which they erase a tensor link, then merge this component with another.

Definition 8.4 *If a component \mathcal{C} of an abstract proof structure \mathcal{A} does not contain vertices which are the output vertex of a par link, but*

- (i) *either $\mathcal{C} = \mathcal{A}$, that is, there are no par links in \mathcal{A}*
- (ii) *or both input vertices of a par link are in \mathcal{C}*

we call the component active. Otherwise, we will call it waiting,

Similarly, if all input vertices of a par link are in the same component, we will call this link active. Otherwise, we will call it waiting.

Example 8.5 *The components in the abstract proof structure of Example 7.10 are drawn in black in Figure 8.5 on the following page. Note that the top component consists of only a single vertex.*

In this abstract proof structure there is only one active component, the middle one, and only the $[L\Diamond_0]$ link is active.

Lemma 8.6 *We can restrict ourselves to conversion sequences of the following form.*

- (1) *Apply a number of structural conversions in an active component.*
- (2) *If the abstract proof structure still has par links, contract an active par link of which the inputs are in the current component, reassess the active components and continue from 1.*

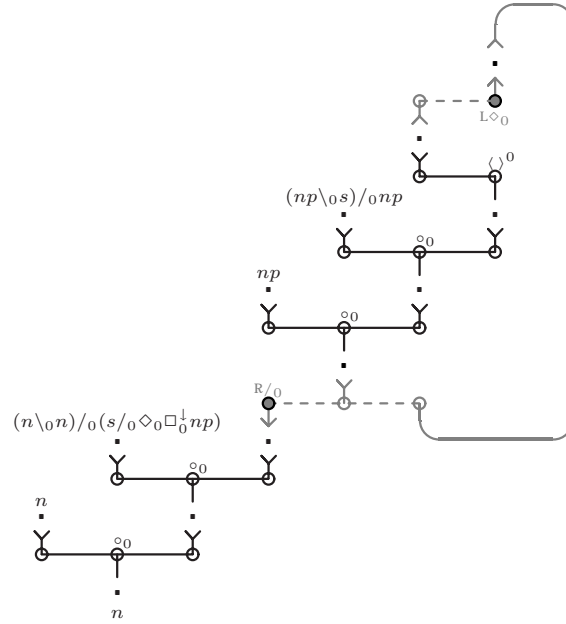


Figure 8.5: The components of Figure 7.9

Proof As noted before, the different components of an abstract proof structure are independent until they are united into one component by a contraction. As the contractions are defined to operate by contracting a par link of which all active vertices are connected to the same tensor link, this means the first contraction in any abstract proof structure must be a par link of which the active vertices are already in the same component.

Because we also prohibit the active component from containing the main vertex of a par link, no contractions other than the contraction of one of the active par links from this component will affect the current component. \square

Some remarks need to be made. When an abstract proof structure has multiple active components, we can apply structural rules to them independently; we can start with any active component we want or we can even operate on all active components in parallel.

However, when a component has multiple active par links, it is possible that we are in the following situation: if we contract one before the other, we can convert the abstract proof structure to a hypothesis tree, but if we contract them in the other order, conversion to a hypothesis tree may not be possible. Figure 8.6 on the next page presents two examples of such situations, which occur in the base logic $NL \diamond$.

On the left of the figure, both the $[L \diamond_0]$ and the $[R \square_0^\perp]$ link are active, but if we contract the $[L \diamond_0]$ link first, we will be unable to contract the $[R \square_0^\perp]$ link. On the other hand, if we contract the $[R \square_0^\perp]$ link first, we produce a redex for the $[L \diamond_0]$ contraction immediately.

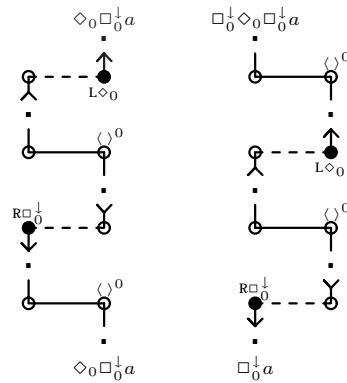


Figure 8.6: Conflicts between two active par links

On the right of the figure, we have the same active component, with the same active par links, only now we are in the opposite situation: contracting the $[L\Diamond_0]$ link is required before contracting the $[R\Box_0^{\perp}]$ link if we want to convert to a hypothesis tree.

Definition 8.7 *A component is completed if none of its vertices are labeled with free formulas.*

When a component is both completed and active, further axiomatic connections will not be relevant for that component, at least not until one of the active par links bordering it will be contracted, which may cause the new component to be incomplete again. Therefore, when we are performing the axiom connections, we have the possibility of performing conversions on active, completed components whenever we produce them. However, it may not necessarily be the best strategy to contract par links bordering completed components whenever we encounter them. We have to be careful which strategy we prefer: do we choose eager evaluation, that is contract par links as soon as the component it is attached to is completed, or lazy evaluation, that is wait with contracting par links as much as possible.

The advantage of eager evaluation appears to be that we can detect par links which are not contractable at an early stage, thereby triggering early failure mechanisms and possibly preventing unnecessary computations.

However, it is possible lazy evaluation gives better performance. Continuing connecting axioms may fail with respect to other early failure mechanisms which are computationally less expensive. Waiting may also produce other active, completed components which are smaller or more likely to fail. Finally, with respect to eager evaluation it is difficult to decide between multiple, active par links and making the wrong choice can lead to a dead end in the search space.

The Grail automated theorem prover discussed in Appendix A gives you the choice of performing eager or lazy evaluation of par links. Section A.3.9

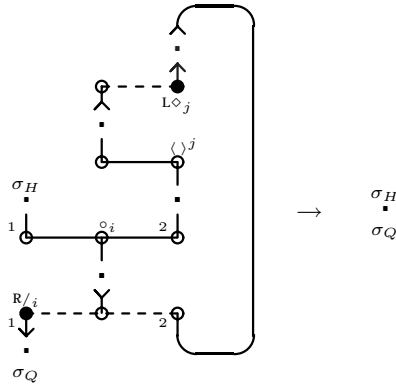


Figure 8.7: A combined $/_i \diamond_j$ contraction

gives details on how to set up these parameters.

8.6 Focusing

One of the insights of focusing proofs (Andreoli 2000) is that multiple par links which are connected in such a way that the active formula of one link is the main formula of the other, can be executed at the same time. We will call such par links *consecutive* par links. In Figure 8.5 on page 142 the $[L \diamond_0]$ and the $[R/_0]$ link are consecutive par links. We can treat these configurations as if they were a single logical operator. In the case of $'/_0 \diamond_0'$ its logical rules would be the following.

$$\frac{\Gamma[A] \vdash C \quad \Delta \vdash B}{\Gamma[A/_0 \diamond_0 B \circ_0 \langle \Delta \rangle^0] \vdash C} [L/_0 \diamond_0] \quad \frac{\Gamma \circ_0 \langle B \rangle^0}{\Gamma \vdash A/_0 \diamond_0 B} [R/_0 \diamond_0]$$

The contraction corresponding to this combination is shown in Figure 8.7.

A similar strategy is not possible with a tensor/par combination. For example, though we can add the following rules for $\diamond_0 \square_0^\perp$ to our sequent calculus

$$\frac{\Gamma[A] \vdash C}{\Gamma[\diamond_0 \square_0^\perp A] \vdash C} [L \diamond_0 \square_0^\perp] \quad \frac{\langle \Gamma \rangle^0 \vdash A}{\langle \Gamma \rangle^0 \vdash \diamond_0 \square_0^\perp A} [R \diamond_0 \square_0^\perp]$$

these rules will be *incomplete*, that is, there are valid derivations with formulas of the form $\diamond_0 \square_0^\perp A$ which you will not find using the combined rules above, but which you will find using a separate \diamond and \square^\perp rule. In the example above, applying the $[L \diamond_0]$ rule instead of the combined rule would possibly open up new structural rules. Example 7.4 on page 106 shows such a situation. If we look at the sequent rules of Table 7.5 on page 105, we see that from a forward chaining proof search perspective the tensor rules add

structural information, whereas the par rules remove it under certain conditions.

With respects to the components of an abstract proof structure, consecutive par links will correspond to components consisting of a single vertex, see for example Figure 8.5. Now, it is possible to extend the definition of active par links to active consecutive par links.

Definition 8.8 *Consecutive par links are active whenever all the inputs to the tree of par links are in the same component.*

Lemma 8.9 *We can restrict ourselves to conversion sequences of the following form.*

- (1) *Apply a number of structural conversions in an active component.*
- (2) *If the abstract proof structure still has par links, contract a tree of active consecutive par links of which all leaves are in the current component, reassess the active components and continue from 1.*

Proof According to Lemma 8.6 we can restrict ourselves to conversion sequences where we apply structural conversions in an active component, then contract an active par link attached to the current component. Now let C be an active component and L be an active par link which is part of an active consecutive par link. Performing this par contraction will remove a tensor link from the active component and if the par link was part of a consecutive par link, the component will now be attached to a single vertex. This means any structural conversion which is possible after this contraction was already possible before this contraction as well, since the new component is a proper substructure of the component just before the contraction. \square

8.7 Word Order

There are cases where we can see from the abstract proof structure we are constructing that it will never convert to an abstract proof structure where the words of the input sequence occur in the right order.

For example, the abstract L -proof structures we introduced in Section 7.8 have only a structural rule representing associativity, and no structural rules which could change the order of the hypotheses to the abstract proof structure. This means that whenever we make a axiomatic connection which does not respect the order of the words in the input sentence, we will never be able to contract the abstract L -proof structure to a single hypothesis comb. Also, when the second conclusion of $[R/]$ link is connected to anything but the last hypothesis of a comb, contracting it will never be possible. Similar arguments can be made for the other par links.

This property is a reflex in our proof net calculus of the planarity condition we discussed in Section 4.7. We call binary modes for which either no structural rules or one or both of the associativity structural rules apply *continuous*.

A way of enforcing planarity for some modes but not for others is by first order approximation. In Section 5.1.4 we gave embedding results for both **L** and **LP** into the first order multiplicative fragment of linear logic. By giving every continuous mode the **L** translation and every discontinuous mode the **LP** translation, we have a simple way of enforcing at least some of the word order constraints on derivation. We can also imagine giving certain modes a slightly more sophisticated translation, such as those suggested in Section 5.2.2 for relative pronouns. Morrill (1999) gives similar suggestions for using first order constructs to enforce word order constraints.

In some cases, we can also use an eager evaluation strategy for performing the structural conversions and put the words in the right order with respect to each other whenever possible; each time we perform an axiom connection between two disjoint abstract proof structures, we merge the words of the two abstract proof structures until the new aps has all words in the right order again. Again, eager evaluation can be dangerous because it can force us to select the wrong alternative or because it can be impossible to put two words in the right order until they are put in a bigger context.

The Grail automated theorem prover of Appendix A gives you the choice of how to evaluate the word order constraints, see Section A.3.9 for details on how to set up these parameters.

8.8 Parallel Computation

Though we have already seen that it is possible to apply structural rules in parallel in different active components, in this section we will see that if we represent a component appropriately, we can apply structural rules in parallel in the *same* component as well.

This is done simply by allowing every vertex to be the conclusion and the premiss of more than one link. We call such a structure a parallel abstract proof structure. This makes it a quite a bit harder to represent the abstract proof structures in an orderly way. I will choose to represent parallel aps's by just listing the links, in what I will call the distributed representation of a parallel aps, even though this will mean vertices can occur in multiple places. I think the alternative, having every vertex fixed and drawing the links between them, will lead to unnecessarily cluttered figures.

Example 8.10 *Let's look at the abstract proof structure corresponding to the sequent $a/a a, a/a a, a/a a \vdash a/a a$ shown in Figure 8.8 on the next page, where we assume mode a is associative but not commutative.*

When we present this abstract proof structure as a parallel aps, it will look as shown in Figure 8.9 on the facing page. The links are named L_1 to L_4 for future reference.

We perform structural conversions on parallel abstract proof structures as follows.

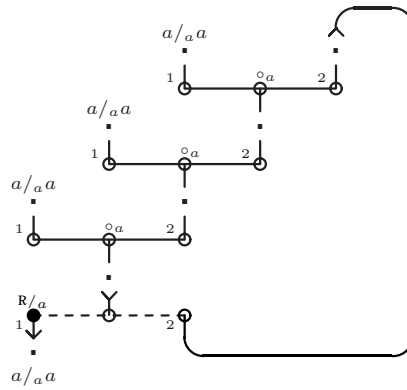


Figure 8.8: Abstract proof structure for $a/a a, a/a a, a/a a \vdash a/a a$

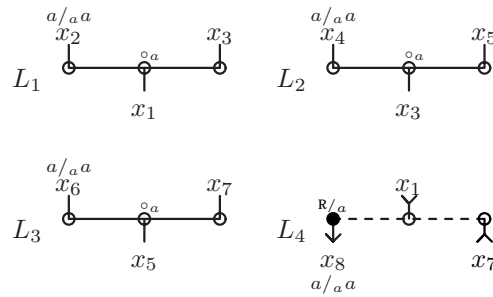


Figure 8.9: The aps of Figure 8.8 in a distributed representation

- find all sets of connected tensor links which are the redex of a structural conversions.
- using fresh internal vertices, add the reduct of the structural conversion to the parallel aps unless the reduct is equivalent, up to renaming of the internal vertices, with links which are already present in the parallel aps.

Example 8.11 Using one application of the associativity rule, we can expand the parallel aps of Figure 8.9 with the links shown in Figure 8.10 on the next page. From L_1 and L_2 associativity gives us L_5 and L_6 , and we can reassociate L_2 and L_3 into L_7 and L_8 .

For the next generation, we know that at least one of the links to which we apply a structural rule must have been introduced in the last generation, because this is the only way we can produce new links.

The third generation tensor links are shown in Figure 8.11 on the following page. L_9 and L_{10} have been obtained from L_3 and L_5 , whereas L_{11} and L_{12} have been obtained from L_1 and L_7 . The redexes formed by L_5 and L_6 and by L_7 and L_8 have not been triggered, because their reducts would be equivalent up to renaming of the

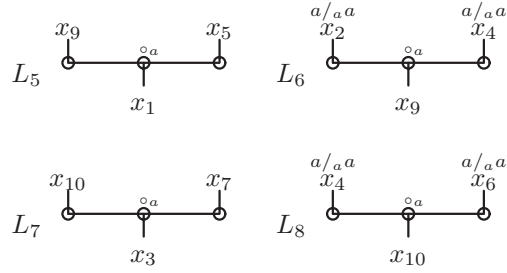


Figure 8.10: Second generation tensor links

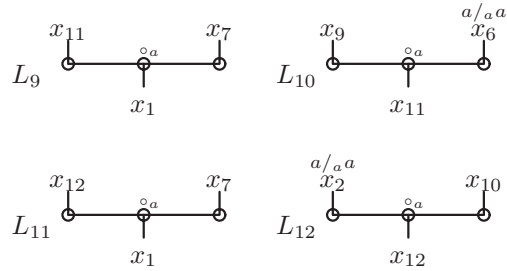
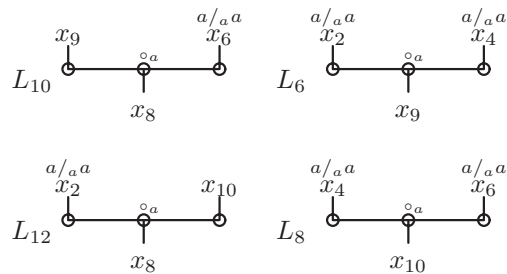


Figure 8.11: Third generation tensor links

internal nodes to L_1 and L_2 and to L_2 and L_3 respectively.

We have completed the structural rule applications: no structural conversion from the current state will produce new links. We are now in a position to apply the $[R/a]$ contraction, which will identify vertices x_8 , x_{11} and x_{12} and which will erase all links which have become unreachable from the root vertex x_8 . The resulting parallel aps is shown in Figure 8.12.

For the current example, the contraction system described in Section 7.8 is, of course, much more efficient. However, the setup described here is very general and works for any set of structural rules

Figure 8.12: Parallel aps after the $[L/a]$ contraction

The methodology outlined in this section is very close to finding a canonical model according to the Kripke semantics discussed in Section 3.6. The two frame constraints for an associative mode a would be the following.

$$\begin{aligned} \forall x_1.x_2.x_3.x_4.x_5. x_1R_ax_2x_3 \wedge x_3R_ax_4x_5 &\rightarrow \exists x_6. x_1R_ax_6x_5 \wedge x_6R_ax_2x_4 \\ \forall x_1.x_2.x_3.x_4.x_5. x_1R_ax_2x_5 \wedge x_2R_ax_3x_4 &\rightarrow \exists x_6. x_1R_ax_3x_6 \wedge x_6R_ax_4x_5 \end{aligned}$$

When we interpret xR_iyz as there is a tensor link of mode i with premisses y and z and conclusion z , all we do when applying the structural rules to a parallel aps is to add tensor links and vertices which must exist according to the frame constraints.

An interesting possibility to investigate would be to extend parallel computation beyond single components, possibly even extending it to compute different lexical assignments to words of the input sentence in parallel and compare the time and space complexity with the sequential version of Table 8.1.

8.9 Rule Filtering

In many cases we can see from the shape of the structural conversions that some structural conversions can never produce a redex for the contraction of the active par link we are looking at.

Example 8.12 *If we want to add features for person, number, gender and case to a simple English grammar, in the style of Heylen (1999), we might do this as shown in Table 8.2. The features on the lexical entry for ‘he’ show it is a 3rd person singular masculine nominative pronoun. For every feature we have a ‘top’ and a ‘bottom’ element. For example the gender feature has, in addition to values m for masculine, f for feminine and n for neuter a value g which includes any gender feature and a value G which is included by any gender feature, as stated by the following structural rules, where $i \in \{m, f, n\}$.*

$$\frac{\Gamma[\langle \Delta \rangle^G] \vdash C}{\Gamma[\langle \Delta \rangle^i] \vdash C} [i, G] \quad \frac{\Gamma[\langle \Delta \rangle^i] \vdash C}{\Gamma[\langle \Delta \rangle^g] \vdash C} [g, i]$$

In the lexicon of Table 8.2, ‘they’ is assigned gender feature G which means it can satisfy any gender requirement from the verb. On the other hand, ‘smiles’ selects for a subject with gender feature g which means any marking for gender will satisfy it. An example derivation of ‘Marla smiles’ is shown in Figure 8.13.

In the example above, we only have structural rules for inclusion and none for interaction. The abstract proof structure for this derivation, where, for the sake of simplicity, we abstract over the \square_3^\perp and \square_{sg}^\perp connectives, is shown in Figure 8.14.

$$\begin{aligned}
l(\mathbf{he}) &= \square_3^\downarrow \square_{sg}^\downarrow \square_m^\downarrow \square_{nom}^\downarrow np \\
l(\mathbf{she}) &= \square_3^\downarrow \square_{sg}^\downarrow \square_f^\downarrow \square_{nom}^\downarrow np \\
l(\mathbf{it}) &= \square_3^\downarrow \square_{sg}^\downarrow \square_n^\downarrow \square_C^\downarrow np \\
l(\mathbf{him}) &= \square_3^\downarrow \square_{sg}^\downarrow \square_m^\downarrow \square_{acc}^\downarrow np \\
l(\mathbf{her}) &= \square_3^\downarrow \square_{sg}^\downarrow \square_f^\downarrow \square_{acc}^\downarrow np \\
l(\mathbf{they}) &= \square_3^\downarrow \square_{pl}^\downarrow \square_G^\downarrow \square_{nom}^\downarrow np \\
l(\mathbf{them}) &= \square_3^\downarrow \square_{pl}^\downarrow \square_G^\downarrow \square_{acc}^\downarrow np \\
l(\mathbf{Marla}) &= \square_3^\downarrow \square_{sg}^\downarrow \square_f^\downarrow \square_C^\downarrow np \\
l(\mathbf{smiles}) &= \square_3^\downarrow \square_{sg}^\downarrow \square_g^\downarrow \square_{nom}^\downarrow np \setminus s \\
l(\mathbf{hates}) &= (\square_3^\downarrow \square_{sg}^\downarrow \square_g^\downarrow \square_{nom}^\downarrow np \setminus s) / \square_p^\downarrow \square_n^\downarrow \square_g^\downarrow \square_{acc}^\downarrow np
\end{aligned}$$

Table 8.2: A lexicon with feature information

$$\begin{array}{c}
\frac{}{np \vdash np} [Ax] \\
\frac{}{\langle \square_C^\downarrow np \rangle^C \vdash np} [L \square_C^\downarrow] \\
\frac{}{\langle \square_C^\downarrow np \rangle^{nom} \vdash np} [nom, C] \\
\frac{}{\square_C^\downarrow np \vdash \square_{nom}^\downarrow np} [R \square_{nom}^\downarrow] \\
\frac{}{\langle \square_f^\downarrow \square_C^\downarrow np \rangle^f \vdash \square_{nom}^\downarrow np} [L \square_f^\downarrow] \\
\frac{}{\langle \square_f^\downarrow \square_C^\downarrow np \rangle^g \vdash \square_{nom}^\downarrow np} [g, f] \\
\frac{}{\square_f^\downarrow \square_C^\downarrow np \vdash \square_g^\downarrow \square_{nom}^\downarrow np} [R \square_g^\downarrow] \\
\frac{}{\langle \square_{sg}^\downarrow \square_f^\downarrow \square_C^\downarrow \rangle^{sg} np \vdash \square_g^\downarrow \square_{nom}^\downarrow np} [L \square_{sg}^\downarrow] \\
\frac{}{\square_{sg}^\downarrow \square_f^\downarrow \square_C^\downarrow np \vdash \square_{sg}^\downarrow \square_g^\downarrow \square_{nom}^\downarrow np} [R \square_{sg}^\downarrow] \\
\frac{}{\langle \square_3^\downarrow \square_{sg}^\downarrow \square_f^\downarrow \square_C^\downarrow np \rangle^3 \vdash \square_{sg}^\downarrow \square_g^\downarrow \square_{nom}^\downarrow np} [L \square_3^\downarrow] \\
\frac{}{\square_3^\downarrow \square_{sg}^\downarrow \square_f^\downarrow \square_C^\downarrow np \vdash \square_3^\downarrow \square_{sg}^\downarrow \square_g^\downarrow \square_{nom}^\downarrow np} [R \square_3^\downarrow] \quad \frac{}{s \vdash s} [Ax] \\
\hline
\square_3^\downarrow \square_{sg}^\downarrow \square_f^\downarrow \square_C^\downarrow np \circ \square_3^\downarrow \square_{sg}^\downarrow \square_g^\downarrow \square_{nom}^\downarrow np \setminus s \vdash s \quad [L \setminus]
\end{array}$$

Figure 8.13: Derivation of ‘Marla smiles’ with feature information

Now it is clear that the $[R \square_{nom}^\downarrow]$ contraction does not depend in any way on what happens to the $[\langle \rangle^f]$ link; only the $[\langle \rangle^C]$ link is relevant to this contraction and there is only one rule which can produce a $[\langle \rangle^{nom}]$ link, namely the $[nom, C]$ conversion. Similarly, after the $[R \square_{nom}^\downarrow]$ contraction we only the $[g, f]$ rule can produce a redex for the $[R \square_g^\downarrow]$ contraction.

The general idea here is: given an active par link which is not a redex for the corresponding contraction, we look at which structural conversions could be the last conversion just before the contraction and then, recursively, we look at which conversions could have produced the redex for the previous conversion. This strategy is particularly effective when modes have only

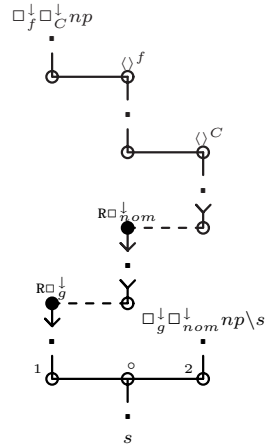


Figure 8.14: Abstract proof structure for ‘Marla smiles’

structural rules for inclusion, as in the example above, but other grammars can benefit from this strategy as well. If we look at Figure 8.8 on page 147 again, it is obvious that in order to produce a redex for a $[R/a]$ contraction we only need to use the $[Ass1]$ structural rule, whereas we would only need to use the $[Ass2]$ structural rule to produce a $[R\backslash_a]$ redex. Either structural rule could be the last conversion before a $[L\bullet_a]$ contraction, however.

8.10 Conclusions

We have seen several ways of improving the efficiency of the initial, naive algorithm by giving heuristics which are applicable in many cases. Though the complexity results from the next chapter make it unlikely that we will find an efficient algorithm for the general problem, it is possible to find algorithms which work reasonably well for large subclasses of the problem.

CHAPTER 9

COMPLEXITY

TABLE 9.1 gives an overview of the complexity results for various fragments of $\text{NL}\diamond_{\mathcal{R}}$. Even without contraction, weakening or similar structural rules where information is copied or deleted, the full logic $\text{NL}\diamond_{\mathcal{R}}$ is undecidable, as we will demonstrate in Section 9.3.

| | | |
|---|-------------|------------------------------------|
| – | Undecidable | $\text{NL}\diamond_{\mathcal{R}}$ |
| ⋮ | | |
| + | PSPACE | $\text{NL}\diamond_{\mathcal{R}-}$ |
| + | NP | MLL1, LP |
| + | ? | L |
| + | P | |
| + | n^6 | NL, LTAG |
| + | n^3 | AB |

Table 9.1: Complexity results for different fragments of $\text{NL}\diamond_{\mathcal{R}}$

In Section 9.2, however, we will show that if we disallow structural rules which remove unary connectives the resulting logic will be PSPACE complete. In Chapter 5 we already gave a number of examples of phenomena which could be described by the NP complete logic **MLL1** and related this logic to fragments of the multimodal Lambek calculus. Other NP complete

systems include **LP**, as shown by Kanovich (1991), and **L** extended with a commutative version of the right implication rule, as shown by Dörre (1996). Emms (1993a) shows the recognizing capacity of this logic is beyond that of context free grammars..

De Groote & Lamarche (2001) prove that, given a bracketed input, the decision problem for **NL** can be solved in $O(n^6)$ time and in the next chapter I will present a fragment of $\mathbf{NL} \diamond_{\mathcal{R}}$ with a restriction on the allowed formulas and fixed set of structural rules for which we can use the $O(n^6)$ parsing algorithms which have been proposed for LTAGs.

Finally, there are simple **AB** grammars, precursors to the Lambek calculus introduced by Ajdukiewicz (1935) and Bar-Hillel (1964), which can be seen as the Lambek calculus with only the $[L/]$, $[L\setminus]$, $[Ax]$ and $[Cut]$ rules. **AB** grammars have been shown to be weakly equivalent to context free grammars by Bar-Hillel, Gaifman & Shamir (1964), and can therefore be compiled into context free grammars after which we can use any of the standard $O(n^3)$ algorithms which exist for context free grammars.

Some problems still remain open: while it is relatively simple to show that the associative Lambek calculus is in NP, for example by the translation function of Section 5.1.4, it is still unknown if **L** is NP complete or if there exists a polynomial algorithm for deciding provability. Though Pentus (1995) (1997) has shown **L** grammars to be weakly equivalent to context free grammars, his proposed translation is exponential. Only some polynomial results for fragments of **L** have been found: Aarts (1994) shows that we can use polynomial parsing for the Lambek calculus with restricted formula complexity, whereas de Groote (1999b) gives an exponential algorithm based on proof nets and tabulation which performs in polynomial time for a subset of the complete logic, but so far the exact complexity for the full logic has remained elusive.

9.1 NP Complete Fragments

It seems quite reasonable to demand that we can determine the existence of a conversion sequence in polynomial time. In other words, we would like to restrict ourselves to packages of structural rules for which we have a polynomial time algorithm for finding a conversion sequence ending in a hypothesis tree, if such a conversion sequence exists.

Providing such an algorithm will immediately show the corresponding instance of $\mathbf{NL} \diamond_{\mathcal{R}}$ is in NP, and, if the logic contains some form of commutativity, even NP complete.

In Section 4.5 we have already given an $O(n^2)$ contraction algorithm for **MLL**, whereas in Section 7.8 we provided a $O(n^2)$ contraction algorithm for **L**, though in the case of **L** this merely shows **L** to be in NP and does not imply NP completeness of the problem.

One possibility for investigating fragments of $\mathbf{NL} \diamond_{\mathcal{R}}$ which are inherently in NP would be to provide specialized versions of the contraction criterion, along the lines of the contraction criterion for the Lambek calculus.

We will not investigate this possibility here, but in the next section we will explore a PSPACE complete formulation of $NL \diamond_{\mathcal{R}}$.

9.2 Restricted Structural Rules

In the previous section we saw how providing a polynomial conversion algorithm would give us a logic which is in NP, or even NP complete if it has some form of commutativity. Table 8.1, however, gives the set of structural rules as an input to the algorithm, so it would be nice if we could see, just from the form of the structural rules what the computational complexity of the logic defined by these structural rules would be. As we will see in the next section, it might be that the logic is undecidable. In this section we will present a way of identifying logics which are decidable in PSPACE by simple, local properties of their structural rules.

A natural restriction on the structural rules is to require that the left hand side of a structural conversion has at least as many unary connectives as the right hand side. We will call these postulates *non-expanding* as, looking at these postulates from the perspective of forward chaining proof search, the structural rules cannot increase the number of symbols during the course of a derivation.

Definition 9.1 *Given an antecedent configuration Ξ , we define the length of Ξ as follows.*

$$\begin{aligned} \text{length}(\Delta_1 \circ_i \Delta_2) &= \text{length}(\Delta_1) + \text{length}(\Delta_2) + 2 \\ \text{length}(\langle \Delta \rangle^i) &= \text{length}(\Delta) + 1 \\ \text{length}(\Delta) &= 0 \end{aligned}$$

We can extend Definition 9.1 to include arbitrary n -ary configurations by noting that the general form is the following.

$$\text{length}(*(\Delta_1, \dots, \Delta_n)) = \text{length}(\Delta_1) + \dots + \text{length}(\Delta_n) + n$$

Definition 9.2 *The logic $NL \diamond_{\mathcal{R}-}$ is the logic $NL \diamond_{\mathcal{R}}$ where for every structural rule $R \in \mathcal{R}$*

$$\frac{\Gamma[\Xi'[\Delta_1, \dots, \Delta_n]] \vdash C}{\Gamma[\Xi[\Delta_{\pi_1}, \dots, \Delta_{\pi_n}]] \vdash C} [R]$$

the following holds.

$$\text{length}(\Xi[\Delta_{\pi_1}, \dots, \Delta_{\pi_n}]) \leq \text{length}(\Xi'[\Delta_1, \dots, \Delta_n])$$

We call structural rules with this property non-expanding.

Note that because every structural rule of \mathcal{R} is linear, Definition 3.4 requires Ξ' to be non-empty, or, equivalently, that $\text{length}(\Xi') > 0$.

The logic $\mathbf{NL}_{\diamond_{\mathcal{R}}}$ still gives us access to a wide variety of structural postulates, including postulates like $[K]$

$$\frac{\Gamma[\langle \Delta_1 \rangle^1 \circ_0 \langle \Delta_2 \rangle^1] \vdash C}{\Gamma[\langle \Delta_1 \circ_0 \Delta_2 \rangle^1] \vdash C} [K]$$

even though its inverse $[K']$ is not allowed.

$$\frac{\Gamma[\langle \Delta_1 \circ_0 \Delta_2 \rangle^1] \vdash C}{\Gamma[\langle \Delta_1 \rangle^1 \circ_0 \langle \Delta_2 \rangle^1] \vdash C} [K']$$

All the structural rules we have seen so far satisfy this condition, with the exception of the $[4]$ postulate.

$$\frac{\Gamma[\langle \Delta \rangle^1] \vdash C}{\Gamma[\langle \langle \Delta \rangle^1 \rangle^1] \vdash C} [4]$$

However, the linguistic applications of this postulate appear to be limited. Moortgat (1997) even shows we can simulate the $[T]$ and $[4]$ postulates in \mathbf{NL}_{\diamond} without using structural rules.

Of the postulates which have been proposed in the literature, only two postulates proposed by Moortgat (1996a), repeated below, violate this condition because the conclusion of both rules contains an extra unary structural connective.

$$\frac{\Gamma[(\Delta_1 \circ_1 \Delta_2) \circ_0 \Delta_3] \vdash C}{\Gamma[\Delta_1 \circ_1 \langle \Delta_2 \circ_0 \Delta_3 \rangle^1] \vdash C} [P1'] \quad \frac{\Gamma[\Delta_2 \circ_0 (\Delta_1 \circ_1 \Delta_3)] \vdash C}{\Gamma[\Delta_1 \circ_1 \langle \Delta_2 \circ_0 \Delta_3 \rangle^r] \vdash C} [P2']$$

We will prove that the languages generated by $\mathbf{NL}_{\diamond_{\mathcal{R}}}$ are equivalent to the context sensitive languages, which are defined as follows.

Definition 9.3 A type 1 or context sensitive grammar G is a tuple $\langle \Sigma, N, S, R \rangle$, where

Σ is the set of terminal symbols,

N is the set of nonterminal symbols,

S is a designated member of N called the start symbol,

R is the set of grammar rules.

We require that N and Σ are disjoint. The set R consists of rewrite rules $\Gamma \rightarrow \Delta$, where Γ and Δ are lists of symbols from $N \cup \Sigma$. We restrict R to rules where Γ contains at least one nonterminal symbol and where $\text{length}(\Gamma) \leq \text{length}(\Delta)$. Specifically, this excludes rules of the form $\Gamma \rightarrow \epsilon$, where ϵ is the empty list.

A type 1 or context sensitive language \mathcal{L}_G is the set of lists of terminal symbols obtained by taking a type 1 grammar G and computing the closure of the start symbol S under the operation of the rules R .

Example 9.4 Context sensitive grammars are a quite expressive formalism. It is possible, for example, to formulate a grammar generating the following language.

$$\{a^n \mid n \text{ is prime}\}$$

A more simple example is the grammar which generates the following language.

$$\{a^n b^n c^n \mid n > 0\}$$

This grammar looks as follows: $\langle \{a, b, c\}, \{S, B, C\}, S, R \rangle$ where R is the following set of rules.

$$\begin{aligned} S &\rightarrow_1 aSBC \\ S &\rightarrow_2 aBC \\ CB &\rightarrow_3 BC \\ aB &\rightarrow_4 ab \\ bB &\rightarrow_5 bb \\ bC &\rightarrow_6 bc \\ cC &\rightarrow_7 cc \end{aligned}$$

This grammar generates the language $a^n b^n c^n$ for all $n > 0$. We can generate the string $aabbcc$, for example, as follows.

$$\begin{aligned} S &\rightarrow_1 \\ aSBC &\rightarrow_2 \\ aaBCBC &\rightarrow_3 \\ aaBBCC &\rightarrow_4 \\ aabBCC &\rightarrow_5 \\ aabbCC &\rightarrow_6 \\ aabbcC &\rightarrow_7 \\ aabbcc & \end{aligned}$$

To facilitate later proofs, we introduce the notion of lexicalization for phrase structure grammars. We formulate the definitions and lemma's in such a way that the don't use the context sensitive restriction on the rules of the grammar, so that we can reuse these results in Section 9.3, where we will apply the same definitions to unrestricted phrase structure grammars.

Definition 9.5 We say a phrase structure grammar is lexicalized if every rule is of one of the following forms.

- (i) $\Gamma \rightarrow \Delta$ where Γ and Δ are lists of nonterminal symbols
- (ii) $A \rightarrow \beta$ where $A \in N$ and $\beta \in \Sigma$.

We call the rules of form (i) the grammar rules and the rules of form (ii) the lexicalization rules.

Lemma 9.6 For lexicalized grammars we can restrict ourselves to derivations where all applications of grammar rules precede the applications of lexicalization rules.

Proof We prove the lemma by induction on the number of lexicalization rules which occur before grammar rules. Let \mathcal{D} be a derivation of a lexicalized grammar and $A \rightarrow \beta$ be a lexicalization rule which occurs before some grammar rules, that is, we are in the following situation

$$S \rightarrow_a \Gamma[A] \rightarrow \Gamma[\beta] \rightarrow_b \Delta[\beta]$$

where at least one of the rules in \rightarrow_b is a grammar rule. Because of the restrictions on the rules, none of the rules in \rightarrow_b can rewrite β , so it is easy to see the following derivation is valid too.

$$S \rightarrow_a \Gamma[A] \rightarrow_b \Delta[A] \rightarrow \Delta[\beta]$$

Because this derivation has one less lexicalization rule occurring before grammar rules we can apply the induction hypothesis giving us a derivation of the required form. \square

Lemma 9.7 *For every phrase structure grammar G there is a lexicalized grammar G' which generates the same language.*

Proof Take any rule ρ of the form $\Gamma \rightarrow \Delta$ from grammar G which is not one of the forms of Definition 9.5.

Let β_1, \dots, β_n be the terminal symbols occurring in ρ . Replace all occurrences of β_i in the grammar by a new nonterminal symbol N_i . This replacement does not introduce any new non-lexicalized rules: any rule previously corresponding to Definition 9.5.(ii) now corresponds to Definition 9.5.(i) and rules corresponding to Definition 9.5.(i) are unaffected. Finally, add a rule $N_i \rightarrow \beta_i$ to the grammar for every i . All these new rules correspond to Definition 9.5.(ii) and rule ρ now corresponds to Definition 9.5.(i). Call this new grammar G' .

It is clear that in G' the nonterminal symbols N_1, \dots, N_n play the role of the terminal symbols β_1, \dots, β_n , that is, any derivation which generates a terminal symbol β_i by means of a rule in G which is not a lexicalization rule, generates a nonterminal symbol N_i in G' . A separate lexicalization rule then rewrites N_i to β_i in G' . Conversely, any rule in G' which produces a nonterminal symbol N_i which is not a nonterminal symbol of G produces a terminal symbol β_i in G . \square

Example 9.8 *The context sensitive grammar from Example 9.4 is not lexicalized. Only rule 3 is a valid grammar rule in a lexicalized context sensitive grammar. But replacing a by A , b by D and c by E and adding the corresponding lexicalization rules produces a grammar in the form shown below, which is a lexicalized context sensitive grammar.*

$S \rightarrow_1 ASBC$
 $S \rightarrow_2 ABC$
 $CB \rightarrow_3 BC$
 $AB \rightarrow_4 AD$
 $DB \rightarrow_5 DD$
 $DC \rightarrow_6 DE$
 $EC \rightarrow_7 EE$
 $A \rightarrow_8 a$
 $D \rightarrow_9 b$
 $E \rightarrow_{10} c$

The derivation of Example 9.4 proceeds as shown below.

$S \rightarrow_1$
 $ASBC \rightarrow_2$
 $AABCBC \rightarrow_3$
 $AABBCC \rightarrow_4$
 $AADBCC \rightarrow_5$
 $AADDCC \rightarrow_6$
 $AADDEC \rightarrow_7$
 $AADDEE \rightarrow_8$
 $aADDEE \rightarrow_8$
 $aaDDEE \rightarrow_9$
 $aabDEE \rightarrow_9$
 $aabbEE \rightarrow_{10}$
 $aabbcE \rightarrow_{10}$
 $aabbc$

Definition 9.9 From a lexicalized context sensitive grammar G we generate the corresponding multimodal Lambek calculus $\mathcal{M}(G)$ as follows. $\mathcal{M}(G)$ has one unary mode for every nonterminal of G and a single binary mode.

Every lexicalization rule $A \rightarrow \beta$ corresponds to a lexical entry as follows.

$$\text{lex}(\beta) = a \setminus \square_A^\perp a$$

The goal formula of $\mathcal{M}(G)$ is $a \setminus \square_S^\perp a$ where S is the mode corresponding to the start symbol of G .

$\mathcal{M}(G)$ has a structural rule for every grammar rule $A_1 \dots A_n \rightarrow_{R1} B_1 \dots B_m$ of G .

$$\frac{\Gamma[\langle \dots \langle \Delta \rangle^{B_m} \dots \rangle^{B_1}] \vdash C}{\Gamma[\langle \dots \langle \Delta \rangle^{A_n} \dots \rangle^{A_1}] \vdash C} [R1]$$

Because $m > 0$ and $n \leq m$, this is a valid $\mathbf{NL}\diamond_{\mathcal{R}^-}$ rule.

Furthermore, the structural rule component of $\mathcal{M}(G)$ contains one of the structural rules for associativity for the single binary mode

$$\frac{\Gamma[(\Delta_1 \circ \Delta_2) \circ \Delta_3] \vdash C}{\Gamma[\Delta_1 \circ (\Delta_2 \circ \Delta_3)] \vdash C} [\text{Ass2}]$$

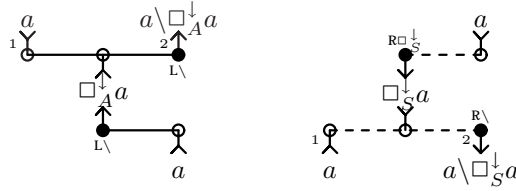


Figure 9.1: Proof structures for the lexicon and the goal formula

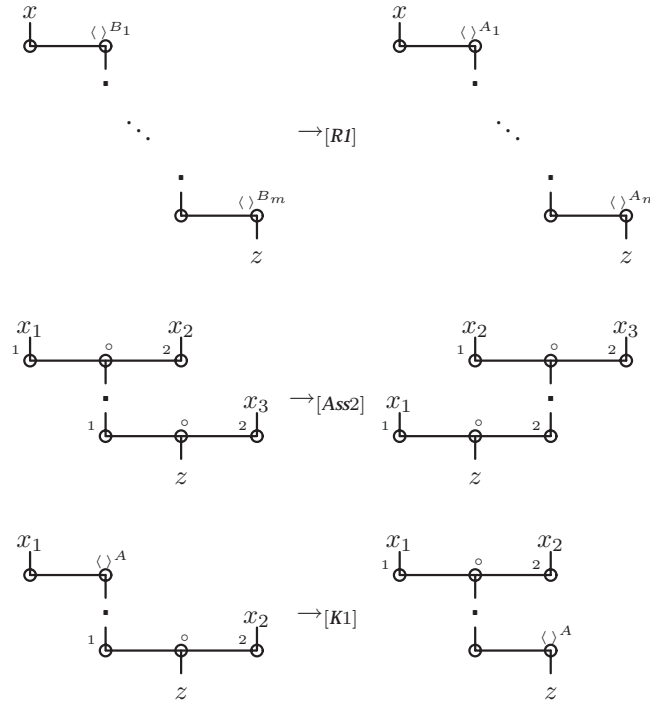


Figure 9.2: Structural conversions for $\mathcal{M}(G)$

and the structural rule of [K1] for every mode $A \in N$.

$$\frac{\Gamma[\langle \Delta_1 \rangle^A \circ \Delta_2] \vdash C}{\Gamma[\langle \Delta_1 \circ \Delta_2 \rangle^A] \vdash C} \text{ [K1]}$$

Seeing this grammar from the proof structure perspective, all lexical entries are of the form shown in Figure 9.1 on the left, where A is a unary index of $\mathcal{M}(G)$. The goal proof structure is shown in Figure 9.1 on the right. Figure 9.2 gives a schematic representation of the structural conversions.

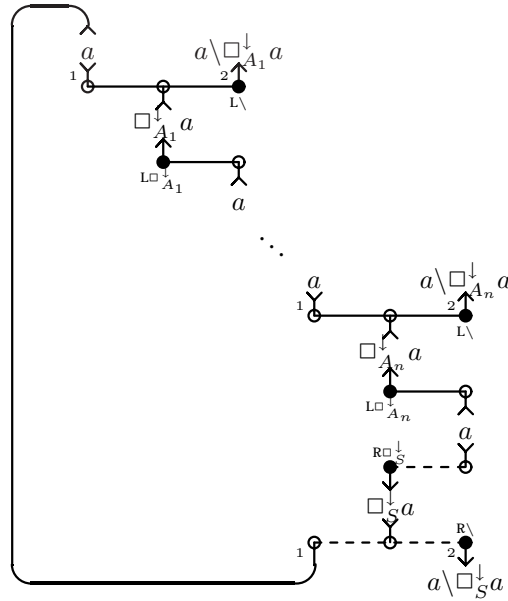


Figure 9.3: Proof structure for a derivation in $\mathcal{M}(G)$

Lemma 9.10 *Let $\mathcal{M}(G)$ be a multimodal Lambek calculus which corresponds to a context sensitive grammar G according to Definition 9.9. For every sequence of hypotheses H_1, \dots, H_n which correspond to lexical entries of $\mathcal{M}(G)$ only the proof structure S , which is shown schematically in Figure 9.3, can convert to a hypothesis tree of $H_1, \dots, H_n \vdash a \ \square \downarrow_S a$.*

Proof Look at the structural rules of $\mathcal{M}(G)$: none of them change the order of the hypotheses, so we need to consider only those proof structures where the hypotheses are in the right order with respect to each other. All hypotheses induce proof structures as shown in Figure 9.1 on the left, whereas the goal formula induces the proof structure shown in Figure 9.1 on the right.

Suppose we connect the a hypothesis of the goal proof structure to any formula other than the a conclusion of the rightmost lexical proof structure. Linking it to its own conclusion produces a disconnected proof structure, whereas linking it to a different lexical proof structure will make the formula corresponding to that proof structure appear as the rightmost hypothesis. We follow this line of reasoning inductively for the next lexical proof structures until after the last lexical proof structure has been connected to its a conclusion, we connect its a hypothesis to the a conclusion of the goal proof structure, producing the proof structure pictured in Figure 9.3. \square

Definition 9.11 *We define a function f_1 which translates a component of an abstract proof structure into a list of nonterminal symbols as follows. ‘||’ is the con-*

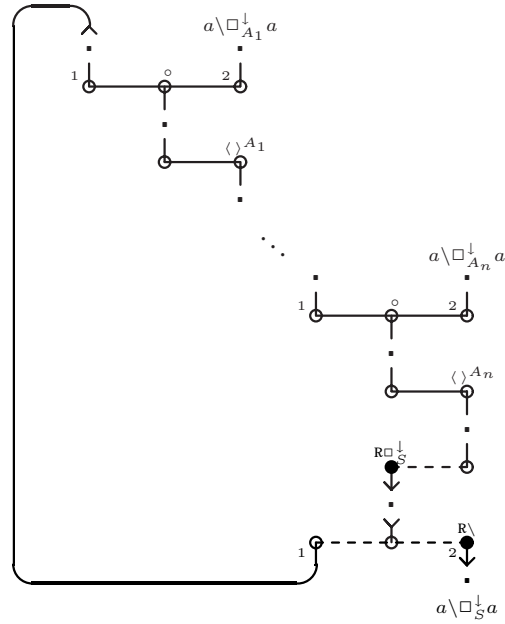


Figure 9.4: Abstract proof structure for a derivation in $\mathcal{M}(G)$

catenation operator.

$$\begin{aligned} f_1(F) &= \epsilon \\ f_1(\langle \Gamma \rangle^i) &= f_1(\Gamma) \parallel i \\ f_1(\Gamma_1 \circ \Gamma_2) &= f_1(\Gamma_1) \parallel f_1(\Gamma_2) \end{aligned}$$

Lemma 9.12 *Let $\mathcal{M}(G)$ be a multimodal Lambek calculus which corresponds to a context sensitive grammar G according to Definition 9.9. For every conversion sequence ρ of a proof net for $\mathcal{M}(G)$, applying function f_1 to the unique active component \mathcal{C} of the successive abstract proof structures of ρ before the two contractions of ρ and removing all identity transitions yields a derivation \mathcal{D} of G .*

Proof Because ρ ends in a hypothesis tree, we can apply Lemma 9.10, and we know \mathcal{S} has to be of the form shown in Figure 9.3 on the preceding page. Converting that proof structure into an abstract proof structure produces the result shown in Figure 9.4.

Because we have only one active component and a single active consecutive par link in it, we can apply Lemma 8.9 to ensure the two contractions can always be performed after any structural conversions in this component.

We will generate a derivation \mathcal{D} of Γ from ρ as follows. For the active component \mathcal{C} of the initial abstract proof structure $f_1(\mathcal{C}) = A_1 \dots A_n$. We extend this by applying the n appropriate lexicalization rules of G to produce a valid suffix of a derivation of $w_1 \dots w_n$ in G .

We now proceed by induction on the length l of the sequence of the structural conversions ρ , simultaneously proving that any time we compute $f_1(\Gamma_2)$ for the final clause of Definition 9.11 the result will be the empty list. For the abstract proof structure of Figure 9.4 this property holds, because the right branch of every $[\circ]$ link is a leaf.

If $l = 0$ then, because the consecutive ' \square_S^\perp ' par link can be contracted $\mathcal{C} = \langle a \circ \Gamma \rangle^S$, and because we have proved by induction that $f_1(\Gamma) = \epsilon$ this means $f_1(\langle a \circ \Gamma \rangle^S)$ is S . Therefore, we have produced a valid derivation in G .

If $l > 0$ we look at the last conversion.

If it is an [Ass2] conversion, we keep the suffix of the derivation in G the same. $f_1(\Gamma_1 \circ (\Gamma_2 \circ \Gamma_3)) = f_1((\Gamma_1 \circ \Gamma_2) \circ \Gamma_3) = f_1(\Gamma_1) \| f_1(\Gamma_2) \| f_1(\Gamma_3)$ and, given that $f_1(\Gamma_2 \circ \Gamma_3)$ is ϵ , $f_1(\Gamma_2)$ and $f_1(\Gamma_3)$ are both ϵ .

If the last conversion is a [K1] conversions, we keep the suffix of the G derivation the same. $f_1(\langle \Gamma_1 \rangle^A \circ \Gamma_2) = f_1(\Gamma_1) \| A \| f_1(\Gamma_2)$ whereas $f_1(\langle \Gamma_1 \circ \Gamma_2 \rangle^A) = f_1(\Gamma_1) \| f_1(\Gamma_2) \| A$, but since we know $f_1(\Gamma_2) = \epsilon$ both are equivalent.

If the last conversion is a grammatical conversion, we prefix the corresponding grammatical rule to the derivation of G . The result is again a valid suffix of a derivation in G . \square

Lemma 9.13 *Given a lexicalized context sensitive grammar G , the multimodal Lambek calculus $\mathcal{M}(G)$ corresponding to it according to Definition 9.9 generates the same language.*

Proof We need to show that the multimodal Lambek calculus $\mathcal{M}(G)$ corresponding to G by Definition 9.9 generates the same language as G , that is that there exists a derivation δ of string s in G iff there exists a derivation ρ of s in $\mathcal{M}(G)$.

$[\Rightarrow]$ Let δ be a derivation in G . Lemma 9.6 ensures there is a derivation δ' generating the same string where every grammar rule precedes every lexicalization rule. We construct the proof structure \mathcal{S} for the proof net derivation corresponding to δ by looking at the lexicalization steps of δ' and selecting the appropriate lexical entries corresponding to it according to the translation, then combining the lexical proof structures and the goal proof structure according to Lemma 9.10.

The abstract proof structure \mathcal{A} corresponding to \mathcal{S} , which looks as shown in Figure 9.4, can be converted as follows: for every grammatical rule of δ' we apply the conversion corresponding to it according to Definition 9.9 to \mathcal{A} . We start by combining all $[\langle \rangle^i]$ links using [K1] and [Ass2] rules alternately until we have a single sequence of $[\langle \rangle^i]$ links which, from hypotheses to conclusion, corresponds to the sequence of terminals just after the last grammatical rule of δ' . Now for every grammatical rule of δ' , starting with the last one, we apply the corresponding structural rule. At every step, the sequence of $[\langle \rangle^i]$ links will have modes B_1, \dots, B_m corresponding to the sequence of nonterminals of δ' , until, before the first grammatical rule, it will contain a single

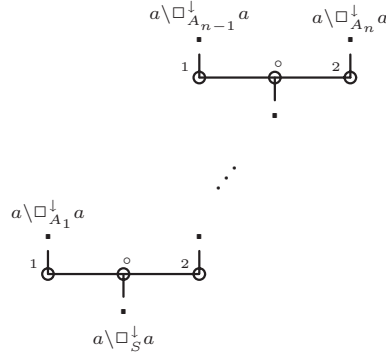


Figure 9.5: The final hypothesis tree of a $\mathcal{M}(G)$ proof net

$[(\rangle^S]$ link, which we can contract using the $[R\square_S^{\perp}]$ contraction. After the $[R\setminus]$ contraction the hypothesis tree will look as shown in Figure 9.5.

$[\Leftarrow]$ This is a direct consequence of Lemma 9.12 which shows how to generate a derivation of G from a conversion sequence of a proof net in $\mathcal{M}(G)$. \square

Lemma 9.14 *A nondeterministic linear bounded Turing machine can encode sequent proof search for any non-expanding multimodal Lambek calculus.*

Proof First, we replace any word which has multiple lexical assignments A_1, \dots, A_m by a single lexical assignment $A_1 \& \dots \& A_m$ and multiple possible goal formulas G_1, \dots, G_n by a single goal formula $G_1 \oplus \dots \oplus G_n$, adding the following logical ruled to the calculus.

$$\frac{\Gamma[A_i] \vdash C}{\Gamma[A_0 \& A_1] \vdash C} [L\&] \quad \frac{\Gamma \vdash C_i}{\Gamma \vdash C_0 \oplus C_1} [R\oplus]$$

Note that these are just the $[L\&]$ and the $[R\oplus]$ rules from linear logic adapted to $\mathbf{NL}_{\diamond \mathcal{R}}$. This addition to the logic is quite inessential and serves only as a way to generate the right amount of space on the input tape of the Turing machine in the case of multiple lexical assignments to a single word or multiple possible goal formulas.

The input tape of the Turing machine for the sequence of words $w_1 \dots w_n$ consists of the lexical formulas $f_1 \dots f_n$ corresponding to these words and the goal formula g , in Polish prefix notation, the different antecedent formulas separated by an arbitrary binary mode \circ_0

$$\vdash \circ_0 f_1 \circ_0 \dots \circ_0 f_n g$$

Now look at the maximum amount of space required for forward chaining proof search for the sequent calculus compared to the size of the input

tape. Forward chaining proof search amounts to starting with the axioms and applying the sequent rules from premisses to conclusion performing all computations nondeterministically and halting with success when we have generated the formulas on the input tape in the right order and with failure when we have exhausted the search space.

Because of the subformula property, the sequents we have to consider only contain subformulas of the end-sequent. We also need a binary sequent separator symbol, for which we use ‘ \bullet ’ to separate the different sequent proofs.

In Polish prefix notation, the sequent rules of $[R\Diamond_j]$ and $[R\bullet_i]$, normally portrayed as follows

$$\frac{\Gamma \vdash C}{\langle \Gamma \rangle^j \vdash \Diamond_j C} [R\Diamond_j] \quad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma \circ_i \Delta \vdash A \bullet_i B} [R\bullet_i]$$

look like shown below.

$$\frac{\vdash \Gamma C}{\vdash \langle \rangle^j \Gamma \Diamond_j C} \quad \frac{\Delta \vdash \Gamma A \vdash \Delta B}{\vdash \circ_i \Gamma \Delta \bullet_i A B}$$

Counting the number of symbols in the premisses and the conclusion, we see that these rules, like the other tensor rules, increase the number of symbols, whereas the par rules keep the number of symbols constant. Furthermore, because all structural rules are non-expanding they may reduce the total amount of space needed, but they can never enlarge it.

We examine the tensor rules more closely. For the $[R\Diamond_j]$ and the $[L\Box_j^\perp]$ rule the conclusion has four symbols more than the premiss (we count $\langle \rangle$ and \Box^\perp as one symbol), for the $[R\bullet_i]$, $[L/i]$ and $[L\setminus_i]$ rule there are two more symbols in the conclusion. Of these extra symbols, two are included on the input tape, which means that only for the $[R\Diamond_j]$ and the $[L\Box_j^\perp]$ we have to add two extra spaces on the tape. This means that the maximum amount of space we need is the length of the input tape l plus two extra symbols for every positive occurrence of \Diamond_j and two extra symbols for every negative occurrence of \Box_j^\perp , which is strictly less than $2l$. \square

Lemma 9.15 (Koroda (1964)) *Any linear bounded Turing machine which does not generate the empty string is equivalent to a context sensitive grammar.*

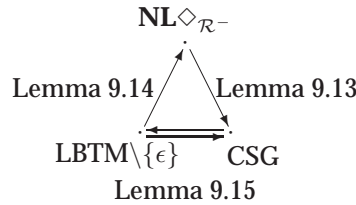


Figure 9.6: A summary of the previous lemma's

Theorem 9.16 *The parsing problem for $\mathbf{NL}\diamond_{\mathcal{R}^-}$ is equivalent to the parsing problem for context sensitive grammars.*

Proof The theorem is an immediate consequence of Lemmas 9.13, 9.14 and 9.15, as shown in Figure 9.6 on the page before. We can decide the derivability of any sequent of $\mathbf{NL}\diamond_{\mathcal{R}^-}$ in nondeterministic linear space according to Lemma 9.14 and any problem which can be solved in nondeterministic linear space can be solved in $\mathbf{NL}\diamond_{\mathcal{R}^-}$ \square

Corollary 9.17 *The decision problem for $\mathbf{NL}\diamond_{\mathcal{R}^-}$ is PSPACE complete.*

Proof Karp (1972) shows that the decision problem for context sensitive grammars, that is, given a description of a context sensitive grammar G and a string s we return 1 if $s \in \mathcal{L}_G$ and 0 otherwise, is PSPACE complete.

Given Theorem 9.16 the PSPACE completeness of the decision problem for $\mathbf{NL}\diamond_{\mathcal{R}^-}$ follows immediately. It is PSPACE hard because we can encode a context sensitive grammar in the structural rule component of $\mathbf{NL}\diamond_{\mathcal{R}^-}$. It is in PSPACE because we can translate any fragment of $\mathbf{NL}\diamond_{\mathcal{R}^-}$, including the structural rules, to a context sensitive grammar. Therefore it is PSPACE complete. \square

9.3 $\mathbf{NL}\diamond_{\mathcal{R}}$

Carpenter (1995) showed that the multimodal Lambek calculus is undecidable if we allow structural rules which duplicate and erase material. A similar result is mentioned by Lincoln, Mitchell, Scedrov & Shankar (1990), where the relation between semi-Thue systems and cyclic linear logic with exponentials is sketched.

In this section I will prove that even if we restrict the structural rules to be linear according to Definition 3.4 the logic $\mathbf{NL}\diamond_{\mathcal{R}}$ is still undecidable. We show this by using the structural rules for the unary connectives to encode the recognition problem for type 0 languages, which is known to be undecidable (Chomsky 1959). This result is mostly symmetric to the results of the previous section, since, by the single restriction on the form of the rules for a type 0 grammar, we will now give a translation encoding top down search for type 0 grammar opposed to the bottom up search of the previous translation.

One of the consequences of the results of this section is therefore that we could have also defined the logic $\mathbf{NL}\diamond_{\mathcal{R}^+}$ which has only non-shrinking structural rules and which also generates the context sensitive languages.

Definition 9.18 *A type 0 grammar G is a tuple $\langle \Sigma, N, S, R \rangle$, where*

Σ is the set of terminal symbols,

N is the set of nonterminal symbols,

S is designated member of N called the start symbol,

R is the set of grammar rules.

As usual, we require N and Σ to be disjoint. The set R consists of rewrite rules $\Gamma \rightarrow \Delta$, where Γ and Δ are lists of symbols from $N \cup \Sigma$. The only restriction on the form of the rules is that Γ contains at least one nonterminal symbol.

A type 0 language \mathcal{L}_G is the set of lists of terminal symbols obtained by taking a type 0 grammar G and computing the closure of the start symbol S under the operation of the rules R .

Since Definition 9.5 and Lemma 9.7 were formulated for general phrase structure grammars, we can apply them here as well to talk about lexicalized type 0 grammars and know that any type 0 grammar G generates the same language as the corresponding lexicalized type 0 grammar G' .

Because our formulation of the Lambek calculus does not allow for empty antecedent derivations, we also need to restrict ourselves to type 0 grammars which do not derive the empty string. However, because determining whether a given type 0 grammar generates the empty string is known to be undecidable, we will propose a simple transformation on type 0 grammars which ensures the strings generated by the grammar are always non-empty.

Definition 9.19 A type 0 grammar G is ϵ free if \mathcal{L}_G does not include the empty string.

From a type 0 grammar G we can create the ϵ free version G' by adding two new nonterminal symbols S' and F and a single new terminal symbol $'.'$ to it. S' will be the start symbol of G' and the rules of G' are exactly those of G with the addition of the following two rules, where S is the start symbol of G .

$$\begin{aligned} S' &\rightarrow SF \\ F &\rightarrow . \end{aligned}$$

It is easy to see that $S \rightarrow \Gamma$ in G iff $S' \rightarrow \Gamma$ in G' because S' , F and $'.'$ don't appear in G .

Note that the ϵ free version of a grammar G is defined in such a way that if G is lexicalized then G' is lexicalized as well.

Definition 9.20 From a ϵ free, lexicalized type 0 grammar G we generate a multimodal Lambek calculus $\mathcal{M}_0(G)$ as in Definition 9.9. $\mathcal{M}_0(G)$ has one unary mode for every nonterminal of G and a single binary mode.

Every lexicalization rule $A \rightarrow \beta$ corresponds to a lexical entry as follows.

$$\text{lex}(\beta) = \diamond_A a/a$$

The goal formula of $\mathcal{M}_0(G)$ is $\diamond_S a/a$

$\mathcal{M}_0(G)$ has a structural rule for every grammar rule $A_1 \dots A_n \rightarrow_{R1} B_1 \dots B_m$ of G .

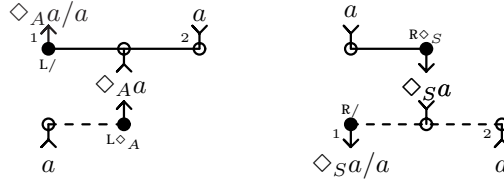


Figure 9.7: Proof structures for the lexicon and the goal formula

$$\frac{\Gamma[\langle \dots \langle \Delta \rangle^{A_1} \dots \rangle^{A_n}] \vdash C}{\Gamma[\langle \dots \langle \Delta \rangle^{B_1} \dots \rangle^{B_m}] \vdash C} [R1]$$

Because $n > 0$ by the single requirement on rules in type 0 grammars, this structural rule is linear according to Definition 3.4. Note that the antecedent and the conclusion of this rule are swapped compared to the translation of Definition 9.9, because in a type 0 grammar m can be 0.

Furthermore, the structural rule component of $\mathcal{M}_0(G)$ contains one of the structural rules for associativity.

$$\frac{\Gamma[\Delta_1 \circ (\Delta_2 \circ \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ \Delta_2) \circ \Delta_3] \vdash C} [Ass1]$$

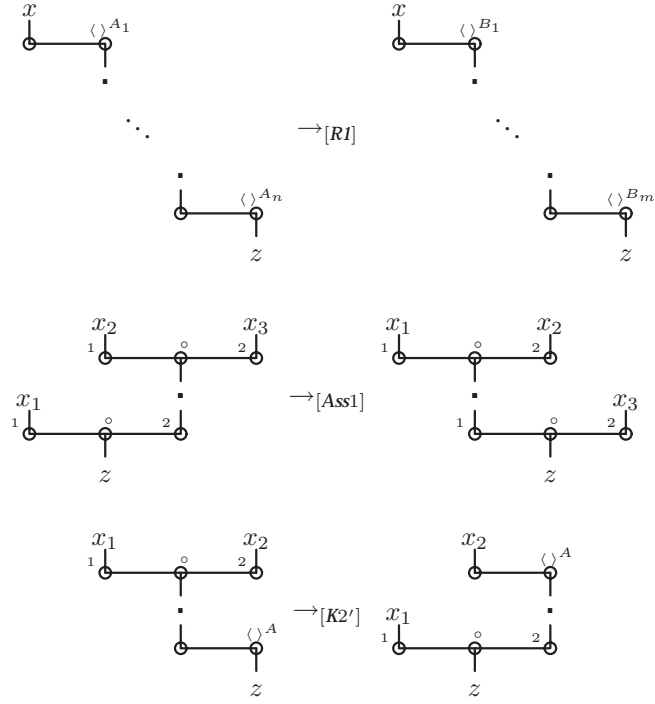
Finally, $\mathcal{M}_0(G)$ has the structural rule of $[K2']$ for every mode $A \in N$.

$$\frac{\Gamma[\langle \Delta_1 \circ \Delta_2 \rangle^A] \vdash C}{\Gamma[\Delta_1 \circ \langle \Delta_2 \rangle^A] \vdash C} [K2']$$

Lemma 9.21 *Let $\mathcal{M}_0(G)$ be a multimodal Lambek calculus which corresponds to a type 0 grammar according to Definition 9.20. For every sequence of lexical hypotheses H_1, \dots, H_n of this grammar, only the proof structure S , which is shown in Figure 9.9, can convert to a hypothesis tree of $H_1, \dots, H_n \vdash \diamond_S a/a$.*

Proof The proof is similar to that of Lemma 9.10. Again, none of the structural conversions modify the order of the lexical formulas so we need to generate a proof structure where the lexical formulas are already in the right order. Connecting the hypothesis a of the proof structure to any conclusion a other than the one from the first lexical formula will result in a proof structure with the wrong order on the hypotheses, connecting the hypothesis a from this first lexical formula to anything but the second lexical formula will also put the hypotheses in the wrong order, and so on. \square

Definition 9.22 *We define a function f_0 which translates the unique active component of the abstract proof structure of Figure 9.10 into a list of terminal and non-terminal symbols as follows. ‘||’ is the concatenation operator and β corresponds to the lexical word which produced the formula in case F is a leaf of the component or ϵ otherwise.*

Figure 9.8: Structural conversions for $\mathcal{M}_0(G)$

$$\begin{aligned}
 f_0(F) &= \beta \\
 f_0(\langle \Gamma \rangle^i) &= f_0(\Gamma) \parallel i \\
 f_0(\Gamma_1 \circ \Gamma_2) &= f_0(\Gamma_1) \parallel f_0(\Gamma_2)
 \end{aligned}$$

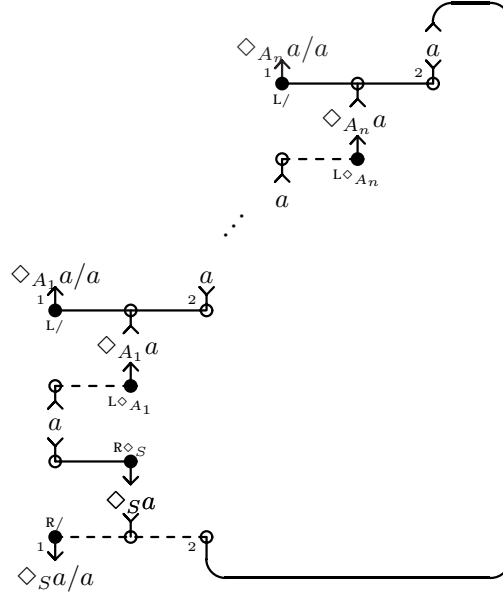
Lemma 9.23 *For every conversion sequence ρ for an abstract proof structure generated from $\mathcal{M}_0(G)$, applying function f_0 of Definition 9.22 to the unique active component of the successive abstract proof structures of ρ and removing all identity transitions yields a derivation \mathcal{D} of G .*

Proof We generate \mathcal{D} by induction on the length l of ρ , simultaneously proving that for every component whenever we compute $f_0(\Gamma_1)$ for the final clause of Definition 9.22 the result will be a non-empty list of terminal symbols.

For the active component \mathcal{C} of the abstract proof structure shown in Figure 9.10, $f_0(\mathcal{I}) = S$ which is a valid prefix for a derivation \mathcal{D} .

Given that the previous n steps of ρ produced a prefix of a derivation \mathcal{D} we extend this derivation as follows, depending on the next conversion $\mathcal{C}_i \xrightarrow{c} \mathcal{R} \mathcal{C}_{i+1}$.

If c is a grammatical conversion, then we extend \mathcal{D} with the corresponding grammatical rule of G .

Figure 9.9: Proof structure for a derivation in $\mathcal{M}_0(G)$

If c is a $[K2']$ conversion we need to show $f_0(\langle \Gamma_1 \circ \Gamma_2 \rangle^A) = f_0(\Gamma_1 \circ \langle \Gamma_2 \rangle^A)$. Because the concatenation operation is associative, both components translate to the same expression: $f_0(\Gamma_1) \parallel f_0(\Gamma_2) \parallel A$. We keep \mathcal{D} unchanged.

If c is an $[Ass1]$ conversion, we do not extend \mathcal{D} . The associativity of the concatenation operation again preserves the translation of the two components. $f_0(\Gamma_1 \circ (\Gamma_2 \circ \Gamma_3)) = f_0((\Gamma_1 \circ \Gamma_2) \circ \Gamma_3) = f_0(\Gamma_1) \parallel f_0(\Gamma_2) \parallel f_0(\Gamma_3)$. Given that the first expression satisfies our invariant, that is, both $f_0(\Gamma_1)$ and $f_0(\Gamma_2)$ are non-empty lists of terminal symbols, $f_0(\Gamma_1 \circ \Gamma_2)$ is the concatenation of these lists and is therefore also a non-empty list of terminal symbols.

If c is a $[L_{\diamond A}]$ contraction, it will remove the $[L_{\diamond A}]$ and $[\langle \rangle^A]$ links, connecting the active component to the $[\circ]$ link above it. This will replace the first nonterminal symbol of $f_0(\mathcal{I})$ with the corresponding terminal symbol according to a lexicalization rule of G , with which we extend the derivation \mathcal{D} . By construction, $f_0(\Gamma_1)$ for the new binary link produces a list consisting of a single terminal symbol.

The final conversion step is always the $[R/]$ contraction. Because the shape of the $[R/]$ redex requires the rest of the abstract proof structure to be on the left branch of the $[\circ]$. Since we have just shown that the $f_0(\Gamma_1)$ produces a non-empty sequence of terminals for every left branch of a $[\circ]$ link, this means we have successfully completed our derivation in G . \square

Lemma 9.24 *Given a ϵ free, lexicalized type 0 grammar G the fragment $\mathcal{M}_0(G)$ of $NL_{\diamond \mathcal{R}}$ which corresponds to it according to Definition 9.20 generates the same*

Proof Because Lemma 9.24 established the decision procedure for $\text{NL}\diamond_{\mathcal{R}}$ is at least as hard as the parsing problem for type zero grammars, which is known to be Turing complete, we only have to give a procedure for enumerating derivations in $\text{NL}\diamond_{\mathcal{R}}$. We have already given many of those, see for example the proof net algorithm of Table 8.1. This establishes $\text{NL}\diamond_{\mathcal{R}}$ is Turing complete. \square

9.4 Conclusions

We have seen how $\text{NL}\diamond_{\mathcal{R}}$, with no other restriction on the structural rules than that they are all linear, is undecidable, but that $\text{NL}\diamond_{\mathcal{R}^-}$ is PSPACE complete when given as its input a set structural rules which are non-expanding in addition to being linear. It appears this restriction gives us more than enough power to adequately describe syntactic phenomena.

An interesting open problem is if it is possible to give a characterization of packages of structural rules for which we can perform the conversions in polynomial time. This would give us a fragment for $\text{NL}\diamond_{\mathcal{R}}$ which is in NP and it will be useful to compare the allowable structural rules for an NP fragment of $\text{NL}\diamond_{\mathcal{R}}$ with the PSPACE fragment $\text{NL}\diamond_{\mathcal{R}^-}$ to see what the trade-off would be.

Another interesting open question is how far we can extend the methods used for the known polynomial fragments of $\text{NL}\diamond_{\mathcal{R}}$, principally NL and formula-restricted versions of L, to more complex fragments.

Though PSPACE is a respectable complexity class for a logic, as a linguistic theory there are formalisms with much better complexity theoretic properties, though several others, such as HPSG (Pollard & Sag 1994), LFG (Kaplan & Bresnan 1982) or definite clause grammars (Tärnlund 1977) are only decidable in restricted cases. Also, PSPACE is only presented as an upper bound on the complexity of $\text{NL}\diamond_{\mathcal{R}}$, where we leave the structural rule component variable. Better complexity results may be obtained by using a fixed structural rule component.

In the next chapter, we will look at a polynomial grammar formalism, Lexicalized Tree Adjoining Grammars and relate it to a fragment of $\text{NL}\diamond_{\mathcal{R}}$, with a fixed set of structural rules and a restriction of the allowed formulas. Moreover, we will prove this correspondence is strong, allowing us to using the polynomial parsing strategies proposed for Lexicalized Tree Adjoining Grammars for this fragment.

CHAPTER 10

PROOF NETS AND TREE ADJOINING GRAMMARS

THIS chapter will relate the proof nets introduced in Chapter 7 to Lexicalized Tree Adjoining Grammars (LTAGs).

After a brief introduction to LTAGs, I will show how the tensor fragment of $\text{NL}\diamond$ corresponds to the substitution only fragment of LTAGs.

Then, I will show how to simulate adjunction with the use of par links and structural conversions. Furthermore, I will show that this correspondence is strong, i.e. that it generates the same trees as LTAGs.

Finally, I will sketch what some proposed extensions to TAGs, notably Multi Component Tree Adjoining Grammars, would look like in the proof net formalism.

This embedding has immediate computational implications, in that it gives us a fragment of the multimodal Lambek calculus for which we can directly use the $O(n^6)$ parsing algorithms which have been proposed for LTAGs (Vijay-Shanker & Joshi 1985, Schabes & Joshi 1988, Schabes & Vijay-Shanker 1990).

Some different perspectives on Lambek calculi and their relation to Tree Adjoining Grammars have been proposed by Joshi & Kulick (1997), Joshi, Kulick & Kurtonina (2001) in their work on what they call ‘partial proof trees’, where they use (partial) Lambek calculus natural deduction proofs as syntactic objects for tree operations.

Especially in the substitution only case, which we will discuss in Section 10.2, the work in this chapter is related to the partial proof tree approach. Our basic objects are just proof structures instead of partial natural deduction proof trees. The main difference with that work are that we formalize the adjunction operation *inside* the proof net calculus instead of defining it as an operation on proof trees and that we prove a formal embedding theorem.

10.1 Tree Adjoining Grammars

Tree Adjoining Grammars (Joshi 1994, Joshi & Schabes 1996) are a formalism where the basic objects are trees. This in contrast with, for example, context free grammars where the tree representation is only a way to portray derivations. Tree Adjoining Grammars, after (Joshi & Schabes 1996), are defined as follows.

Definition 10.1 (Tree Adjoining Grammar) A TAG or Tree Adjoining Grammar is a tuple $\langle \Sigma, N, I, A, S \rangle$, where

Σ is a finite set of terminal symbols,

N is a finite set of nonterminal symbols,

I is a finite set of finite trees, the initial trees, which satisfy the following.

- the root node and all internal nodes are labeled with a nonterminal symbol,
- every leaf of an initial tree is labeled either with a terminal symbol or with a nonterminal symbol which is marked for substitution. We will indicate a nonterminal symbol A is marked for substitution by writing it as A^\downarrow .

A is a finite set of finite trees, the auxiliary trees, which satisfy the following.

- the root node and all internal nodes are labeled with a nonterminal symbol,
- every leaf of an auxiliary tree is labeled either with a terminal symbol or with a nonterminal symbol. One of these leaves must be labeled with the same nonterminal as the root of the tree and be marked as the foot of the auxiliary tree, all other nonterminals are marked for substitution. We will indicate the nonterminal A which is the foot of an auxiliary tree by writing it as A^* .

We will refer to the root node, the foot node and all internal nodes on the path from the root to the foot as the spine of the auxiliary tree.

S is a distinguished nonterminal symbol, the start symbol.

We will call a TAG a Lexicalized Tree Adjoining Grammar or LTAG if every initial and auxiliary tree has at least one leaf labeled with a terminal symbol.

Trees in $I \cup A$ are called elementary trees. Trees built by the composition of other trees are called derived trees.

Example 10.2 The elementary trees for an example Tree Adjoining Grammar are shown in Figure 10.1 on the facing page. The elementary trees for ‘quietly’ and ‘tasteless’ are auxiliary trees, the other trees are initial trees.

Note that this TAG is lexicalized, as every tree has a leaf labeled with a terminal symbol.

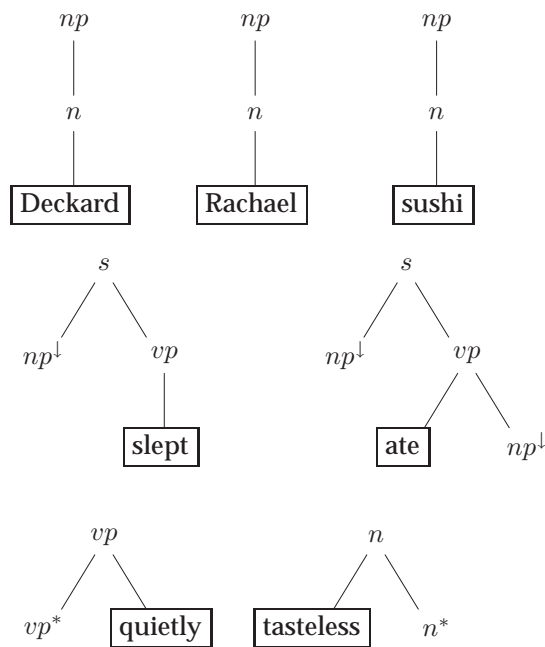


Figure 10.1: An example LTAG lexicon

From the elementary trees we can produce derived trees by the following operations of substitution and adjunction. These are the only rules available in a Tree Adjoining Grammar.

Definition 10.3 (Substitution) Given a tree α which has a leaf l with label A^\downarrow and a tree β with root labeled A , which is derived from an initial tree (i.e. none of its leaves is marked as a foot), the substitution of β at l in α is defined as replacing the leaf A^\downarrow by the tree β .

Visually, substitution looks as shown in Figure 10.2 on the next page.

Definition 10.4 (Adjunction) Given a tree α with root A , which is derived from an auxiliary tree (i.e. one of its leaves is marked as the foot A^*) and a tree β where one of its nodes n is labeled with A , the adjunction of α at n in β is defined as follows. We cut β at n into two trees, β' which has a copy of n labeled with A as a leaf and β'' which has a copy of n also labeled with A as its root. We then substitute α for the leaf A in β' and substitute β'' for the foot node A^* of α .

Perhaps this is most clearly illustrated by the graphical representations of the trees, shown in Figure 10.3 on the following page.

We allow β'' to be empty, that is, A can be a leaf of tree β and in that case it is a leaf of the resulting tree as well and keeps its marking, either for substitution or as a

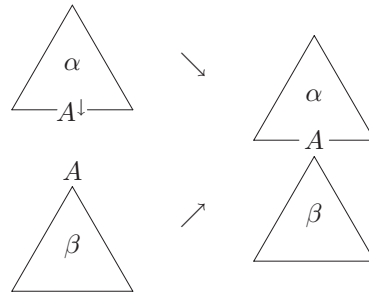


Figure 10.2: Substitution

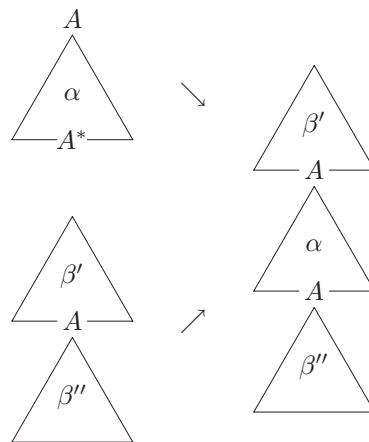


Figure 10.3: Adjunction

foot. In other words, the only marking which disappears after an adjunction step is the foot marking of tree α .

Note that the A node from tree β has been replaced by tree α .

Definition 10.5 (Derivation) A TAG derivation d is a binary tree of trees such that

- every leaf of d is an elementary tree,
- every branch of d is a valid application of the substitution or the adjunction operation,
- the root of d is a tree δ , where the root of δ is S and the leaves of δ are all terminal symbols.

Example 10.6 From the elementary trees of the TAG in Example 10.2, we can derive the sentence 'Rachael slept quietly' as shown in Figure 10.4 on the next page.

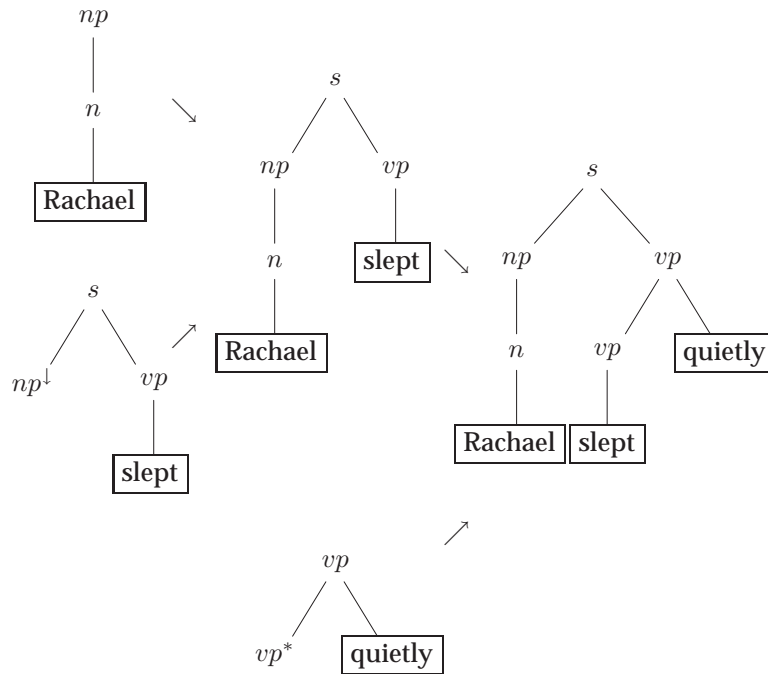


Figure 10.4: LTAG derivation of 'Rachael slept quietly'

Definition 10.7 We say a node in a tree at some step of an LTAG derivation is saturated if no adjunction takes place at that node in any of the remaining steps of the derivation.

We say a tree is saturated at some step of an LTAG derivation if all its nodes are.

The notion of saturation will be important in the next sections, because, depending on whether a node is saturated or not, we will translate it differently.

Example 10.8 Because only one adjunction takes place in the derivation of Figure 10.4, the tree for 'quietly' is trivially saturated during the entire proof.

LTAGs allow multiple terminal leaves on their elementary trees. For the remainder of this chapter, however, we want to restrict ourselves to LTAGs where every elementary tree has exactly one terminal leaf.

Lemma 10.9 For every LTAG grammar g there is an LTAG grammar g' which generates the same language and where every elementary tree has exactly one terminal leaf. We will call this grammar an LTAG1 grammar.

Proof By definition every elementary tree in an LTAG grammar has at least one terminal leaf. We need to show that every elementary tree α with

more than one terminal leaf can be transformed into one with exactly one terminal leaf. For all but one of the terminal leaves of α we do the following: we add a new nonterminal symbol A to N , replace the terminal leaf l by A^\downarrow and add a new initial tree consisting only of root A and the single terminal leaf l . The resulting tree α' , as well as any newly introduced initial trees, have exactly one terminal leaf. Because the new nonterminal symbols of α' occur only in the grammar g' as the root of these newly introduced initial trees, any derivation in g' using α' will contain substitution steps which transform α' to α , with the only difference that the new derivation will have some extra unary branches corresponding to the new nonterminal symbols just before some terminal leaves. \square

In the following sections, we will also restrict ourselves to LTAGs and proof nets where all branches are either unary and binary, though the results generalize immediately to arbitrary n -ary branching systems.

10.2 Substitution Only Tree Adjoining Grammars

The substitution only fragment of LTAGs is a restriction of LTAGs to those where the set of auxiliary trees A is empty. As a consequence, derivations in this fragment can only use the substitution operation.

In the tensor only fragment of $\mathbf{NL}\diamond$, formulas in the lexicon are defined as follows. Note the use of polarity to ensure that every proof structure consisting of formulas from this fragment will contain only tensor links.

Definition 10.10 *Over a set of atomic formulas \mathcal{A} , the set of tensor only formulas is defined as follows.*

$$\begin{aligned} \mathcal{F} &::= \mathcal{N} \\ \mathcal{N} &::= \mathcal{A} \\ &\quad | \square^\downarrow \mathcal{N} \\ &\quad | \mathcal{P} \setminus \mathcal{N} \\ &\quad | \mathcal{N} / \mathcal{P} \\ \mathcal{P} &::= \mathcal{A} \\ &\quad | \diamond \mathcal{P} \\ &\quad | \mathcal{P} \bullet \mathcal{P} \end{aligned}$$

To make the correspondence with LTAGs visually more clear, we write our abstract proof structures slightly differently than we did in Chapter 7, making the correspondence between LTAGs and the tensor only fragment of the multimodal Lambek calculus more immediate.

Instead of writing our abstract proof structures which the conclusions at the bottom, we will write them with the conclusions at the top. Table 10.1 illustrates the translation from proof structures in the notation of Chapter 7 to abstract proof structures in the new notation. The difference with the abstract

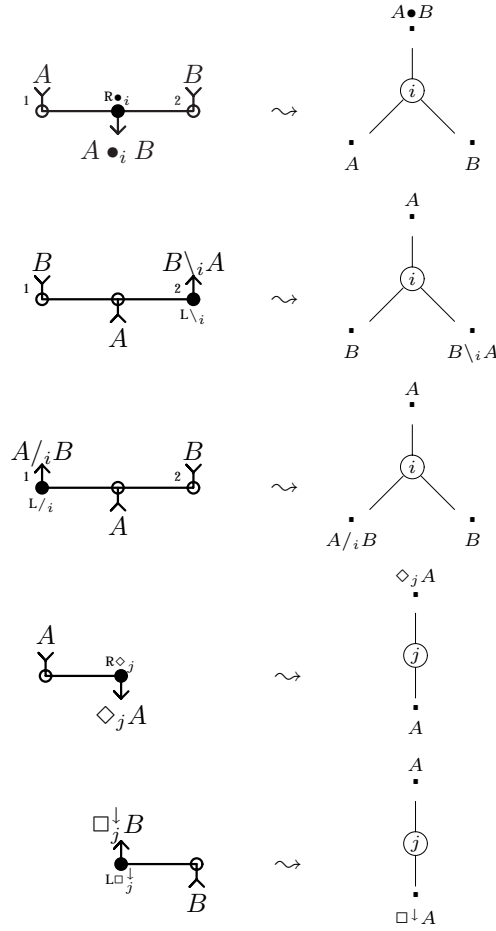


Table 10.1: Tensor links for abstract proof structures

proof structures we’ve seen before is purely notational and doesn’t change the formal properties of the system.

Throughout this chapter, we will label the hypotheses of abstract proof structures by the corresponding word label, like we did in Sections 8.2 and 8.4.

Definition 10.11 *The translation τ_i from LTAG1 trees which have been derived only from initial trees to lexical abstract proof structures is defined as follows.*

Terminal leaves of the LTAG will correspond to word labels in the abstract proof structures.

Nonterminals of the LTAG will correspond to atomic formulas in $NL\Diamond$. The translation of a nonterminal n of an initial tree depends on their position in this tree.

- if n is the root of the tree, it will be translated as $\overset{n}{\bullet}$.

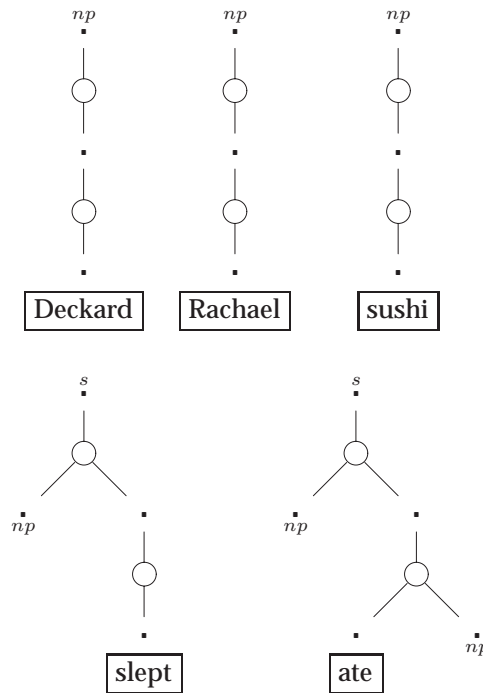


Figure 10.5: Lexical abstract proof structures for the initial trees of Figure 10.1

- if n is a leaf of the tree, it will be translated as \cdot .
- if n is an internal node of the tree, it will be translated as \cdot .

Finally, a unary branch will be translated as a unary tensor link, whereas a binary branch will be translated as a binary tensor link.

Example 10.12 The initial trees of the lexicon of Figure 10.1 will look as shown in Figure 10.5. With the exception of the nonterminals which appear on the internal nodes on Figure 10.1, the two figures are completely isomorphic.

It is not immediately clear that all lexical abstract proof structures generated by the translation τ_i correspond to formulas of $NL\Diamond$. They might instead correspond to lexical *modules* in the sense of Lecomte & Retoré (1998). The difference between a lexical formula and a lexical module is that the latter can have cut or axiom connections lexically determined. That is, modules are lexical proof structures where the internal formulas can be axiomatic or cut formulas.

It is in fact easier to show an embedding of LTAGs using lexical modules, as it would allow us, for example, to have a lexical abstract proof structure with multiple terminal leaves. In the following lemma we prove that every abstract proof structure obtained by the translation of Definition 10.11 corresponds to a formula.

$$\begin{aligned}
l(\text{Deckard}) &= \square^\downarrow \square^\downarrow np \\
l(\text{Rachael}) &= \square^\downarrow \square^\downarrow np \\
l(\text{sushi}) &= \square^\downarrow \square^\downarrow np \\
l(\text{slept}) &= \square^\downarrow (np \setminus s) \\
l(\text{ate}) &= (np \setminus s) / np
\end{aligned}$$

Table 10.2: The lexical formulas corresponding to Figure 10.5

Lemma 10.13 *For every LTAG1 initial tree α , $\tau_i \alpha$ corresponds to a tensor only formula of $NL \diamond$.*

Proof In order to show that we only produce lexical formulas we need to show our lexical abstract proof structure corresponds to a lexical proof structure where all internal formulas are *flow* formulas, i.e. the main formula of exactly one link. If we can do this, we can apply the translation of Table 10.1 in reverse to determine the proof structure corresponding to the lexical entry and thereby the formula corresponding to it.

The leaves and the root of the lexical abstract proof structures our translation produces are all axiomatic formulas, so they can't be the main formula of any link, whereas the unique terminal leaf must be the main formula of its link L . If a is the arity of L , removing it will produce a distinct abstract proof structures. Unless they are trivial, for each of these abstract proof structures the vertex which was connected to L must be the main vertex of its link, because it would be axiomatic otherwise. In this way we inductively assign every internal vertex to be to main vertex of a link. \square

Example 10.14 *The lexical abstract proof structures of Figure 10.5 on the preceding page correspond to the lexical entries of Table 10.2.*

Lemma 10.15 *Let d be an LTAG1 derivation tree with root δ and leaves $\alpha_1, \dots, \alpha_n$, such that every α_i is an initial tree. There is a proof net derivation, where ρ is the empty conversion sequence, of*

$$\tau_i \alpha_1, \dots, \tau_i \alpha_n \vdash s$$

such that the hypothesis tree Γ_S is isomorphic to δ .

Proof We prove by induction on the depth D of the derivation tree d , that, given the isomorphism of the immediate subtrees, the tree at the current depth is also isomorphic.

If $D = 0$, then tree consists of a single initial tree α and $\tau_i \alpha$ is by definition isomorphic to it.

If $D > 0$, then by induction hypothesis the two immediate subtrees of α , β_1 and β_2 , are isomorphic to $\tau_i \beta_1$ and $\tau_i \beta_2$. β_2 has root A and is substituted for one of the leaves of β_1 which is marked for substitution as A^\downarrow . But this means $\tau_i \beta_1$ has a leaf with hypothesis A and $\tau_i \beta_2$ has a root with conclusion A , so

we can connect these two abstract proof structures with an axiom connection to produce $\tau_i\alpha$, which is isomorphic to α . \square

10.3 Lexicalized Tree Adjoining Grammars

Now the correspondence between the abstract proof structures of Figure 10.5 and the initial trees of Figure 10.1 is very clear. The only difference is that the nonterminal symbols on the internal nodes are absent from the abstract proof structures. As these nonterminal symbols play a crucial role for the adjunction operation we need to find a way to make them appear in the abstract proof structures as well.

To do this we reintroduce the par links for our abstract proof structures in Table 10.3. We only need the $[L\bullet_i]$ and the $[R\backslash_i]$ links in this section. In Sections 10.4 and 10.5 we will find uses for the $[R\Box_i^{\perp}]$ and the $[L\Diamond_i]$ links respectively. By left to right symmetry, we could have used the $[R/i]$ link instead of the $[R\backslash_i]$; the current choice is arbitrary.

For our encoding of the internal nodes we make use of the following two theorems, which hold in the base logic for every formula A and B and for every mode i . We will instantiate i with specific adjunction modes shortly.

$$\begin{array}{l} (A/iB) \bullet_i B \vdash A \\ A \vdash (B/iA)\backslash_i B \end{array}$$

When an internal node in an LTAG is labeled with nonterminal B we encode this with a formula $(A/iB) \bullet_i B$ in a negative context and with a formula $(B/iA)\backslash_i B$ in a positive context. In the abstract proof structures, an insertion point will therefore look as shown in Figure 10.6, with the negative insertion point shown on the left and the positive insertion point on the right. Note the symmetry between the two configurations.

When no adjunction operations take place at an insertion point, we can use an axiom connection followed by a contraction to remove the formulas corresponding to the B nonterminal and continue our derivation as usual. When an adjunction *does* take place at this node we will need to use structural conversions to eliminate the insertion point. The structural conversions, however, will take care that we produce the result corresponding to the adjunction operation of LTAGs.

Again, if we would choose to use lexical modules, a single type of insertion point would suffice. When we prove in Lemma 10.19 that our lexical abstract proof structures correspond to formulas of the multimodal Lambek calculus, we crucially need the two different insertion points of Figure 10.6.

For auxiliary trees, instead of giving the foot node some special marking, we mark the *path* to it by two binary modes: 1, indicating the foot node is in the first (left) subtree, and 2, indicating the foot node is in the second (right) subtree. For the unary branches a single mode 1 suffices, whereas, in general, for an n -ary branch we would need n different modes to indicate in which of the subtrees the foot is located. Mode 0, occurring both as a unary and a

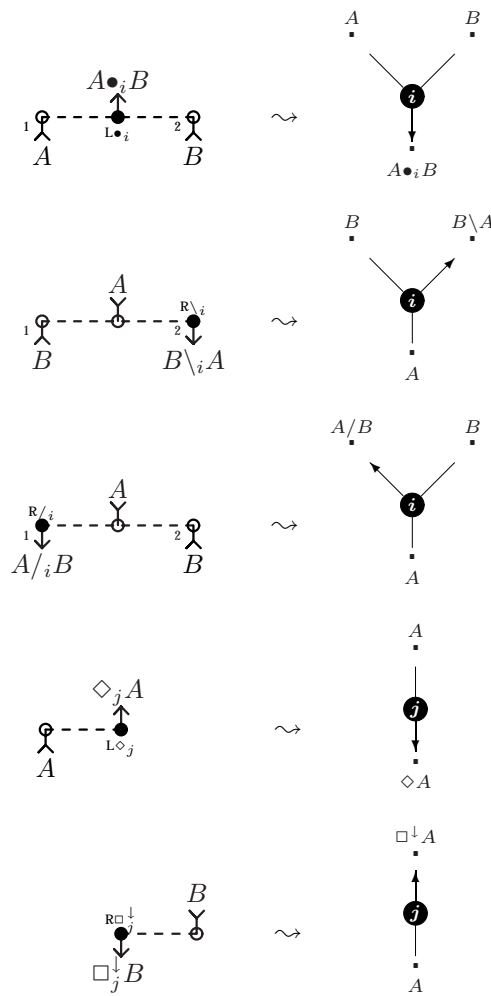


Table 10.3: Par links for abstract proof structures

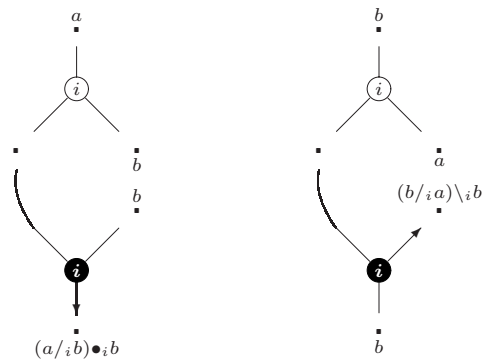


Figure 10.6: Abstract proof structures for insertion points

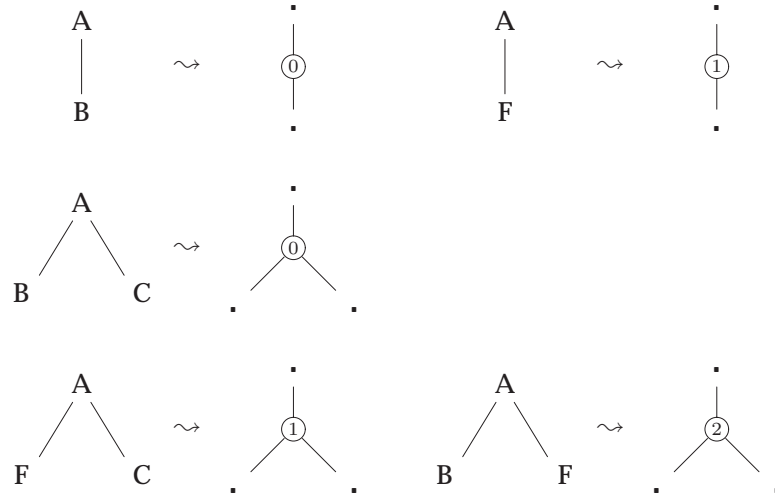


Table 10.4: Translation for LTAG connections

binary mode, indicates none of the subtrees contain a foot node. The unary and binary mode 0 are the only external modes of this grammar.

For the adjunction operation we need to distinguish at least between nodes which are on the spine of an auxiliary tree and other nodes. However, it is also convenient to have a different mode for upward and downward movement. This gives us four adjunction modes: f for movement downward to the foot node, h for movement upward from the foot node, d for downward movement anywhere else and u for upward movement anywhere else.

Definition 10.16 We define a translation τ from LTAG trees to abstract proof structures. Table 10.4 shows how the connections are translated; in all cases the node marked F indicates this subtree contains a leaf which is marked as the foot node, B and C indicate the respective subtrees do not contain as leaf which is marked as the foot node.

The nodes themselves are translated as shown in Table 10.5 on the facing page. We distinguish between the root node, positive and negative internal nodes, positive and negative foot nodes, terminal leaves and leaves marked for substitution. Note that a root node is just a negative internal node with an additional A conclusion and that foot nodes are treated as internal nodes with an additional A hypothesis. The vertices marked α_1 and α_2 denote where the nodes are attached to connections or, in the case of α_1 , to a substitution or terminal leaf.

Mode x is instantiated as either f or d depending on whether the node it translates is on the spine or not. Mode y is instantiated as either h or u depending on whether it is on the path to the foot node.

With respect to an LTAG derivation d , τ_d is defined as τ with the exception that a node in an LTAG tree which is saturated according to Definition 10.7 is translated

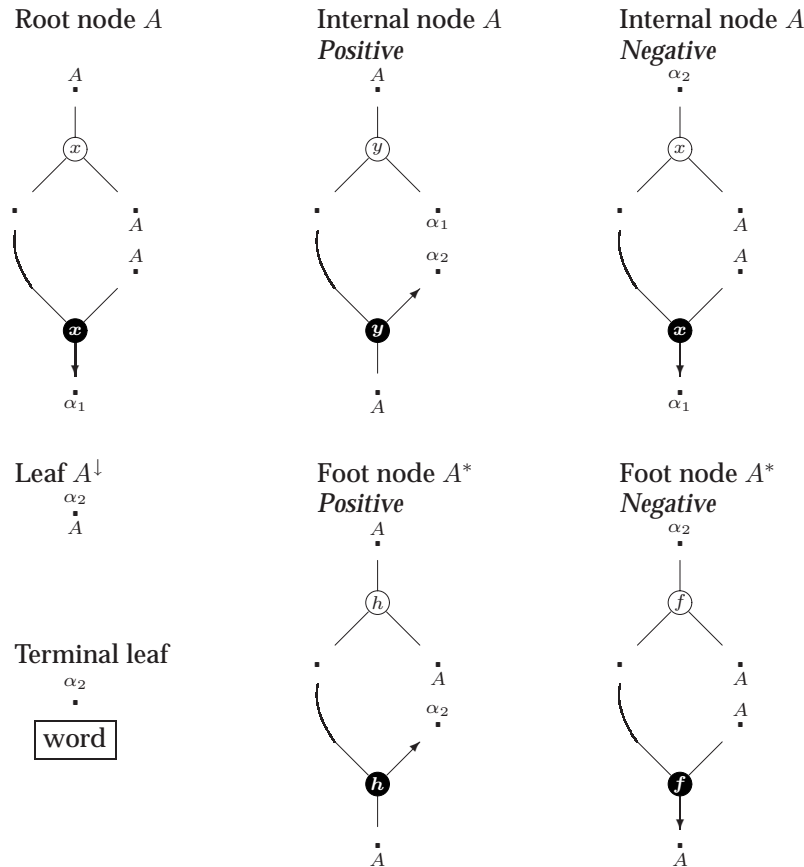


Table 10.5: Translation for LTAG nodes

as a simple vertex like in Definition 10.11 instead of as one of the insertion points shown in Table 10.5.

Definition 10.17 We say an abstract proof structure obtained from an LTAG1 tree by means of τ is saturated if it does not contain mode u , d , f or h and unsaturated if it does.

We obtain the saturated abstract proof structure corresponding to an unsaturated one by performing the appropriate axiom connections and contractions.

The obvious intuition here is that a saturated tree α in an LTAG derivation d will correspond to the saturated abstract proof structure $\tau_d \alpha$.

Definition 10.18 We say an abstract proof structure obtained from an LTAG1 tree by translation τ is auxiliary if the corresponding saturated abstract proof structure has a path from its root to one of its leaves passing only unary mode 1, binary mode 1 through its left leaf and binary mode 2 through its right leaf.

We say an abstract proof structure is initial otherwise.

$$\begin{aligned}
l(\text{Deckard}) &= \square_0^\downarrow((\square_0^\downarrow((np/dnp) \bullet_a np)/_d n) \bullet_a n) \\
l(\text{Rachael}) &= \square_0^\downarrow((\square_0^\downarrow((np/dnp) \bullet_a np)/_d n) \bullet_a n) \\
l(\text{sushi}) &= \square_0^\downarrow((\square_0^\downarrow((np/dnp) \bullet_a np)/_d n) \bullet_a n) \\
l(\text{slept}) &= \square_0^\downarrow(((np \setminus_0 (s/ds) \bullet_a s)/_d vp) \bullet_a vp) \\
l(\text{ate}) &= (((np \setminus_0 (s/ds) \bullet_a s)/_d vp) \bullet_a vp)/_0 np \\
l(\text{quietly}) &= ((vp/_f vp) \setminus_f vp) \setminus_1 ((vp/_h vp) \bullet_h vp) \\
l(\text{tasteless}) &= ((n/_f n) \bullet_f n)/_2 ((n/_h n) \setminus_h n)
\end{aligned}$$

Table 10.6: The lexicon corresponding to the elementary trees of Figure 10.1

Auxiliary abstract proof structures will correspond to LTAG trees which have been derived from auxiliary trees and which can therefore be adjoined to other trees. Initial abstract proof structures will correspond to LTAG trees derived from initial trees.

Lemma 10.19 *For every elementary LTAG1 tree α , $\tau\alpha$ corresponds to a formula of the multimodal Lambek calculus.*

Proof This lemma is a simple extension of Lemma 10.13. We again need to show we can assign a main vertex to every tensor link in such a way that every internal vertex is the main vertex of exactly one link, because then we can apply the translation of Tables 10.1 and 10.3 in reverse to find out the lexical formulas.

Look at Table 10.5 again. In every case the tensor link is a $[L/a]$ link, which means the main formula of this tensor link is the internal vertex. The difference between the positive and the negative versions of the internal and foot nodes is that for the former the vertex labeled with α_1 is not the main vertex of any links in the insertion point but the vertex labeled with α_2 is, whereas for the latter the situation is reversed. So we first follow the strategy of Lemma 10.13 to obtain the proof structure $\tau_i\alpha$, assign modes according to Table 10.4, and finally, depending on whether the internal vertices are the main formula of the link above or below them, assign them a negative or a positive insertion point respectively. \square

Example 10.20 *The elementary trees of Figure 10.1 correspond to the lexical assignments of Table 10.6. The saturated versions of the translated initial trees in this table are the formulas shown in Table 10.2, with the exception that all ‘modeless’ connectives of Table 10.2 are now mode 0. The saturated versions of ‘quietly’ and ‘tasteless’ are $vp \setminus_1 vp$ and $n/_2 n$ respectively.*

The structural conversions come in four groups: one conversion for each of the adjunction modes in combination with each of the modes used to mark the spine, as shown in Table 10.7. Note that the structural rules for mode h are inverses to the structural rules for mode f and that the only difference between modes f and d and between modes h and u is that modes d and u replace an internal mode by an external mode, that is, modes d and u erase the path to the foot node, whereas modes f and h preserve it.

| Mode f | Mode h |
|--|--|
| $\frac{\Gamma[\Delta_1 \circ_f \langle \Delta_2 \rangle^1] \vdash C}{\Gamma[\langle \Delta_1 \circ_f \Delta_2 \rangle^1] \vdash C} [K2]$ | $\frac{\Gamma[\langle \Delta_1 \circ_h \Delta_2 \rangle^1] \vdash C}{\Gamma[\Delta_1 \circ_h \langle \Delta_2 \rangle^1] \vdash C} [K2']$ |
| $\frac{\Gamma[\Delta_1 \circ_f (\Delta_2 \circ_1 \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_f \Delta_2) \circ_1 \Delta_3] \vdash C} [F1]$ | $\frac{\Gamma[(\Delta_1 \circ_h \Delta_2) \circ_1 \Delta_3] \vdash C}{\Gamma[\Delta_1 \circ_h (\Delta_2 \circ_1 \Delta_3)] \vdash C} [1H]$ |
| $\frac{\Gamma[\Delta_1 \circ_f (\Delta_2 \circ_2 \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_f \Delta_3) \circ_2 \Delta_2] \vdash C} [F2]$ | $\frac{\Gamma[(\Delta_1 \circ_h \Delta_3) \circ_2 \Delta_2] \vdash C}{\Gamma[\Delta_1 \circ_h (\Delta_2 \circ_2 \Delta_3)] \vdash C} [2H]$ |
| Mode d | Mode u |
| $\frac{\Gamma[\Delta_1 \circ_d \langle \Delta_2 \rangle^1] \vdash C}{\Gamma[\langle \Delta_1 \circ_d \Delta_2 \rangle^0] \vdash C} [K2]$ | $\frac{\Gamma[\langle \Delta_1 \circ_u \Delta_2 \rangle^1] \vdash C}{\Gamma[\Delta_1 \circ_u \langle \Delta_2 \rangle^0] \vdash C} [K2']$ |
| $\frac{\Gamma[\Delta_1 \circ_d (\Delta_2 \circ_1 \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_d \Delta_2) \circ_0 \Delta_3] \vdash C} [D1]$ | $\frac{\Gamma[(\Delta_1 \circ_u \Delta_2) \circ_1 \Delta_3] \vdash C}{\Gamma[\Delta_1 \circ_u (\Delta_2 \circ_0 \Delta_3)] \vdash C} [1U]$ |
| $\frac{\Gamma[\Delta_1 \circ_d (\Delta_2 \circ_2 \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_d \Delta_3) \circ_0 \Delta_2] \vdash C} [D2]$ | $\frac{\Gamma[(\Delta_1 \circ_u \Delta_3) \circ_2 \Delta_2] \vdash C}{\Gamma[\Delta_1 \circ_u (\Delta_2 \circ_0 \Delta_3)] \vdash C} [2U]$ |

Table 10.7: Structural rules

Lemma 10.21 *If tree α is a saturated LTAG tree with respect to derivation d , which is adjoined at node n of a tree β to form tree γ , then we can transform $\tau_d\alpha$ and $\tau_d\beta$ to $\tau_d\gamma$ by using axiom connections, the structural rules and a contraction.*

Proof Given that node n is unsaturated, its translation will have one of the forms shown in Table 10.5. We attach $\tau_d\alpha$, at its root and at its foot, to the atomic formulas of the insertion point. By construction, all nodes of $\tau_d\alpha$ are saturated and there is a path marked from the root to the foot node.

After we attach the root and foot nodes of $\tau_d\alpha$ to the atomic formulas of the insertion point, the abstract proof structure will look schematically as shown in the initial situation of Figure 10.7 in case of a negative insertion point and as shown in the initial situation of Figure 10.8 in case of a positive insertion point. Mode x is again instantiated by either d or f , while y represents either mode u or h . In every case we can use the structural conversions for the mode we encounter to produce the conversions of sequence ρ in the figure, after which we can contract the par link by the appropriate contraction. Compare the final stage of both figures to the result of the adjunction operation shown in Figure 10.3.

Since none of the conversions operate in $\tau_d\beta'$ or $\tau_d\beta''$, every node which was saturated or unsaturated in $\tau_d\beta$ is still saturated or unsaturated respectively in $\tau_d\gamma$, with the exception of the unsaturated node n , which has been replaced by the saturated abstract proof structure $\tau_d\alpha$.

If the insertion point was on the path to the foot of an auxiliary tree then modes f and h will have left the path of $\tau_d\alpha$ intact, and $\tau_d\gamma$ still has a path to

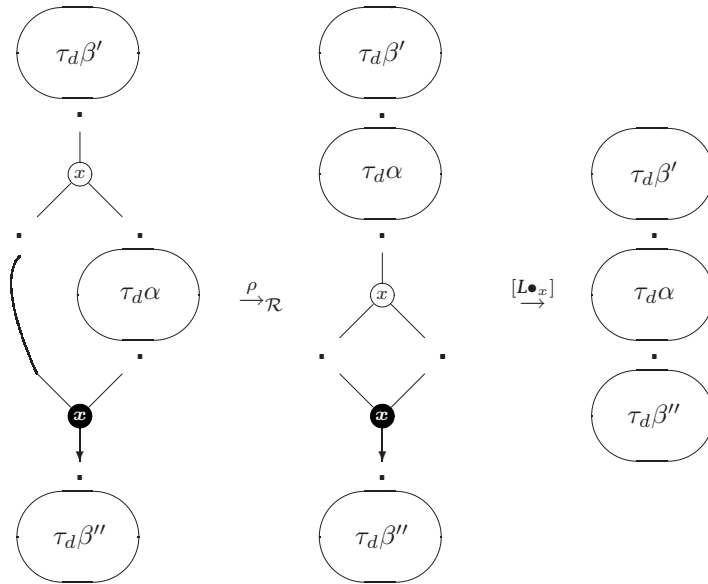


Figure 10.7: Simulating adjunction for a negative insertion point

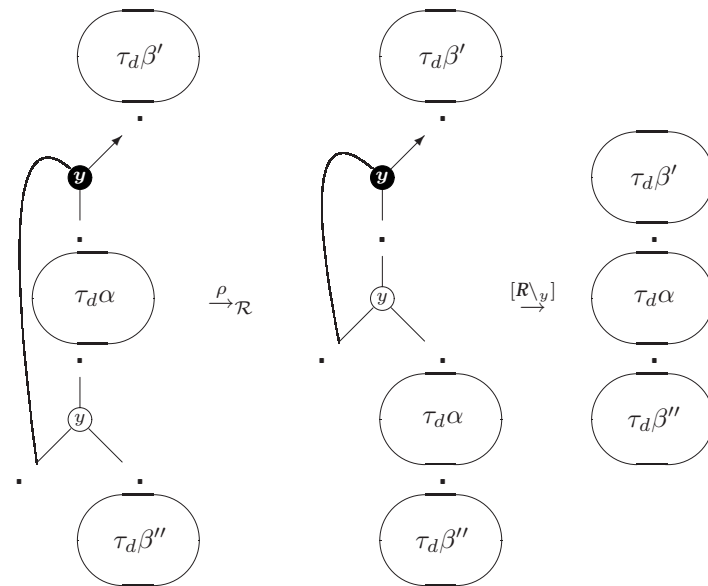


Figure 10.8: Simulating adjunction for a positive insertion point

its foot, only now passing through $\tau_d\alpha$ as well. Therefore the abstract proof structure we have produced is $\tau_d\gamma$.

If the insertion point was not on the path to the foot of an auxiliary tree then modes u and d will have replaced all internal modes by mode 0 and we

have again produced $\tau_d\gamma$. □

Lemma 10.22 *Let $\tau\alpha_1, \dots, \tau\alpha_n$ be abstract proof structures obtained from elementary trees of an LTAG by the translation of Definition 10.16. If we want to produce a proof net from these abstract proof structures, we only need to consider the following three types of axiom connections.*

- (i) *connect the two atomic formulas of an insertion points to eachother,*
- (ii) *connect the root and foot node of an auxiliary abstract proof structures to the atomic formulas of a single insertion point,*
- (iii) *connect the root of an initial aps to any leaf which is not a foot node of an aps.*

Proof To convert an abstract proof structure generated by τ to a hypothesis tree containing only external modes, we need to do two things: remove all par links by the appropriate contraction and remove all internal modes.

For modes u, d, f and h , given that all structural conversions preserve them, we can only remove them using the appropriate contraction. This can be done either directly, by connecting the atomic formulas of an insertion point to eachother, or indirectly, by connecting an auxiliary aps at the root and foot node to the two atomic formulas of the insertion point.

For modes 1 and 2, given that τ does not produce par links for these modes, we can only removing them using the structural conversions for the u and d modes, which erase the spine of a given auxiliary aps.

Connecting an initial aps to an insertion point would prevent the contraction which is necessary for removing this insertion point from taking place.

Connecting an atomic formula of one insertion point to an atomic formula of another insertion point would prevent the contractions of both insertion points. Similarly for connecting the root and foot of an auxiliary aps to the atomic formulas of different insertion points.

Connecting an initial aps to the foot node of an auxiliary aps results in an aps without marking to its foot but with some 1 and 2 modes still in it, thereby preventing future contraction of the d or u par link we would attach it to.

Attaching an auxiliary aps \mathcal{A}_1 to the foot node of an auxiliary aps \mathcal{A}_2 results in a new auxiliary aps \mathcal{B} . However, this new auxiliary aps is identical to the one we would obtain by connecting the root and foot of \mathcal{A}_1 to the insertion point at the foot of \mathcal{A}_2 and removing the h or f mode of the foot by the appropriate structural conversions and contraction once \mathcal{A}_1 is saturated. □

Lemma 10.23 *For every LTAG derivation d we can construct an LTAG derivation d' , which ends in the same tree and where*

- (i) *no adjunctions take place at leaves which are marked for substitution until after the substitution has taken place,*

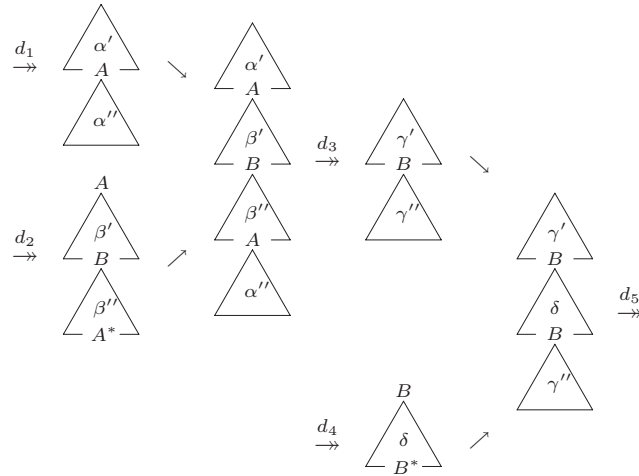


Figure 10.9: Node B of tree β is unsaturated

(ii) every time an auxiliary tree α is adjoined to a node, α is saturated.

Proof For property (i), we only need to realize that if we can perform an adjunction at node n before the substitution, we can perform it after the substitution as well and obtain the same result.

We prove property (ii) of the lemma by induction on the total number of adjunctions taking place in auxiliary trees after they have been adjoined. If there are no such adjunctions, our derivation is of the required form.

If there are, our derivation must be of the following form: an auxiliary tree β is adjoined to tree α and, at some later point in the derivation, a tree δ is adjoined at node B of β . So we are in the situation shown in Figure 10.9.

We can rearrange this derivation by adjoining tree δ to tree β before tree β is adjoined to tree α . This will make node B saturated. The result is shown in Figure 10.10 on the facing page.

The number of adjunctions in the auxiliary tree β is decreased by one, whereas for tree α it has remained constant. Therefore, the total number of unsaturated nodes in the derivation has decreased by one. \square

Theorem 10.24 *There is an LTAG1 derivation d ending in tree γ if and only if there is a LTAG-proof net derivation ending in the tree $\tau_d\gamma$.*

Proof

[\Rightarrow] First, we transform our LTAG derivation d , according to Lemma 10.23, into an LTAG derivation d' also ending in α where all trees which are adjoined are saturated.

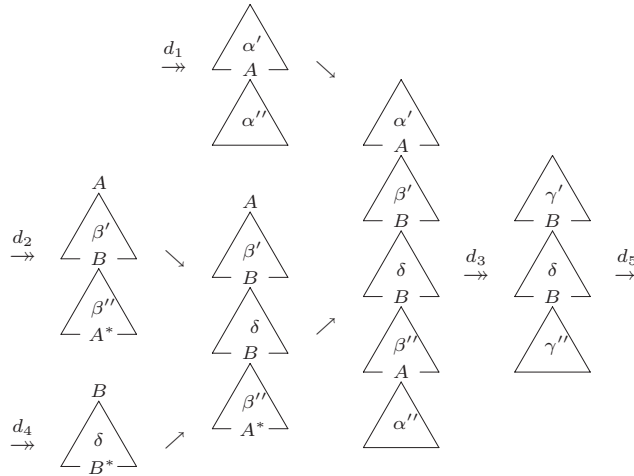


Figure 10.10: Node B of tree β is saturated

By Lemma 10.19 all leaves $\alpha_1, \dots, \alpha_n$ of the derivation tree d , which are by definition elementary trees, correspond to $\tau\alpha_1, \dots, \tau\alpha_n$. By performing an axiom connection and a contraction for all insertion points which are saturated with respect to d , we obtain $\tau_d\alpha_1, \dots, \tau_d\alpha_n$.

We now prove that whenever two LTAG trees β_1 and β_2 are combined using the adjunction or substitution operation to form tree β , we can likewise combine $\tau_d\beta_1$ and $\tau_d\beta_2$ to form $\tau_d\beta$, thereby proving by means of induction on the depth of d that we produce a derivation of $\tau_d\gamma$.

If β_1 and β_2 are combined using the substitution operation, then performing an axiom connection to $\tau_d\beta_1$ and $\tau_d\beta_2$ produces $\tau_d\beta$.

If β_1 and β_2 are combined by means of adjunction, then, because we only adjoin saturated trees, Lemma 10.21 tells us we can produce $\tau_d\beta$ from $\tau_d\beta_1$ and $\tau_d\beta_2$.

[\Leftarrow] Lemma 10.22 tells us that we only have to consider axiom connections of the following patterns: attaching the root and foot of a single auxiliary aps to the atomic formulas of an insertion point, connecting the two atomic formulas of a single insertion point and attaching the root of an initial aps to a leaf of another aps.

We define an interpretation function τ^{-1} from abstract proof structures obtained by τ to LTAG trees as follows. We travel from the root of the abstract proof structure to the leaves. Insertion points are treated as shown in Figure 10.11. If the formula a is not connected, we take the left path and generate the nonterminal a , if a is connected, we follow the right path, not generating any nonterminal symbol for the current node.

Binary branches of modes 0, 1 and 2 are translated as binary branches in

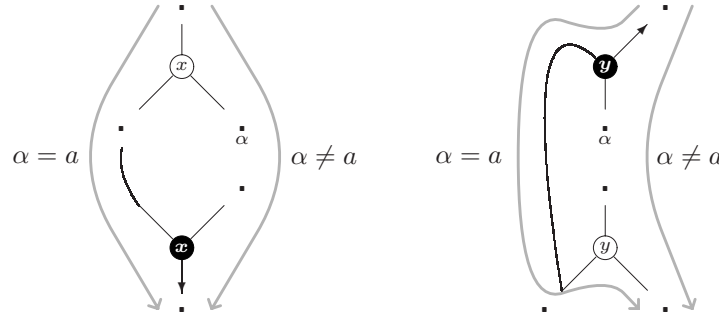


Figure 10.11: Traversal of inserion points

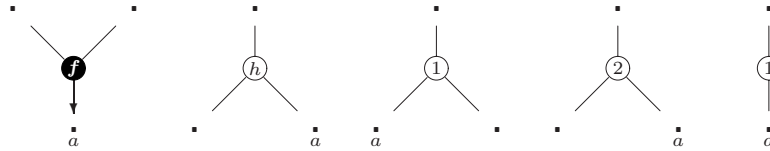


Figure 10.12: Foot nodes

the LTAG, whereas unary branches of modes 0 and 1 are translated as LTAG unary branches. All nonterminal leaves are translated as leaves marked for substitution, with the exception of leaves of the forms shown in Figure 10.12 which are translated as foot nodes.

Now it is easy to verify that for every LTAG tree α , $\tau^{-1}\tau\alpha = \alpha$, that connecting the atomic formulas of an insertion point n of \mathcal{A}_1 such that $\tau^{-1}\mathcal{A}_1 = \alpha_1$ to the root and foot of an auxiliary \mathcal{A}_2 such that $\tau^{-1}\mathcal{A}_2 = \alpha_2$ produces an \mathcal{A}_3 such that $\tau^{-1}\mathcal{A}_3$ is equivalent to adjoining α_2 to node n of α_1 , and that connecting an atomic leaf to an atomic root formula will produce a translation equivalent to the substitution operation applied to the two LTAG trees.

The final possible axiom connection according to Lemmas 10.22 is to attach the two atomic formulas of an insertion point to eachother. If the node is internal, this will have the effect of erasing the nonterminal label from that node of the tree. However, we can reconstruct the nonterminal label from earlier parts of the derivation, so this is not a problem.

Now let \mathcal{A} be the abstract proof structure we obtain after performing all axiomatic connections. For every conversion c such that $\mathcal{A}_n \xrightarrow{c} \mathcal{A}_{n+1}$ we have that $\tau^{-1}\mathcal{A}_n = \tau^{-1}\mathcal{A}_{n+1}$; in case c is a contraction we replace a configuration producing an empty nonterminal labeling with a single vertex also producing an empty nonterminal labeling, in case c is a structural conversion only one of the two branches has an interpretation according to τ^{-1} , so again the conversion is neutral with respect to τ^{-1} .

Because we have already established that every axiom connection or pair

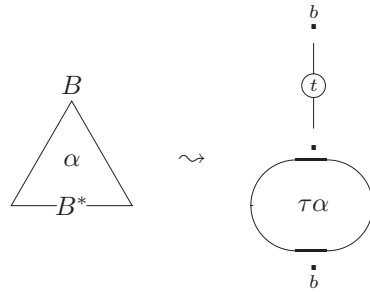


Figure 10.13: Mode information for selective and obligatory adjunction

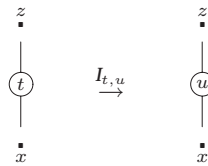


Figure 10.14: Inclusion conversion

of axiom connections corresponds to an LTAG operation under τ^{-1} this establishes that the final hypothesis tree Γ_S is isomorphic to the final LTAG tree α . \square

10.4 Adjoining Constraints

For linguistic reasons, Tree Adjoining Grammars are sometimes defined with constraints on adjunction. That is, for every node in a tree one of the following constraints can be specified.

[Selective Adjunction ($SA(T)$)] Only members of the set $T \subseteq A$ can be adjoined to the node.

[Null Adjunction (NA)] No adjunction is allowed at the node. This is equivalent to $SA(\emptyset)$.

[Obligatory Adjunction ($OA(T)$)] An auxiliary tree of $T \subseteq A$ must be adjoined to the node.

Nodes without constraints can be seen as have $SA(A)$ as their specification. For modeling these adjoining constraints, we use the unary connectives to code features, conform Heylen (1999, Chapter 8).

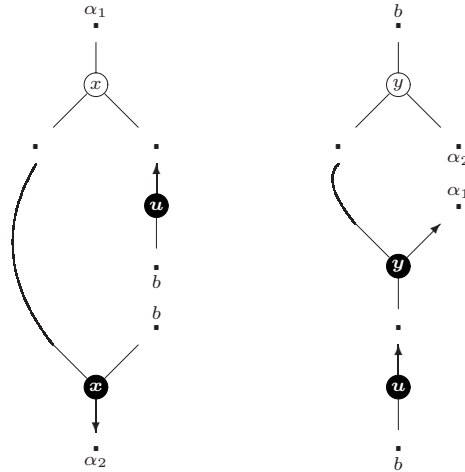


Figure 10.15: Insertion points for obligatory adjunction

In principle, the constraints allow us to refer to every possible subset of the set A , so for a grammar with a auxiliary trees, we would need 2^a modes. However, we will assume that a realistic grammar will only refer to some quite limited number of subsets.

Every abstract proof structure corresponding to an auxiliary tree α will have an extra unary branch with mode t which corresponds to the singleton set containing only α , as shown in Figure 10.13 on the page before.

In addition, we will have a structural conversion $[I_{t,u}]$ of the form shown in Figure 10.14 on the preceding page for every singleton set t and every set u such that $t \subsetneq u$.

The insertion points will now look as follows: for obligatory adjunction the insertion point will make use of the underderivability of the following sequents. $x \in \{f, d\}, y \in \{h, u\}$

$$\begin{aligned} (A/x \square_t^\perp B) \bullet_x B \not\vdash A \\ A \not\vdash (B/y A) \setminus_y \square_t^\perp B \end{aligned}$$

The corresponding abstract proof structures are shown in Figure 10.15.

For the abstract proof structures above, we won't be able to contract the $[R \square_u^\perp]$ link unless we attach the B formula to an auxiliary aps where the root is connected to a $[\langle \rangle^t]$ link to which we can apply the $[I_{t,u}]$ conversion followed by the $[R \square_u^\perp]$ contraction. After that, we can simulate adjunction with the structural rules as before.

For selective adjunction we make use of the fact that $\square_t^\perp B \vdash \square_t^\perp B$ for every formula B . So our insertion points will correspond to the following theorems

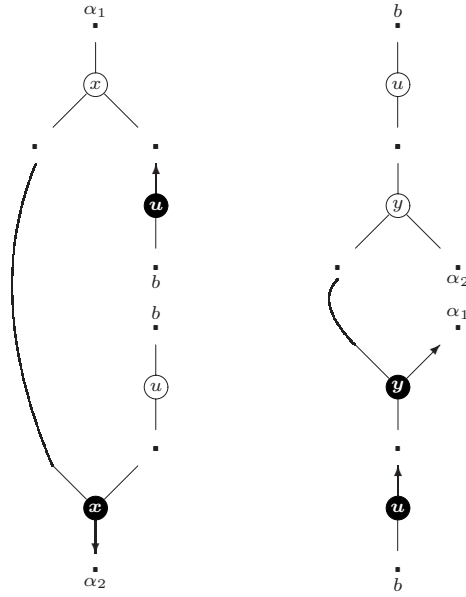


Figure 10.16: Insertion points for selective adjunction

$$\begin{aligned} & (A/x \square_u^\perp B) \bullet_x \square_u^\perp B \vdash A \\ & A \vdash (\square_u^\perp B/y A) \setminus_y \square_u^\perp B \end{aligned}$$

and will look as follows.

In this case, we have two possibilities.

- (i) no adjunctions take place at B , which means we can connect the two B formulas of the insertion point after which we are in the right configuration to perform the two necessary contractions.
- (ii) the B formulas are attached to an aps α where the root B of α is connected by means of a $[\langle \rangle^t]$ link such that there is a $[I_{t,u}]$ conversion.

In the second case, we apply the $[I_{t,u}]$, contract the two unary links and proceed as before. Only after the adjunction is complete do we add a final structural rewrite to remove the last $[\langle \rangle^t]$ link as well. We need the two structural conversions of Figure 10.17 on the following page for this.

Observe that if we are only interested in selective adjunction and not in obligatory adjunction, we can do without the extra structural rules if we replace the auxiliary trees as shown in Figure 10.18 on the next page.

As a final remark, the easiest way to model null adjunction constraints in a proof net system is to make a node which is not an insertion point at all. Treating it as a special case of selective adjunction, however, makes it possible to see the nonterminal symbol of the node with the null adjunction constraint in the proof net system as well.

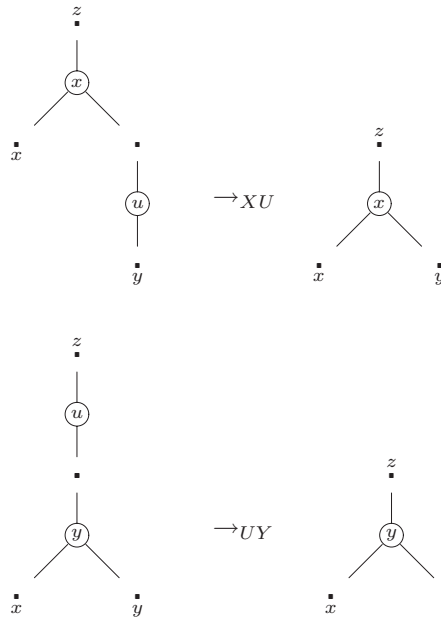


Figure 10.17: Erasing the u subset marking

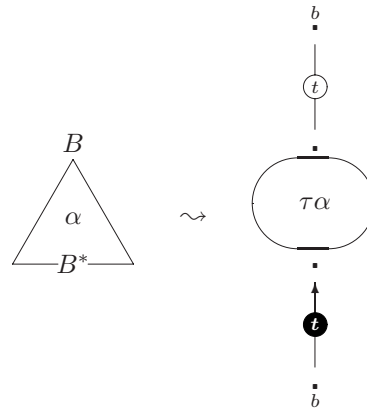


Figure 10.18: Alternative auxiliary tree for selective adjunction only

10.5 Multi Component Tree Adjoining Grammars

Multi Component TAGs (MCTAGs) were first introduced by Joshi, Levi & Takahashi (1975), then under the name simultaneous TAGs. MCTAGs are a natural extension to TAGs where, instead of adjoining a single auxiliary tree, a (non-empty) set of auxiliary trees is adjoined at the same time.

Vector Multi Component TAGs (VMCTAGs), which were introduced by

Rambow (1994), further extend this by allowing at most one initial tree to be a member of the set of trees.

The term component used in this section is unrelated to the term component from Definition 7.17, we trust this will not lead to confusion.

Definition 10.25 A Vector Multi Component Tree Adjoining Grammar (VMCTAG) is a tuple of the form $\langle \Sigma, N, I, A, \mathcal{I}, \mathcal{A}, S \rangle$, where

$\langle \Sigma, N, I, A, S \rangle$ is a TAG

$\mathcal{A} \subseteq \wp(A) \setminus \emptyset$

$\mathcal{I} \subseteq \{T\} \cup \wp(A)$ where $T \in I$

An VMCTAG is called lexicalized if every $T \in \mathcal{A} \cup \mathcal{I}$ has an $A \in T$ which has a leaf marked with a terminal symbol. It is called 1-lexicalized if exactly one $A \in T$ has exactly one leaf marked with a terminal symbol.

If \mathcal{I} contains only singleton sets, we call the grammar a Multi Component Tree Adjoining Grammar (MCTAG).

An MCTAG is called tree local if adjunction of a set of auxiliary trees is restricted to take place on a single initial tree.

An MCTAG is called set local if adjunction of a set of auxiliary trees is restricted to take place on a single initial tree or inside a single element of \mathcal{A} .

An MCTAG is called non local if there is no restriction of the adjunction of auxiliary trees.

Depending on the restriction we impose on the adjunction operation for MCTAGs, we generate different languages.

For tree local MCTAGs, the class of languages we generate is strongly equivalent to the class of languages for TAGs.

For set local MCTAGs, Weir (1988) shows we still have a mildly context sensitive and polynomially parsable formalism, which is weakly equivalent to several other interesting grammar formalisms, like Linear Context Free Rewrite Systems (Weir 1988), Multiple Context Free Grammars (Seki, Matsumura, Fujii & Kasami 1991) and Stabler's (1997) derivational minimalism.

Finally, Rambow (1994) shows that non local MCTAGs and VMCTAGs are NP complete.

Example 10.26 Figure 10.19 on the following page shows an example of a vector multi component lexical entry for the quantifier 'everyone', consisting of one initial tree and one auxiliary tree. Figure 10.20 on the next page shows how these lexical components would combine with 'ate sushi' to derive 'Everyone ate sushi'.

From a proof net perspective, there is not much difference between the different multi component systems of Definition 10.25, but as usual we restrict ourselves to uniquely lexicalized systems, (V)MCTAG1.

Combining two components into a single abstract proof structure is done as shown in Figure 10.21 on page 199. The node marked A is the main leaf of

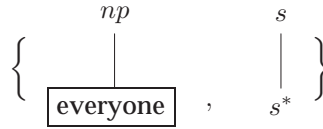


Figure 10.19: A multi component lexical entry

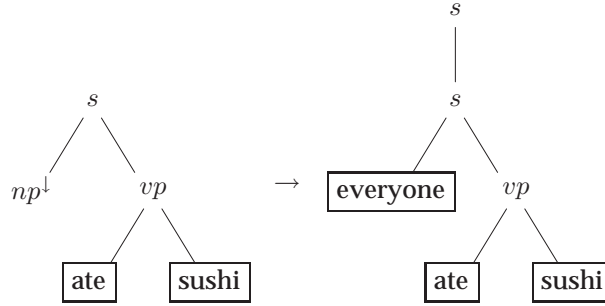


Figure 10.20: A derivation of 'Everyone ate sushi'

$\tau\beta$, i.e. it is the main vertex of the link above it. We can choose an arbitrary leaf of β to be the main leaf.

If β is an auxiliary tree and the node marked A is its foot node, then mode m is equal to 1. Otherwise, mode m is equal to 0.

The structural conversions for mode c are shown in Table 10.8. They are essentially the right extraction/right infixation rules proposed by Moortgat (1999), with the addition of the structural rules $[Pr0]$ and $[Pr0']$. Note that structural rules $[Pr1]$ and $[Pr2]$ are equivalent up to the names of the unary modes with structural rules $[P1]$ and $[P2]$ from Section 7.2.

This package of structural rules has the property that the $\langle \Gamma \rangle^c$ constituent can move into and out of every structural configuration of unary and binary mode 0.

Now it is relatively simple to see that whenever we would produce a proof net using both separate abstract proof structures, we can use the com-

| | |
|---|--|
| $\frac{\Gamma[\langle \Delta_1 \circ_0 \langle \Delta_2 \rangle^c \rangle^0] \vdash C}{\Gamma[\langle \Delta_1 \rangle^0 \circ_0 \langle \Delta_2 \rangle^c] \vdash C} [Pr0]$ | $\frac{\Gamma[\langle \Delta_1 \rangle^0 \circ_0 \langle \Delta_2 \rangle^c] \vdash C}{\Gamma[\langle \Delta_1 \circ_0 \langle \Delta_2 \rangle^c \rangle^0] \vdash C} [Pr0']$ |
| $\frac{\Gamma[\Delta_1 \circ_0 (\Delta_2 \circ_0 \langle \Delta_3 \rangle^c)] \vdash C}{\Gamma[(\Delta_1 \circ_0 \Delta_2) \circ_0 \langle \Delta_3 \rangle^c] \vdash C} [Pr1]$ | $\frac{\Gamma[(\Delta_1 \circ_0 \Delta_2) \circ_0 \langle \Delta_3 \rangle^c] \vdash C}{\Gamma[\Delta_1 \circ_0 (\Delta_2 \circ_0 \langle \Delta_3 \rangle^c)] \vdash C} [Pr1']$ |
| $\frac{\Gamma[(\Delta_1 \circ_0 \langle \Delta_3 \rangle^c) \circ_0 \Delta_2] \vdash C}{\Gamma[(\Delta_1 \circ_0 \Delta_2) \circ_0 \langle \Delta_3 \rangle^c] \vdash C} [Pr2]$ | $\frac{\Gamma[(\Delta_1 \circ_0 \Delta_2) \circ_0 \langle \Delta_3 \rangle^c] \vdash C}{\Gamma[(\Delta_1 \circ_0 \langle \Delta_3 \rangle^c) \circ_0 \Delta_2] \vdash C} [Pr2']$ |

Table 10.8: Structural rules for mode c

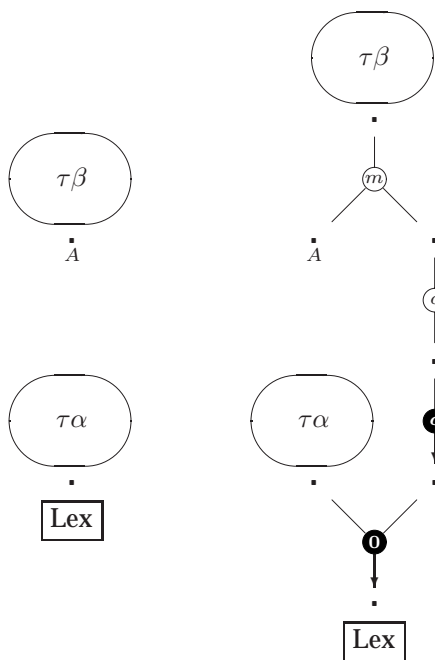


Figure 10.21: Combining two component abstract proof structures

bined aps in its place, adding an extra structural conversion in case $m = 1$ and converting $\tau\alpha$ and $\tau\beta$ to hypothesis trees as before. We only have to show we can contract the two par links we used to attach $\tau\alpha$ to $\tau\beta$. We do this by applying conversions $[Pr0]$, $[Pr1]$ and $[Pr2]$ to move the $\langle \rangle^c$ constituent first up as far as necessary, then applying the $[Pr0']$, $[Pr1']$ and $[Pr2']$ to move it down to the leaf of $\tau\alpha$ after which we can contract the $[L\Diamond_c]$ and the $[L\bullet_0]$ link.

10.6 Discussion

We have presented a way of embedding various formulations of the TAG formalism into multimodal proof nets. The resulting embedding gives us a new polynomial fragment of the multimodal Lambek calculus, with a fixed set of structural rules and a restricted use of par links.

An interesting line of research would be how could extend the translation to remove or relax the formula restrictions while maintaining the polynomial time parsing complexity of the fragment. For this, it would be useful to make use of the automata models which have been proposed for TAGs, see for example (Joshi & Schabes 1996, Rambow 1994), and show how to simulate proof search for fragments of $NL\Diamond_{\mathcal{R}}$ directly in the automata models.

TROUGHOUT this book, we have seen how proof nets in their various forms can be used as a logical tool for the description of linguistic analyses. The main advantage of proof nets over other formulations of the multimodal Lambek calculus, such as natural deduction or sequent calculi, is that proof nets are inherently redundancy free, that is, different proof nets correspond to different linguistic objects, they are not merely different representations of the same linguistic object. In this sense, proof nets capture the ‘essence’ of proofs.

This property is not merely conceptually attractive, but also offers computational advantages in that we do not need to formulate procedural restrictions on the order of rule applications in the system to avoid generating duplicate solutions, or worse, *check* every time we find a solution to see if it is a new solution or a syntactic variant of an old solution.

We have given proof net calculi for **MLL**, **L_ε**, **MLL1** and finally for **NL_{◇ \mathcal{R}}** and looked at correctness criteria for these calculi in the form of graph contractions, switchings and labeling. We have also given several possible linguistic applications of these calculi.

The logic **NL_{◇ \mathcal{R}}** , because it contains a variable structural rule component \mathcal{R} , was given a modular correctness criterion where the structural rules of \mathcal{R} correspond to graph rewrite rules and where the logical rules correspond to special cases of the graph contractions for **MLL**. We have given an algorithm for checking this contraction criterion and analyzed its complexity.

Finally, we have presented the Grail grammar development tool, which is based on this algorithm and which includes many of the heuristics we discussed in Chapter 8. This system has been used as courseware for introductory to advanced level courses in linguistics and artificial intelligence.

11.1 Further Research

Some questions were left open during throughout the chapters of this book.

First of all, though PSPACE is a good upper bound on the complexity for multimodal Lambek calculi, a similar characterization of NP complete and polynomial fragments would be useful. I have suggested the use of special cases of the contraction criterion for NP decidability and of extensions of the LTAG translation for polynomial decidability.

Instead of restricting the logic, another approach would be to *extend* the logic with operators like second order quantifiers or a contraction modality. We have suggested some uses for these connectives in Chapter 2, though it is still unclear if we can give an account of the linguistic phenomena discussed there without using the exponentials or second order quantifiers. Adding these connectives to the logic will be a delicate undertaking, since it can have a devastating effect on the complexity of the resulting logic. Proposals for decidable extensions of the Lambek calculus with a contraction modality or second order quantifiers are given by Jäger (2001) and Emms (1993*b*) respectively. Also, it would be interesting to see what proof nets for these logics would look like.

Finally, not all of the algorithmic improvements discussed in Chapter 8 have been implemented in the Grail automated theorem prover we will discuss in Appendix A. Measuring the effect of the different heuristics on performance and discovering other, more powerful heuristics will be useful for practical applications of the system.

APPENDIX A

THE GRAIL THEOREM PROVER

THIS appendix gives an overview of the Grail system, developed as part of my PhD project, and its use as a tool for the development and prototyping of grammar fragments for the multimodal Lambek calculus.

Grail is an automated theorem prover based on proof nets and algebraic labeling, a combination discussed in Chapter 6. The theorem prover is implemented in SICStus Prolog, the user interface in TclTk.

Though the underlying logic, with a minor restriction on the structural rules, is decidable, and the theorem prover can operate automatically, user guidance is often desirable during the proof search. It can increase the performance of the algorithm and, more importantly, help the user visualize the status of the proof attempt thereby showing *why* a given statement is provable or not.

The Grail user interface is based on the Prolog debugger. At each proof step the user can take one of the following actions: *select* allows the user to select an inference step, *leap* performs automatic proof search until a proof is found, *fail* marks the current branch of the search tree as unsuccessful and *abort* abandons the entire proof attempt.

In my experience, the interface gives users better insight in the operation of the theorem prover and greatly enhances its facilities for prototyping and debugging of fragments of the multimodal Lambek calculus.

A.1 History

In the end of 1995, the first incarnation of Grail was a piece of Prolog code of some 250 lines. You could enter a logical statement and wait until it produced an answer in the form of `yes` or `no`, or until you got bored (which happened

a lot those days). In spite of a number of improvements to the efficiency of the original code, several grammar fragments designed in it could not handle longer sentences in a reasonable amount of time.

I therefore tried to give a user-friendly representation of the computation state which the user can inspect to guide the computation. The benefits of this are twofold: firstly, the user can select a promising continuation from the possible ones and abandon hopeless subgoals, and secondly, it gives the user insight into *why* specific statements are underivable, without having to use the Prolog debugger where tracing the execution is difficult even for the programmer.

The current version is some 8000 lines of mixed Prolog and TclTk code, using the TclTk library included with SICStus Prolog. It can be used without knowledge of Prolog and produces output in human-friendly natural deduction format.

Grail is used as a research tool and as courseware for introductory to advanced level courses in Lambek grammars. Grail is free software distributed under the GNU General Public License, and can be downloaded as source code and binaries from my personal home page.

<http://www.let.uu.nl/~Richard.Moot/personal/grail.html>

A.2 Tutorial

Before I give an in-depth overview of all possibilities at the different windows in Grail, it is perhaps useful to give a short introduction to designing and running grammar fragments in Grail.

A.2.1 Getting Started

You start Grail from the directory where you installed the source code or the binaries by giving the follow command.

```
sicstus -l grail
```

This will start Grail and, if SICStus and TclTk are correctly installed on your system, the window shown in Figure A.1 will appear.

This is the main window, where you activate the theorem prover and open new windows to edit the current grammar fragment. See Section A.3.1 for an overview of all options here.

A.2.2 The Lexicon

Since we start with an empty grammar fragment, we will first fill the lexicon with a few useful words. From the main window, select [Window/Lexicon Window] to open the lexicon window. The lexicon, being empty, does not

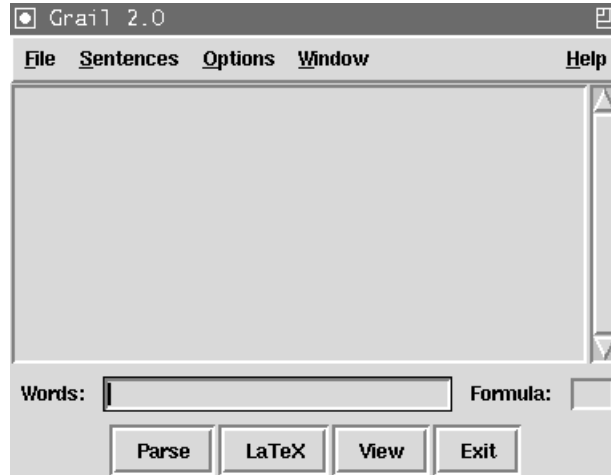


Figure A.1: The Grail startup window.

look very interesting right now, so select [Edit/New Entry...] from the menu bar of the lexicon window. The following window should appear.

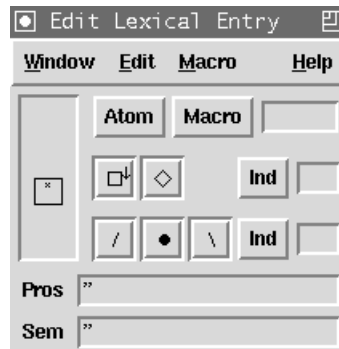


Figure A.2: The lexicon edit window.

Lexical Entries A lexical entry in Grail consists of three things: a prosodic entry, representing the word being described, a formula and a semantic entry, which encodes the lambda term meaning of the current entry. The entire top part of the lexicon edit window is dedicated to entering the formula.

Simple Formulas We start by assigning a simple np formula to a word in the lexicon. Because no atomic formulas have been defined for our fragment yet, we type np , followed by $\langle \text{Enter} \rangle$ in the text entry field next to the

[Atom] and [Macro] selection fields. You can click on [Atom] to verify that np has been added as an atomic formula. Enter 'tony' in both the Pros and the Sem entry field. The lexical edit window should now look as shown in Figure A.3.



Figure A.3: The lexicon edit window after typing in the first lexical entry.

Storing Entries It is important to note that the edits in the lexical edit window functions will only affect the lexicon once you select [Edit/Store Entry] from the menu. If you select [Edit/Store Entry] from the menu now, you will see the entry for 'tony' appear in the lexicon window.

Exercise 1 Add some more np 's to the lexicon by editing the Pros and Sem fields followed by [Edit/Store Entry] until the lexicon looks as in Figure A.4 on the next page.

Complex Formulas Now, we will add some entries with complex formulas to the lexicon, assigning the formula $(np \setminus_a s) /_a np$ to 'shot'. First, erase the previous lexical entry by selecting [Edit/Clear Entry] from the menu, then type 'shot' in both the Pros and Sem fields and 'a' in the index field next to the binary connectives as shown in Figure A.5 on the facing page.

Formulas are entered top-down left-to-right, always starting with the main connective of the current (sub)formula, and always specifying the left subformula before the right subformula. For $(np \setminus_a s) /_a np$ the main connective is $/_a$, so we proceed by pressing the [/] button. The result is shown in Figure A.6 on page 208.

We proceed with the left subformula, which is the complex formula $np \setminus_a s$ with main connective \setminus_a . The correct index a should still be in the index field, if not, reenter it or select it from the [Index] menu next to it. Pressing the [\] button now should result in Figure A.7 on page 208.

We've entered all connectives now and we only have to fill in the atomic formulas. We can do this by entering them in the atom field and pressing



Figure A.4: The lexicon after inputting some np's.

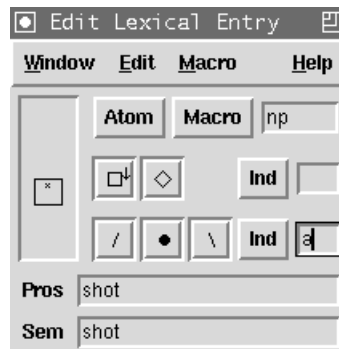
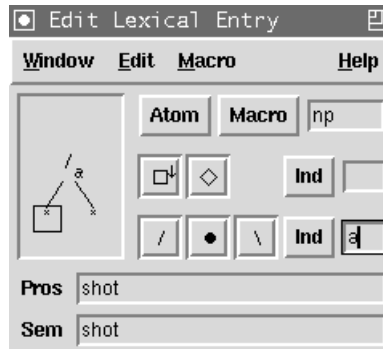
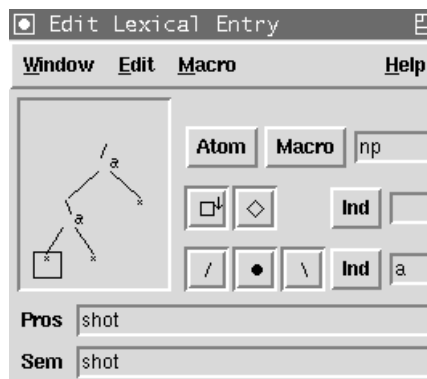


Figure A.5: The edit window after entering the mode information.

<Enter> or by selecting them from the [Atom] menu if we've entered them before. After selecting 'np', entering 's' and selecting 'np' again, our formula looks as shown in Figure A.8 on page 209 and we can select [Edit/Store Entry] to store it in the lexicon.

Exercise 2 Add entries for 'likes', 'hates' and 'distrusts' to the lexicon.

The Macro Facilities You can mark any subformula of the current lexical entry by clicking it and give it a name to use later. For example, we can click on the main connective $/_a$ of the lexical entry for 'shot' to select the entire $(np \backslash_a) /_a np$ formula, we can give it the abbreviation 'tv', for transitive verb,

Figure A.6: The edit window after entering $/a$.Figure A.7: The edit window after entering $\backslash a$.

by entering this in the atom field and selecting [Macro/Store Selection as Macro] from the menu. We can also click on the connective \backslash_a to select the formula $np\backslash_a a$ and store it as iv for intransitive verb, again by entering this in the atom field and selecting [Macro/Store Selection as Macro].

We can now enter more complex entries quite simply. Suppose we want to assign the word 'himself' the formula $((np\backslash_a s)/_a np)\backslash_a (np\backslash_a s)$. With the macro's we just stored this is equivalent to $tv\backslash_a iv$, so we can press [\backslash], select tv from the [Macro] menu, then select iv from the macro menu and we have entered the correct formula. After storing it, the lexicon should look as shown in Figure A.9 on the facing page.

The macro's are also the simplest way of producing complex goal formulas for use in the main window.

Exercise 3 Give 'someone' an entry of the form $s/_a iv$. Store this entry as a macro for $gq-s$, a subject generalized quantifier.

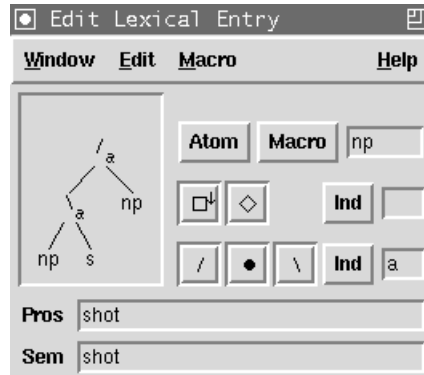


Figure A.8: The edit window after entering all atomic formulas.

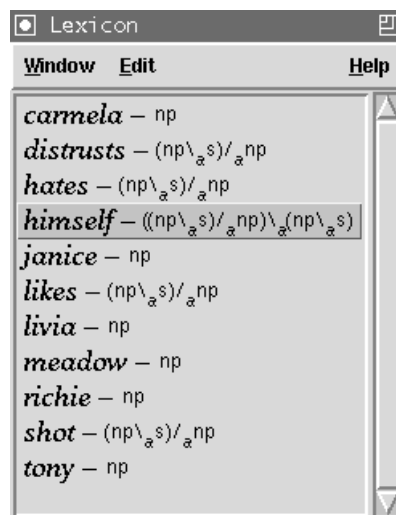


Figure A.9: The lexicon window after entering some complex formulas.

Exercise 4 Use the macro from the previous exercise to assign a lexical entries of the form $\langle gq_s/a \rangle$ to $\langle a \rangle$ and $\langle every \rangle$.

Correcting Mistakes When you notice you have made a mistake when entering the current formula there are several ways of correcting it.

First of all, if you want to erase the lexical entry window completely, you can select [Edit/Clear Entry] from the menu.

If you want to erase the current formula, or a part of it, while keeping the [Pros] and [Sem] fields intact, you can select the root of the formula tree you want to erase and select [Edit/Cut] from the menu or press (Control-

k) on the keyboard. This will replace the currently selected formula tree by an insertion point. Selecting [Edit/Paste] or pressing (Control-y) while you have selected an insertion point will paste the formula you have cut at the insertion point. Selecting [Edit/Copy] or pressing (Control-c) will store the selected formula for the next paste operation without deleting anything.

Finally you may want to replace a connective with a different connective, this is done simply by selecting the connective and pressing one of the connective buttons. Replacing ‘/’ by ‘\’ or vice versa will also switch the order of the subformulas, all other replacements will leave the order of the subformulas unchanged.

It is not possible to replace an atom with a different atom or a connective in this way: you have to erase this explicitly with [Edit/Cut] or (Control-k).

If you notice that you have stored an incorrect entry into the lexicon and you wish to correct it, the best way to proceed is to double-click on the lexical entry or to click on it followed by selecting [Edit/Edit Entry...] from the lexicon menu. This will open the incorrect entry into the lexical edit window. Then, select [Edit/Delete Entry] from the lexicon menu or press (Control-Button 1) to delete the incorrect entry from the lexicon, and proceed by editing the incorrect entry in the lexical edit window.

Exercise 5 Use the edit facilities to change a transitive verb from the lexicon to produce an entry for ‘talks’ of the form $(np \backslash_a s) /_a pp$ and an entry for ‘needs’ of the form $(np \backslash_a s) /_a ((s /_a np) \backslash_a s)$.

Exercise 6 Edit the lexical assignment to ‘someone’ from Exercise 3 to produce an additional lexical entry of the form $(s /_a np) \backslash_a s$. Store this formula as a macro for gq_{-o} , an object generalized quantifier. Use this macro to assign appropriate new lexical entries to ‘a’ and ‘every’ as well.

Loading and Saving Your Fragment Now that we have a simple lexicon, it is time to put the theorem prover to work, but before we do that, we first save the grammar fragment we have produced so far to a file. Select [File/Save Fragment...] from the menu of the main window, then type in a file name in the entry field and press the [Save] button.

To load this fragment again from a new Grail session select [File/Consult Fragment...] from the menu, select the file from the file select window, then press the [Open] button.

Saving or loading a file will change the name of the main Grail window to the name of the current fragment file.

A.2.3 The Theorem Prover

Entering a New Sentence From the main window, type a sentence in the entry field marked ‘words’ and a formula in the entry marked ‘formula’. The sentence can be any string.

Lexical Lookup By default, Grail does not distinguish between upper and lower case and all interpunction symbols are treated as spaces. The reference guide details how to modify this standard behavior. Press <Enter>, select [Sentences/Parse] from the main window menu or press the [Parse] button to start the theorem prover. The status window will now appear.

Any string between spaces is treated as a word and will be looked up in the lexicon. If a word has no lexical entries at all, Grail will complain and abort the attempt. Check the lexicon and your spelling if this happens.

Proof Status If lexical lookup succeeds, the status window will appear to give updates on the status of the proof attempt and show a rough estimate of the computation remaining for the current lookup.

While the theorem prover is running, it is not possible to edit the lexicon or any other aspect of the current grammar. However, if you get tired of waiting, you can press the [Abort] button from the status window or use (Control-c).

After the theorem prover has completed its computations, the status window will have one of the following messages.

[Done] One or more solutions were found.

[Failed] No solutions were found.

[Aborted] User aborted the computation.

The sentence will now be added to the list of sentences in the main window. If no solutions were found before the user aborted the computation, the sentence will have a '?' prefixed to it. If no solutions we found even though the computation finished, the sentence will have a '*' prefixed to it. Otherwise, the sentence will have a space as its first symbol. Keep in mind that Grail only remembers the last proof attempt for a sentence: retrying after changing the grammar fragment can result in new derivability markings.

You can retry a sentence without entering it again by double-clicking on it, or by selecting it and the pressing [Parse], the <Enter> button or selecting [Edit/Parse] from the menu.

Exercise 7 Enter the sentences shown in Figure A.10 on the next page in the main window. The goal formula in all these cases is s .

Proofs If you have \LaTeX installed on your computer, Grail can produce natural deduction output for any proofs it has found. Pressing the [View] button will cause your selected \LaTeX previewer to appear with one or more natural deduction proofs for the last successfully parsed sentence. For the different types of natural deduction output format, we will refer to the reference manual. A typical natural deduction proof produced by Grail is shown in Figure A.11 on the following page.

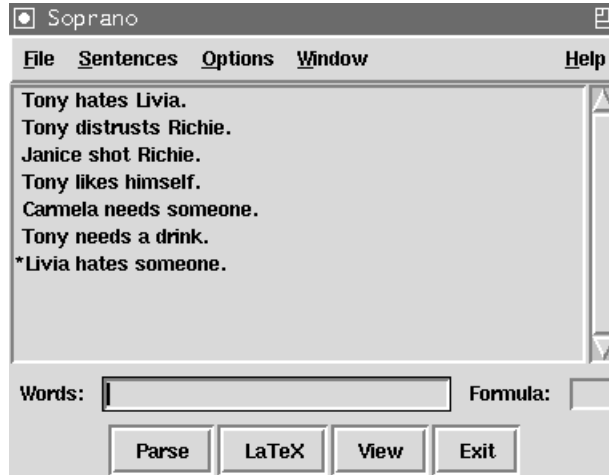


Figure A.10: The main window after entering some sentences.

$$\frac{\text{tony} \vdash np \quad \frac{\text{distrusts} \vdash (np \backslash_a s) /_a np \quad \text{richie} \vdash np}{\text{distrusts} \circ_a \text{richie} \vdash np \backslash_a s} [/E]}{\text{tony} \circ_a (\text{distrusts} \circ_a \text{richie}) \vdash s} [\backslash E]$$

1. ((distrusts richie) tony)

Figure A.11: L^AT_EX natural deduction output

Debug Mode There can be cases where you want to get more detail as to why a certain sentence was underivable in the current fragment, or — equally important — to guide the theorem prover to a proof which would take too long to find without guidance. For these situations, Grail has a debug mode, which you can activate by selecting [Debug/Interactive] from the status window. This will cause the status window to expand to the proof net window.

Double-click on the sentence ‘Livia hates someone’. Grail has marked it as underivable and we want to know why. The proof net window should look like shown in Figure A.12 on the next page.

The proof net window displays the formulas in a way similar to the lexical edit window, with the following differences.

- the main connective of a formula is at the bottom,
- positive atomic formulas are drawn in white, negative atomic formulas are drawn in black,
- par links are drawn with dotted lines.

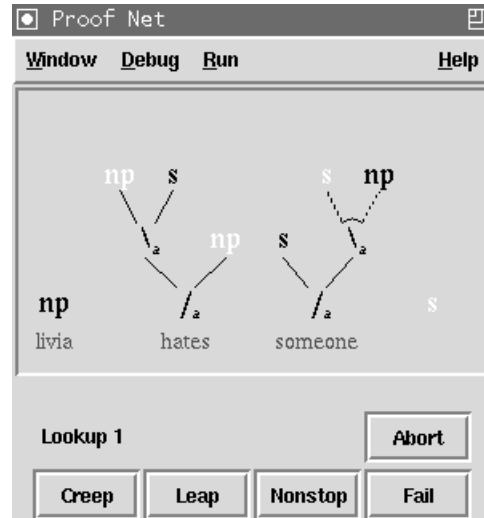


Figure A.12: The proof net window for ‘Livia hates someone’.

The links used in the proof net window are essentially the same as those we used for the Lambek calculus in Section 4.7. This makes it easy to identify proof nets which need some form of the commutativity rule, as those proof nets will have crossing axiom links.

The lookup shown in Figure A.12 uses the subject generalized quantifier type for ‘someone’, which seems unlikely to be correct. We can press the [Fail] button to reject this lookup and force Grail to find new formula assignments to the words of the sentence. Using [Fail] carelessly can lead to Grail failing to find proofs which it normally would have found; selecting [Fail] means *you* take responsibility for the absence of proofs in the current branch of the search space. After pressing [Fail], Grail returns with the second lexical lookup and the proof net window looks as shown in Figure A.13 on the next page.

The second lookup uses the object generalized quantifier type for ‘someone’, which appears the right choice in the current situation.

We can now select [Nonstop], which causes Grail to continue until it has found all proofs for the current sentence. However, because we already know the current proof attempt will fail, this is not the right choice.

We can also select [Leap], which causes Grail to continue until either.

- it has found a complete linking of all axioms,
- or it failed to produce a complete linking of all axioms at which point Grail will try to find new lexical assignments and, if successful, wait for your input again.

Selecting [Creep] takes us through the axiom links one at a time, though,

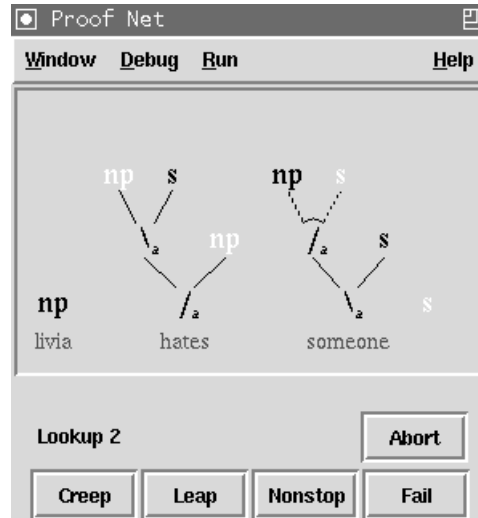


Figure A.13: The proof net window for 'Livia hates someone'.

by selecting [Fail] we can at any point mark the current linking as unsuccessful and continue with the next untried axiom link.

Manual Axiom Links The final possibility is to perform the axiom links manually. This is especially recommended in the case of larger proof nets. To make an axiom link manually, click one of the atomic formulas in the proof net window. Start with the leftmost, negative *np* which corresponds to 'Livia'. After clicking this *np* all possible positive *np*'s you can link it to will be marked by a white box, as shown in Figure A.14 on the facing page.

Select the leftmost positive *np*. An axiom link connecting the two *np*'s will now appear. Grail will keep track of the other possibility for this axiom link for you, so you don't have to worry about mistakes. If at any point you produce an incorrect proof structure, for example by creating a cycle, Grail will complain and ask you to retry the last choice you made where alternatives were available.

If you are very sure there is only one correct way of linking the current formula, you can use <Shift> in combination with the left mouse button. This will commit you to the current axiom link. It is equivalent to selecting all other possible axiom links first and immediately following them by [Fail]. Be careful that this may prevent proofs from being found.

Exercise 8 *Connect all axiomatic formulas until the proof structure looks as shown in Figure A.15 on the next page.*

After all axiom links have been made, you are given a final opportunity

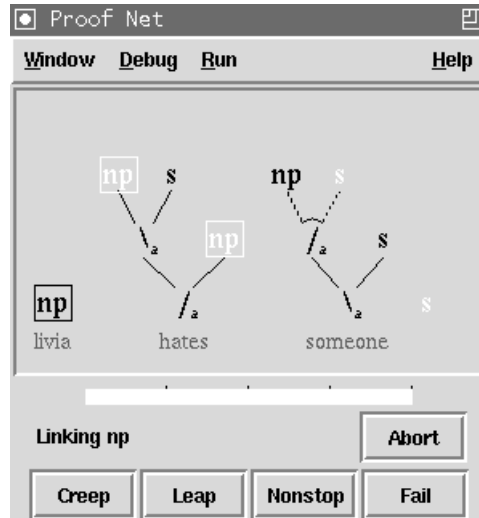
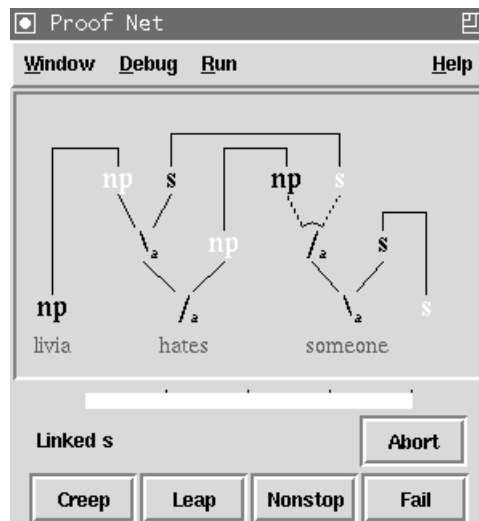
Figure A.14: Possibilities for linking the first *np*.

Figure A.15: Proof net after making all axiom links.

press [Fail] and try to find another linking. Pressing [Creep] or [Leap], however will take you to the rewrite window.

Rewriting The rewrite window contains the label computed for the current proof structure. Section 6.3 explains how the labels are obtained from acyclic

and connected proof structures. For the current proof structure, the label looks as shown in Figure A.16.

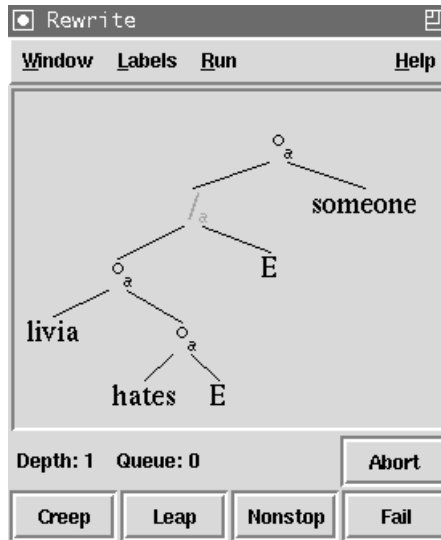


Figure A.16: Label for 'Livia hates someone'.

In order to produce a correct label, we have to do two things.

- remove all auxiliary constructors by means of their rewrite rules. For '/' we have to use the conversion shown in Figure A.17. The conversions for the other connectives as shown in Figure A.28 on page 229,
- left to right traversal to the label tree should produce the words in the order they appear in the input sentence.

In this case, the words are already in the correct order, but it is impossible at the moment to use the '/' conversion because we need to be in the configuration shown in Figure A.18 on the next page for that.

You can check for yourself that we cannot produce a correct label. [Leap] or [Creep] will not succeed and Grail will continue by trying to find a new

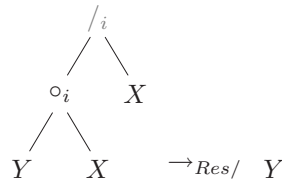


Figure A.17: Conversion for '/'.

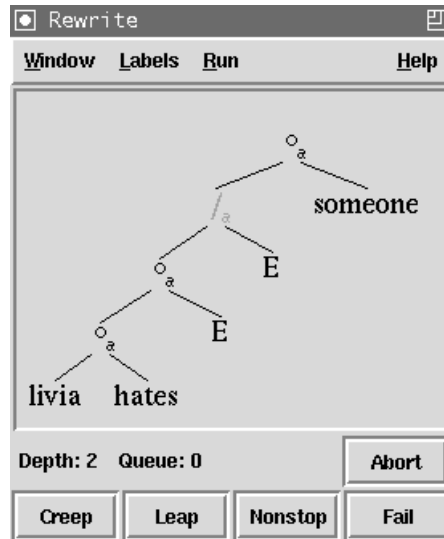


Figure A.18: Label we need for the ‘/’ conversion.

axiom link. You can also click on every node of the tree to see a list pop up which shows which conversions are applicable at this node and see this list is empty for all nodes in the tree.

When you find a sentence which isn’t derivable, while you would want it to be, you can

- either add a new entry to the lexicon for one of the words in the sentence or correct or modify an old entry,
- or add a new structural rule to your grammar.

The first option seems unattractive in this case; we already have two assignments for the quantifier ‘someone’ and assigning a new formula for every new construction we encounter would lead to a huge lexicon. So in this case, we would like to generalize a bit by adding some structural rules.

A.2.4 The Structural Postulates

Editing a Postulate You open the postulate window from the main window by selecting [Window/Postulate Window] from the menu or by pressing (Control-p). Initially, your grammar fragment will have no structural postulates. But you can create a postulate by selecting [Edit/New Postulate] from the postulate window to make the postulate edit window appear, which looks as shown in Figure A.19 on the following page.

The edit postulate window is similar in structure to the edit lexical entry window, only now we only have ‘•’ and ‘◇’ as connectives and instead of atomic formulas we have structural variables.

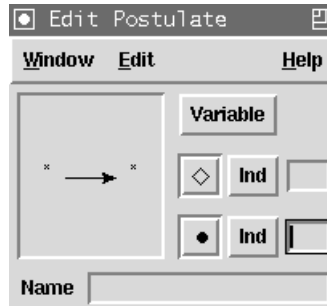


Figure A.19: Creating a new postulate.

To add a structural rule for associativity to the current grammar, first select a from the [Ind] menu next to the [\bullet] button. Then click on the $*$ on the left hand side of the arrow and press the [\bullet] button twice. The edit postulate window should now look as shown in Figure A.20.

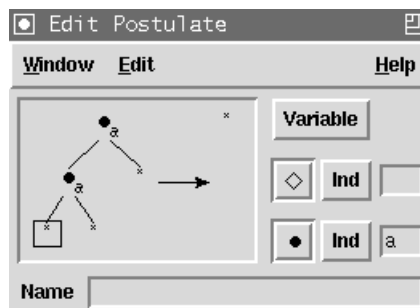


Figure A.20: Entering the left hand side of the postulate.

All that remains is to create the leaves. Select the [Variable] menu. Only one variable 'A' is available initially, so select it. Now, since 'A' has been used, a new variable 'B' will be added to the menu. Select this. Finally, select 'C' from the variable menu. We have now completed the left hand side of the structural rule.

Exercise 9 Complete the right hand side of the postulate, so that it the postulate looks as shown in Figure A.21 on the facing page.

Storing a Postulate Like the changes you make in the lexical edit menu, the changes in the postulate edit menu don't take effect until you select [Edit/Store Postulate] from the menu.

Before storing the postulate, let's give it a name by typing 'Ass' in the Name field. We have three possibilities for storing a postulate: we can store

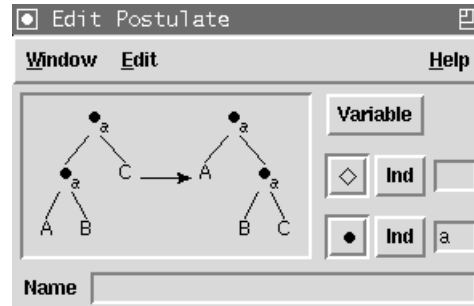


Figure A.21: Completed postulate.

the left-to-right version — which is the default —, we can store the right-to-left version or we can store both. We can toggle between these possibilities by clicking on the arrow. To save time, click on the arrow once then select [Edit/Store Postulate] store both versions of this postulate. The postulate window should now look as shown in Figure A.22.

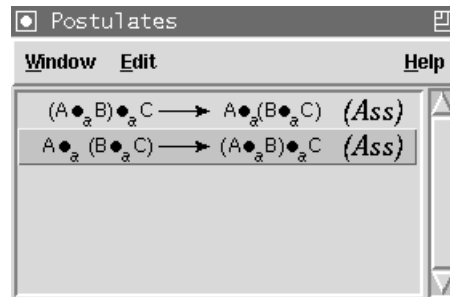


Figure A.22: Two postulates for associativity.

Note that Grail will complain if one of the following holds.

- there are multiple occurrences of a variable on either side of the postulate arrow,
- a variable occurs on only one side of the postulate arrow,
- a postulate has more occurrences of unary connectives on the left hand side than on the right hand side.

In the last case, you can overrule Grail's complaints and store the postulate anyway, though this might lead to nontermination of Grail's proof search mechanism.

Finally, if you try to store a postulate which is already a consequence of other postulates in you fragment, Grail will notify you of this.

Editing a Postulate To edit an existing postulate, you basically have the same options as for editing a lexical entry. You can cut, copy and paste by using [Edit/Cut], [Edit/Copy] and [Edit/Paste] respectively.

You can delete a postulate by selecting it and using [Edit/Delete Postulate] from the postulate window. A final option is to disable postulates. This makes a postulate unavailable without actually deleting it and is a way of experimenting with different sets of structural postulates to see which structural rules you really need. You can disable a postulate from the postulate window by clicking on the middle mouse button or by selecting [Edit/Disable/Enable Postulate] from the menu.

Rewriting Now that we have added the structural postulates for associativity to the grammar, we can see if this finally makes the sentence 'Livia hates someone' derivable. Double-click again on this sentence from the main menu, select [Fail] after the first lexical lookup, then select [Leap] to generate the first acyclic, connected proof structure for this lookup and the rewrite window will appear, looking exactly as before in Figure A.16 on page 216.

Now, however when we select the ' \circ_a ' node just below the '/' node, the popup menu will display we have the option here to use the 'Ass' structural rule we have just added. We select this from the menu, to produce the label shown in Figure A.18 on page 217. Grail automatically keeps track of all alternatives, and if we change our mind about the current rewrite, we can select [Run/Undo!] from the menu or press <u>. If we select the '/' node from this configuration the popup menu will display 'Res', which indicates we can perform the residuation conversion for '/' and eliminate this node from the label, which gives the result shown in Figure A.23.

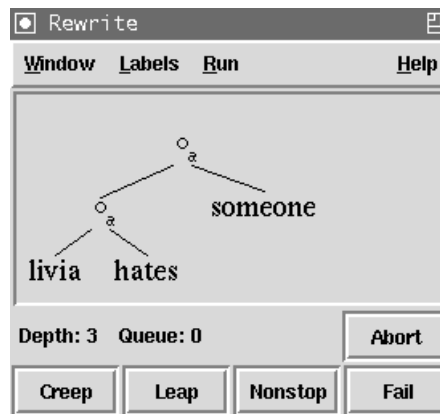


Figure A.23: Correct label for 'Livia hates someone'.

This is a correct label, but if you want you can apply an extra associativity step by selecting the top ' \circ_a ' node. When you are satisfied with the correct

label, press either [Creep] or [Leap] and Grail will start producing the \LaTeX output, then continue trying to find proofs for a different axiom linking.

A.3 Reference Guide

A.3.1 The Main Window

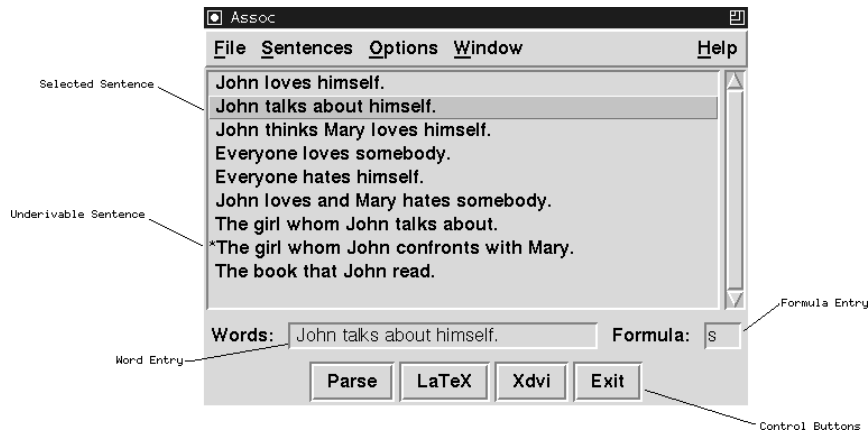


Figure A.24: The main window.

The main window is your interface to the theorem prover. From here you can parse grammatical expressions, load and save grammar fragments, and open other windows to view and edit these fragments.

The window will display a list of previously parsed sentences for the current fragment and an input section where you can enter new expressions.

Clicking on one of the sentences will display the words in the word entry section, and the goal formula in the formula entry section. You can edit the words and the formula, and parse the sentence by pressing (Enter) or the [Parse] button. Double clicking one of the sentences will parse it immediately.

When parsing, a status window will appear which gives an indication of the computations being performed, and when ready will display either 'done' or 'failed' depending on whether a proof was found. The [Abort] button cancels the computation when you run out of patience.

Tokenization Grail will tokenize an input sentence in the following way. First of all, the following characters are, by default, defined as interpunction characters and will be treated as spaces.

! " ' - . : ; ? `

Grail contains a Prolog hook you can use to override the default behaviour. If your fragment contains a declaration of the form

```
special_string(String,Atom).
```

Grail will tokenize the String which is given as the first argument of `special_string/2` as the Prolog atom given as the second argument. Examples would be the following.

```
special_string("?", '?').
special_string("can't", 'cannot').
```

Note that you still have to add lexical entries for '?' and 'cannot' if you want to use this in your fragments.

Currently, you can only add `special_string/2` declarations to your fragment by editing your file manually.

Command Buttons below the entry sections you will find the following command buttons.

[Parse] Parses the words in the input entry as a formula described in the formula entry. The output is sent to \LaTeX .

[LaTeX] Sends the results of the previous parse to \LaTeX ; this is useful if you have changed some of the output options.

[Xdvi] Sends the result of the previous parse to \LaTeX and displays them using the xdvi previewer.

[Exit] Exits the program.

Menu Bar from the menu bar of the main window, you can access the following operations.

[File]

[About] Prints information on the release date and version number.

[New Fragment] Starts a new grammar fragment from scratch. All previous information will be lost.

[Consult Fragment...] Loads a grammar fragment.

[Save Fragment...] Saves your fragment.

[Compile Prolog Source...] Compiles a Prolog file.

[Close] Iconifies the main window.

[Quit...] Hasta la vista, baby.

[Sentences]

[Clear Entry] Erases the words and formula.

[Parse] Parses the selected sentence.

[Delete] Deletes the selected sentence from the sentence list.

[Options]

[Prolog Messages] A choice between Quiet and Verbose. When set to Quiet, only some information about the time a computation takes will be sent to screen. When set to Verbose, a lot more information about the state of the computation will be printed. Defaults to Quiet.

[View Format] A selection of L^AT_EX output formats for the Grail natural deduction output. Current possibilities are 'none' (no L^AT_EX output, resulting in faster execution of Grail because it is unnecessary to keep track of the *path* to the solution), 'dvi', 'postscript' and 'pdf'. Defaults to dvi.

[Viewer Geometry] A choice of the window size of the L^AT_EX previewer. Possible values are 320x200, 640x400, 800x600, 1024x800 and 1280x1024. Defaults to 800x600. Note that some previewers ignore their geometry parameters.

[Natural Deduction Style] A choice between Prawitz style natural deduction and Fitch style natural deduction. Defaults to Prawitz.

[Proofs]

[Eta Long Proofs] When this checkbox is on, eta long natural deduction proofs will be produced. Defaults to off.

[Hypothesis Scope] When this checkbox is on, the scope of a hypothesis in Fitch style natural deduction will be indicated by a vertical bar. Defaults to on.

[Labels]

[Output Labels] When this checkbox is on, labeled deduction proofs will be produced. Defaults to on.

[Implicit Structural Rules] Hides structural rule applications.

[Collapsed Structural Rules] Successions of multiple structural rules will be collapsed into one.

[Explicit Structural Rules] Each structural rule is portrayed explicitly. This is the default setting.

[Formulas]

[Reduce Macros] When this checkbox is on, complex formulas will be reduced by the macro definitions. Defaults to off.

[Semantics]

[Output Semantics] When this checkbox is on, lambda term semantics will be printed with the formulas. Defaults to off.

[Functional Notation] Switches off the Montague-style notation conventions and displays complex function terms normally.

[Predicate Notation] Uses Montague-style notation conventions displaying a term like $((f\ y)\ x)$ as $f(x, y)$.

[Reduce Semantics] When this checkbox is on, lambda term reductions will be performed whenever possible. Defaults to off.

[Substitute Lexical Semantics] Formulas will be assigned their lexical meaning recipes instead of semantic variables. Defaults to off.

[Semantics For Unary Connectives] When this checkbox is off, the semantic constructors for the unary connectives will be ignored. Defaults to on.

[Colors...] Opens a color selection window allowing you to change the standard colors of the Grail application.

[Fonts...] Open a font selection window allowing you to change the font family, weight, slant, width and size. Font selection changes will only affect the proof net and the rewrite window.

[Save Current Options] Save the current settings for all options to the file `.grail.default.options.pl` which will be automatically loaded the next time you start Grail.

[Restore Default Options] Return the options to their initial state, as if you had just restarted Grail.

[Window]

[Status/Proof Net Window] Opens the status or proof net window, depending on whether debugging is turned on or off. See sections A.3.2 and A.3.3.

[Rewrite Window] Opens the rewrite window, only available when debugging is turned on. See section A.3.4.

[Lexicon Window] Opens the lexicon window. See section A.3.5.

[Postulate Window] Opens the postulate window. See section A.3.7.

[Analysis Window] Opens the analysis window. See section A.3.9.

[Help]

[On This Window] Gives a help message.

A.3.2 The Status Window

The status window gives information about the current state of the computation, and allows you to abort time-consuming parses. It will open automatically during proof search, or you can open it by selecting **[Window/Status Window]** or by typing (Control-s).

The white part of the status bar gives an estimate of the number of links which have not been tried yet.

The status message can be one of the following.

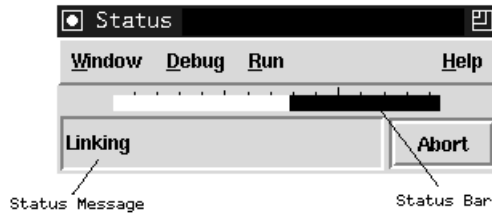


Figure A.25: The status window.

Initializing Garbage collecting, preprocessing.

Linking Performing axiom links.

Rewriting Performing label conversions.

Generating Output Generating L^AT_EX output, and sending it to a file.

Done Computation terminated, one or more derivations were found.

Failed Computation terminated, no derivations were found.

Aborted User got bored and pressed the [Abort] button. If derivations were found, they can still be viewed.

Menu Bar From the menu bar, we can select the following.

[Window]

[Close] Iconifies this window.

[Debug]

[Automatic] Debugging off. Grail will search for proofs without user guidance.

[Interactive] Switches on the interactive debugger.

A.3.3 The Proof Net Window

When the interactive debugger is on, the status window will be replaced by the proof net window. In the proof net window we see the current partial proof structure, with the decomposition trees of the formulas the current lookup assigns to the words from the sentence above the corresponding word. Positive atomic formulas are drawn in white and negative atomic formulas drawn in black. Here atomic formulas of opposite polarity are linked until we find a proof structure which is both acyclic and connected.

The console buttons offer the following options.

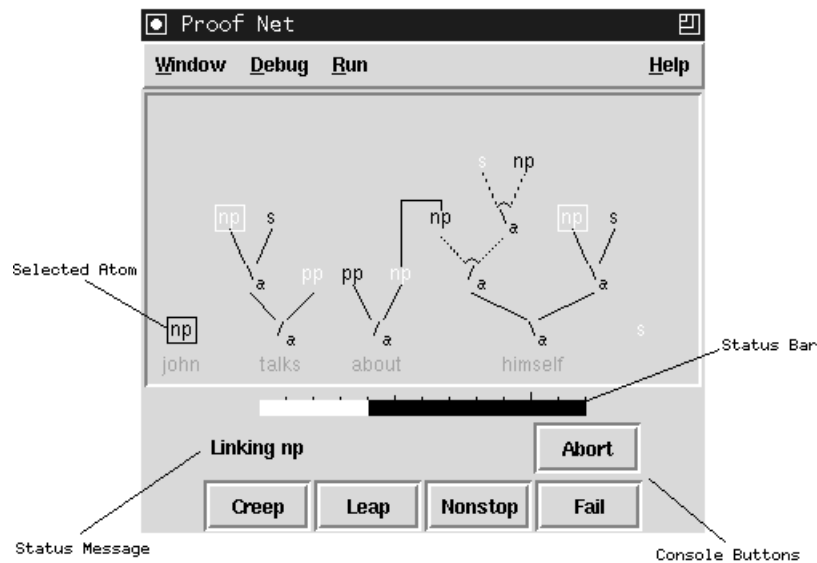


Figure A.26: The proof net window.

[Creep] Will perform the next step in the computation, then wait for interaction.

[Leap] Will return after a total linking for the current lookup has been found or to the next lookup if no such linking exists.

[Nonstop] Will perform the rest of the proof search automatically.

[Fail] Will abandon the current branch of the search space and continue with the next untried branch.

[Abort] Aborts the current proof attempt.

In addition, you can click on the atomic formulas themselves to have complete control over the order in which the axioms are linked. As a first step you select any atom not currently linked by an axiom link. The selected formula will then appear in a black box and the atoms of opposite polarity which have not been tried before will appear in a white box, as shown in Figure A.26. You can then click any of the boxed formulas to perform an axiom link.

In addition, if you know you are only interested in one specific choice of the possible axiom links, you can keep the (Shift) key depressed when you press the mouse button in order to *commit* yourself to a specific axiom

link. This is equivalent to first selecting every other possibility followed by pressing the [Fail] console button.

If at any time you perform a link which results in a cyclic or disconnected proof structure, you will get a message and the current link will fail.

Be warned that by selecting [Fail], [Abort] or using the commit option, you cut part of the search space and may miss valid proofs if you are not careful.

Menu Bar For the menu bar, we can select the following.

[Window]

[Save Postscript] Saves the current (partial) proof structure to a post-script file.

[Close] Iconifies this window.

[Debug]

[Automatic] Debugging off. Grail will search for proofs without user guidance.

[Interactive] Switches on the interactive debugger.

[Run] Setting this option to [Nonstop] will cause Grail perform all axiom links without user interaction. Defaults to [Creep].

A.3.4 The Rewrite Window

When the interactive debugger is on, you can open the rewrite window by selecting **[Window/Rewrite Window]** or by typing (Control-r).

The rewrite window (Figure A.27 on the following page) displays the current label and allows you to perform rewrite operations on this label. Clicking on a node of the label will cause a pop-up menu with the label conversions rooted at that node to appear. You can apply a conversion by selecting it from the menu. Any alternatives to your choice will be added to the queue.

The status message gives you an indication of the number of unvisited labels in the queue and of the current depth.

As shown in the figure, some label constructors are drawn in dark grey. These correspond to unsatisfied constraints, which are checked by the label conversions shown in Figure A.28.

For the non-associative base logic, these are all available conversions. However, you can relax the constraints by specifying your own structural postulates as specified in section A.3.7. Each structural postulate can be applied backwards as a label conversion.

You can rewrite a label until you reach one where all constraints have been satisfied and the words are in the order required by the input sentence. Grail will only check if you meet these conditions when you press the [Creep]

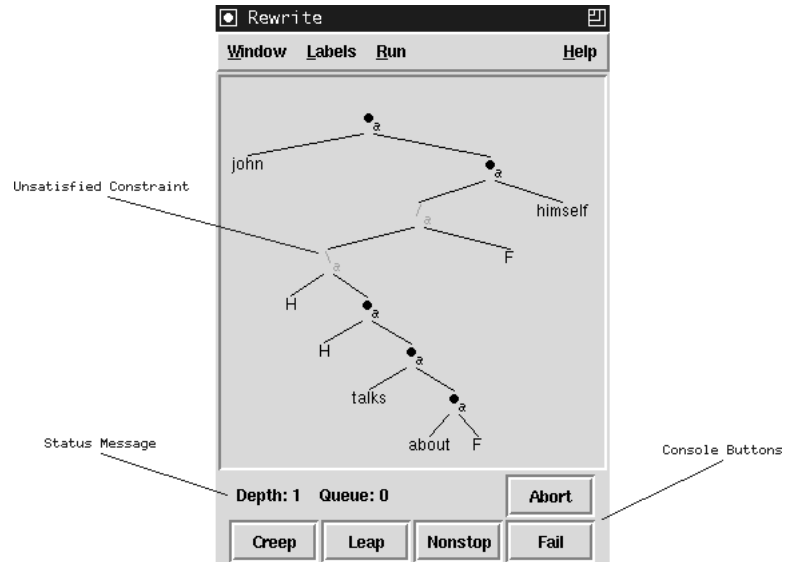


Figure A.27: The rewrite window.

or [Leap] button in order to allow you to continue rewriting a label even if all constraints have been satisfied.

The console buttons offer the following options.

[Creep] Will add all one step conversions from the current label to the back of the queue, then continue with the first element of the queue.

[Leap] Will return only after all label constraints have been satisfied.

[Nonstop] Will perform the rest of the proof search automatically.

[Fail] Will abandon the current branch of the search space and continue with the next item on the queue.

[Abort] Aborts the current proof attempt.

Menu Bar For the menu bar, we can select the following.

[Window]

[Postulate Window] Opens the postulate window.

[Save Postscript] Saves the current label to a postscript file.

[Close] Closes this window.

[Labels]

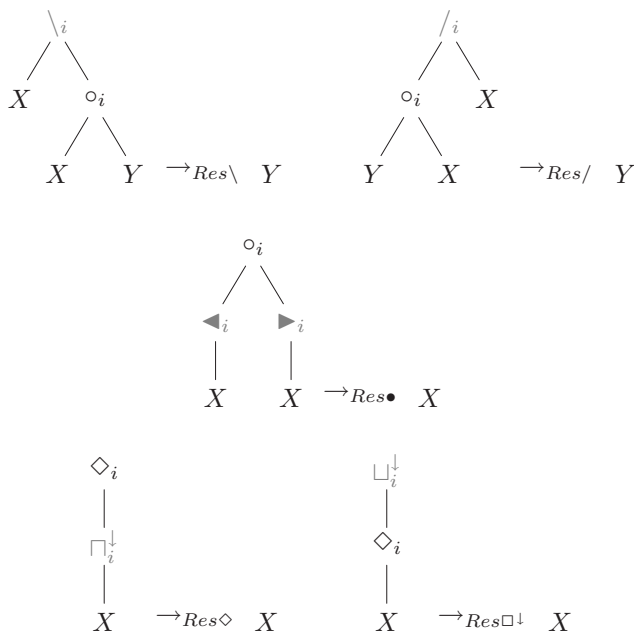


Figure A.28: Residuation conversions

[No Eager Evaluation] Will prevent Grail from doing any early failure on label conditions.

[Automatic Eager Evaluation] Will cause Grail to perform automatic eager label conversions. This is the default.

[Manual Eager Evaluation] Will allow the user to perform eager label conversions himself. Be careful, as careless eager conversions may prevent solutions from being found.

[Run] Setting this option to [Nonstop] will cause Grail perform all label conversions without user interaction. Defaults to [Creep].

A.3.5 The Lexicon Window

You can open the lexicon window from the menu bar in the main window by selecting **[Window/Lexicon Window]**, or by typing (Control-l).

The lexicon window (Figure A.29 on the next page) displays a list of the words in the fragment and of the formulas assigned to them. From here you can edit, delete or enter new lexical entries.

Clicking an entry will select it, indicated by the selection bar. The next edit or delete command will then be applied to that entry.

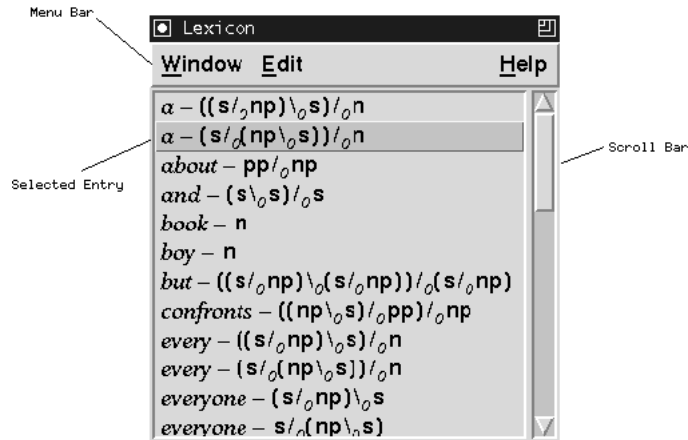


Figure A.29: The lexicon window.

Double-clicking one of the entries will open the edit lexical entry window, with that entry displayed in it (see section A.3.6 for more on the editing of lexical entries).

Clicking one of the entries with the (Control) key depressed will delete it.

Menu Bar from the menu bar of the lexicon window the following operations are available.

[Window]

[Close] Closes the lexicon window.

[Edit]

[New Entry] Opens the edit entry window.

[Edit Entry] Shows the selected lexical entry in the edit entry window.

[Delete Entry] Deletes the selected lexical entry.

[Help]

[On This Window] Gives a help message.

A.3.6 Editing a Lexical Entry

The edit entry window (Figure A.30 on the facing page) is where you modify existing entries in the lexicon or create new entries from scratch. You can open the edit entry window by selecting [Edit/New Entry] or [Edit/Edit Entry] from the lexicon window.

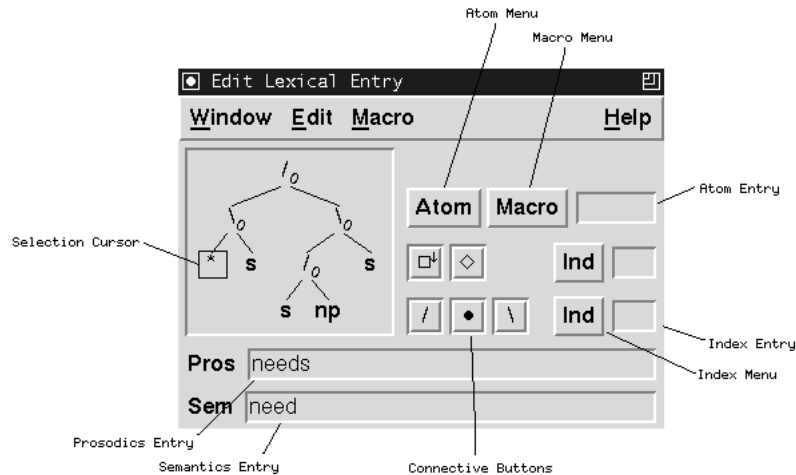


Figure A.30: The edit entry window.

The edits you perform here will only be stored in the lexicon when you press (Control-s) or select **[Edit/Store Entry]** from the menu bar, so you don't have to worry about accidentally modifying your lexicon.

A lexical entry consists of three parts: *prosodics*, a syntactic *formula* and *semantics*.

Formula the formula edit fields take up the upper section of the window.

Selection The formula is displayed as its construction tree. You can select a part of the formula by clicking on it. The selection cursor appears as a box surrounding the root of the selected tree.

Insertion Points A special constant "*" functions as an insertion point in the formula. It is not a part of the formula language. By pressing (Control-k) the selected tree will be replaced by this constant, and copied to the paste buffer.

When an insertion point is selected (as shown in Figure A.30), you can insert something at that position in one of the following ways.

[Paste] Pressing (Control-y) will insert the contents of the paste buffer to this position.

[Atom] By clicking on the atom menu, you can insert one of the atomic formulas found in this fragment. Alternately, you can type in a new atom in the atom entry, followed by (Enter). Atoms should start with

a lower case letter, and be followed by any number of alphanumeric characters or `_`.

If you want to use complex Prolog terms as atomic formulas, you will have to explicitly declare them in your fragment file. For example by using the following.

```
atomic_formula(np(nom)).
atomic_formula(np(acc)).
```

Note that you can currently do this only by editing your fragment manually.

[Macro] A very simple macro facility is provided, where you can give a name to commonly occurring formulas. Selecting one of the macros from the macro menu will insert it at the current position.

[Constructor] You can insert a unary or binary constructor by selecting an index from the index menu next to the buttons for these constructors (or typing in a new index in the index entry next to it) and pressing the button for the connective you wish to insert.

Prosodics The prosodics of an entry is the way it will appear in your expressions. You can enter a Prolog term in the prosodics entry section. The current version does not support lexical entries consisting of more than one word.

Semantics You can give your lexical entry a Montague-style lambda term meaning in the semantics entry section. Editing the semantics in the current version is very cumbersome, as it requires you to type in the internal semantic representation. It is recommended you leave the semantics field empty or type in a single constant. If you really want to enter lambda term meaning recipes you can use Table A.1 on the next page to convert lambda terms to Prolog terms.

Menu Bar In the edit lexical entry window, you can select the following from the menu bar

[Window]

[Close] Closes this window.

[Edit]

[Clear Entry] Erases the formula, prosodics and semantics fields.

[Store Entry] Stores the current lexical entry in the lexicon.

[Cut] Cuts the current selection to the paste buffer.

| Lambda Term | Prolog Term |
|-----------------------|--------------------------------|
| <i>variable</i> | Prolog variable |
| <i>constant</i> | Prolog constant |
| $(f\ x)$ | <code>appl(F,X)</code> |
| $\lambda x.t$ | <code>lambda(X,T)</code> |
| $\langle x,y \rangle$ | <code>pair(X,Y)</code> |
| $\pi^1 x$ | <code>fst(X)</code> |
| $\pi^2 x$ | <code>snd(X)</code> |
| $\vee t$ | <code>debox(T)</code> |
| $\wedge t$ | <code>conbox(T)</code> |
| $\cup t$ | <code>dedia(T)</code> |
| $\cap t$ | <code>condia(T)</code> |
| $\neg x$ | <code>not(X)</code> |
| $x \wedge y$ | <code>bool(X,&,Y)</code> |
| $x \vee y$ | <code>bool(X,\/,Y)</code> |
| $x \rightarrow y$ | <code>bool(X,->,Y)</code> |
| $\forall x.t$ | <code>quant(forall,X,T)</code> |
| $\exists x.t$ | <code>quant(exists,X,T)</code> |
| $\iota x.t$ | <code>quant(iota,X,Y)</code> |

Table A.1: Representation of semantic terms in Prolog

[Copy] Copies the current selection to the paste buffer.

[Paste] Pastes the buffer to the current position.

[Macro]

[Store Entry As Macro] Stores the formula of the current entry as a macro. The macro will take its name from the atom entry field.

[Store Selection As Macro] Stores the selection as a macro. The macro will take its name from the atom entry field.

[Help]

[On This Window] Prints a help message.

A.3.7 The Postulate Window

The postulate window (shown in Figure A.31 on the following page) portrays the structural postulates of the current fragment. From here you can delete or edit structural postulates.

You can open the postulate window from the menu bar in the main window by selecting **[Window/Postulate Window]**, or by typing (Control-p).

Clicking on a postulate will select it. This will cause a selection bar to appear over it, and allows you to perform the operations in the edit menu on it.

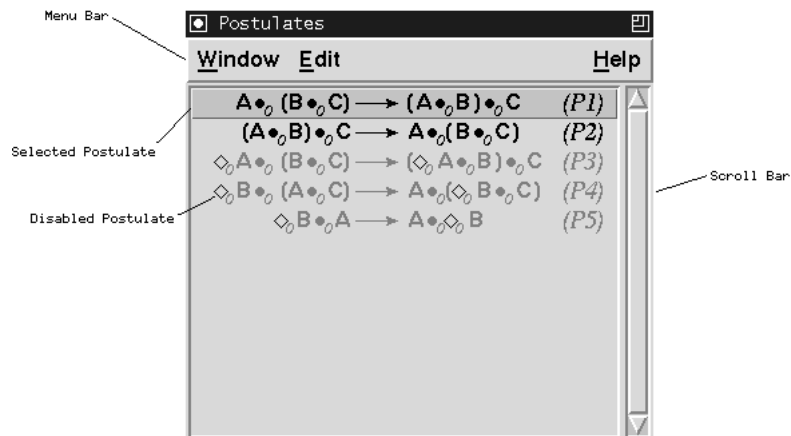


Figure A.31: The postulate window.

Double clicking a postulate will display that postulate in the edit postulate window.

Clicking a postulate with (Control) depressed will delete that postulate.

Pressing mouse button 2 over a postulate will change the status of the postulate from enabled to disabled or vice versa. This allows you to experiment with the effects of structural postulates without having to create several versions of the same fragment.

Menu Bar From the menu bar, the following options are available.

[Window]

[Close] Closes the postulate window.

[Edit]

[New Entry] Opens the edit entry window.

[Edit Entry] Shows the selected structural postulate in the edit postulate window.

[Delete Entry] Deletes the selected structural postulate.

[Disable/Enable Postulate] Toggles the selected postulate between enabled and disabled.

[Help]

[On This Window] Gives a help message.

A.3.8 Editing a Postulate

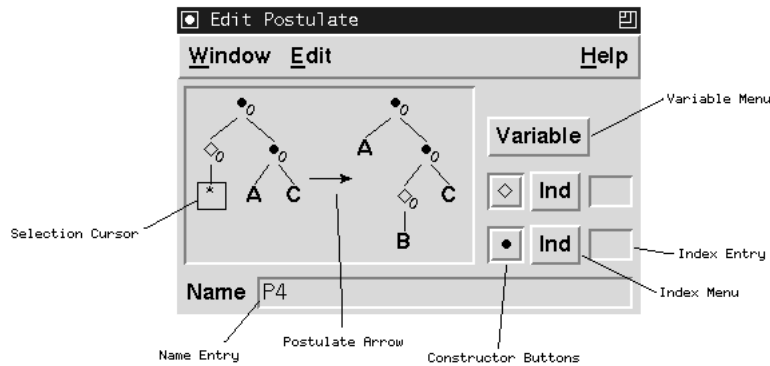


Figure A.32: The postulate edit window.

Editing a postulate is much like editing a formula. There is now a formula on both the left and the right hand side of the postulate arrow. Selection and cut/copy/paste can be performed as before.

Instead of atomic formulas we now have structural variables, which can be inserted from the variable menu, and our choice of constructors is limited to \diamond and \bullet .

Postulate Arrow By clicking on the postulate arrow it will change from \rightarrow to \leftrightarrow to \leftarrow . This makes it easier to store equivalences or inverses of postulates. In the postulate window, all postulates will appear in their left to right version regardless of the postulate arrow, so storing a postulate $A \leftrightarrow B$ will in fact be the same as storing both $A \rightarrow B$ and $B \rightarrow A$.

Postulate Names You can give a postulate any name which is printable in \LaTeX math mode.

Valid Postulates The computational architecture poses some limitations on the type of postulates allowed in your fragments. Grail will report an error when you try to store postulates of the following form.

- There are multiple occurrences of a variable on either side of the postulate arrow.
- A variable occurs on only one side of the postulate arrow.

In addition, because of the backward chaining proof search strategy, a warning will be generated when a postulate has more constructors on the

left hand side than on the right hand side. If you add one of these postulates to your fragment, the proof search algorithm is not guaranteed to terminate. This is the same restriction discussed in Section 9.2 and ensures that proof search is PSPACE complete as opposed to (potentially) undecidable.

Menu Bar the menu bar allows you to access the following functions.

[Window]

[Close] Closes the edit postulate window.

[Edit]

[Clear Postulate] Erases the postulate in this window.

[Store Postulate] Stores the postulate in memory. It will now appear in the postulate window.

[Reverse Postulate] Swaps the left and right hand sides of the postulate.

[Cut] Deletes the selected part of the postulate, and copies it to the paste buffer.

[Copy] Copies the selected part of the postulate to the paste buffer.

[Paste] Pastes the contents of the buffer to the place of the selected variable.

[Help]

[On This Window] Hmmm, what does this window do?

A.3.9 The Analysis Window

The analysis window (see Figure A.33 on the next page) is where you can improve the performance of the theorem prover by setting the parameters for early failure. This can be done either automatically or by hand.

External Modes Sometimes you may want to prevent a mode from occurring in the output, because it is used only as a grammar internal or auxiliary mode. By default all modes will be external, but you can set modes to internal by turning off their checkbox here.

Lazy, Transparent and Continuous Modes Three forms of early failure are supported which apply only to structural postulates satisfying some criteria. These are the following.

- lazy versus eager contraction of the $[R/i]$, $[R\setminus i]$ and the $[R\Box_j^\perp]$ links, as discussed in Section 8.5.

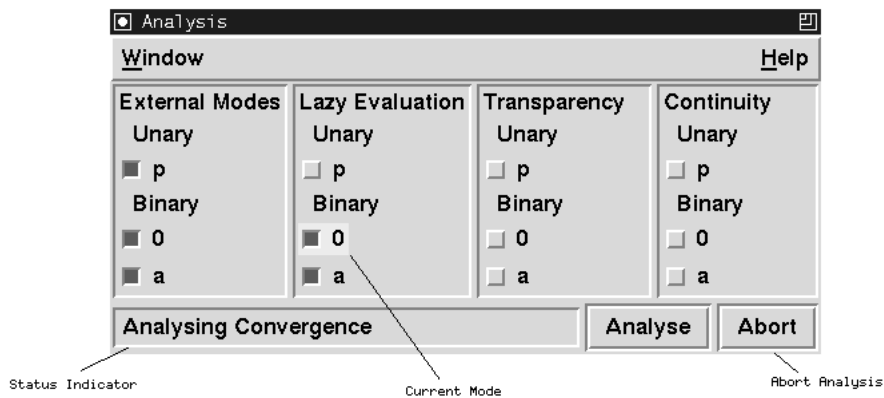


Figure A.33: The analysis window.

- lazy versus eager evaluation of word order constraints for transparent modes, as discussed in Section 8.7.
- first order approximation for continuous modes, as discussed in Section 8.7.

All can be detected by the program, and only the lazy reductions test is expensive to compute. When the program suspects checking for lazy reductions will take up an unreasonable amount of time, you will get a choice to set these parameters to their default, safe settings and only perform the other tests.

You can overrule any of these settings manually, but this may cause the theorem prover to miss valid derivations.

Menu Bar From the menu bar, the following options are available.

[Window]

[Close] Closes the analysis window.

[Options]

[Show Status] Gives a description of Grail's estimate of the current analysis settings. This can be *manual* if the settings were performed by the user, *safe* if performance is perhaps not optimal but will not prevent solutions from being found, *optimal* if a complete analysis has been performed on the current postulates, or *unknown* if postulates were added after the last analysis.

[Analyse Postulates] Performs a complete analysis of the postulate set.

[Analyse Convergence] Will only check if the label reductions converge for eager evaluation. This is generally time-consuming.

[Analyse Transparency] Will only check if word order constraints can be applied eagerly.

[Analyse Continuity] Will only check for which modes continuity labeling applies.

[Safe Settings] Switches off *all* early failure.

A.4 Conclusions

We have given an overview of the Grail interactive theorem prover and its underlying logical theory. Grail displays an intuitive representation of the state of the computation and allows the user to guide the computation by interacting with this representation.

On the proof net level, an advantage over sequent or natural deduction systems is that linking atomic formulas is a relatively trivial way to generate all proofs for a given statement. User guidance allows more experienced users to perform the axiom links they are interested in immediately, thereby sidestepping the $O(n!)$ complexity.

On the label rewrite level, it is often enlightening to see Grail (ab)use your carefully chosen structural rules in unintended ways, showing linguistically incorrect predictions of your logical theory, or to see it fail to satisfy a critical constraint, pointing to a missing or not sufficiently general structural rule. User interaction can considerably improve the performance by allowing the user to perform the intended label conversions himself.

Finally, though proof nets are in many ways an optimal proof theory for proof *search*, as previous chapters ought to have shown, natural deduction is generally a better theory to *display* them. Therefore, source code which transforms the completed proof net into \LaTeX natural deduction output is included with the release.

BIBLIOGRAPHY

- Aarts, E. (1994), 'Proving theorems of the second order Lambek calculus in polynomial time', *Studia Logica* **53**, 373–387.
- Abramsky, S. (1993), 'Computational interpretations of linear logic', *Theoretical Computer Science* **111**, 3–57.
- Abrusci, V. M. & Ruet, P. (1999), 'Non-commutative logic I: the multiplicative fragment', *Annals of Pure and Applied Logic* **101**(1), 29–64.
- Ajdukiewicz, K. (1935), 'Die syntaktische Konnexität', *Studies in Philosophy* **1**, 1–27.
- Andreoli, J.-M. (2000), 'Focussing and proof construction', *Annals of Pure and Applied Logic* . to appear.
- Bar-Hillel, Y. (1964), *Language and Information. Selected Essays on their Theory and Application*, Addison-Wesley, New York.
- Bar-Hillel, Y., Gaifman, C. & Shamir, E. (1964), On categorial and phrase structure grammars, in Y. Bar-Hillel, ed., 'Language and Information. Selected Essays on their Theory and Application', Addison-Wesley, New York, pp. 99–115.
- Bayer, S. & Johnson, M. (1995), Features and agreement, in 'Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics', San Francisco, pp. 70–76.
- Bellin, G. & van de Wiele, J. (1995), Empires and kingdoms in MLL, in J.-Y. Girard, Y. Lafont & L. Regnier, eds, 'Advances in Linear Logic', Cambridge University Press, pp. 249–270.
- Carpenter, B. (1991), 'The generative power of categorial grammars and head-driven phrase structure grammars with lexical rules', *Computational Linguistics* **17**(3), 301–314.

- Carpenter, B. (1995), The Turing-completeness of multimodal categorial grammars. Manuscript.
- Chomsky, N. (1959), 'On certain formal properties of grammars', *Information and Control* **2**(2), 137–167.
- Chomsky, N. (1995), *The Minimalist Program*, MIT Press, Cambridge, Massachusetts.
- Danos, V. (1990), La Logique Linéaire Appliquée à l'étude de Divers Processus de Normalisation (Principalement du λ -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. (1996), Partially commutative linear logic: sequent calculus and phase semantics, in V. M. Abrusci & C. Casadio, eds, 'Proofs and Linguistic Categories, Application of Logic to the Analysis and Implementation of Natural Language', CLUEB, pp. 199–208. Proceedings 1996 Roma Workshop.
- de Groote, P. (1999a), 'An algebraic correctness criterion for intuitionistic proof-nets', *Theoretical Computer Science* **224**(1–2), 115–134.
- de Groote, P. (1999b), A dynamic programming approach to categorial deduction, in H. Ganzinger, ed., 'CADE-16, 16th International Conference on Automated Deduction', Vol. 1632 of *Lecture Notes in Computer Science*, Springer, pp. 1–15.
- de Groote, P. & Lamarche, F. (2001), Classical non-associative Lambek calculus, in W. Buszkowski & M. Moortgat, eds, 'Studia Logica', Kluwer Academic Publishers. Special Issue Dedicated to Joachim Lambek.
- de Groote, P. & Retoré, C. (1996), On the semantic readings of proof nets, in G.-J. Kruijff, G. Morrill & R. T. Oehrle, eds, 'Formal Grammar', pp. 57–70.
- Dechter, R., ed. (2000), *Principles and Practice of Constraint Programming*, Vol. 1894 of *Lecture Notes in Computer Science*, Springer. Proceedings of the 6th International Conference CP2000.
- Dijkstra, E. W. (1979), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey.
- Dörre, J. (1996), Parsing for semidirectional Lambek grammar is NP-complete, in 'Proceedings of the 34th Annual Meeting of the ACL', University of California, Santa Cruz, California, pp. 95–100.
- Dörre, J. & Manandhar, S. (1995), Constraint-based Lambek calculi, in P. Blackburn & M. de Rijke, eds, 'Specifying Syntactic Structures. Studies in Logic, Language and Information', CSLI, Stanford.

- Eisinger, N. & Ohlbach, H.-J. (1993), Deduction systems based on resolution, *in* 'Handbook of Logic in Artificial Intelligence and Logic Programming', Vol. I, Oxford University Press, Oxford, chapter 4.
- Emms, M. (1993a), Extraction covering extensions of the Lambek calculus are not context free, *in* P. Dekker & M. Stokhof, eds, 'Proceedings 9th Amsterdam Colloquium', pp. 268–286.
- Emms, M. (1993b), Parsing with polymorphism, *in* 'Proceedings of the Sixth Conference of the European Association of Computational Linguistics', pp. 120–129.
- Gabbay, D. M. (1996), *Labeled Deductive Systems*, Clarendon Press.
- Gentzen, G. (1934), 'Untersuchungen über das logische Schließen', *Mathematische Zeitschrift* **39**, 176–210, 405–431.
- Girard, J.-Y. (1987), 'Linear logic', *Theoretical Computer Science* **50**, 1–102.
- Girard, J.-Y. (1991), Quantifiers in linear logic II, *in* G. Corsi & G. Sambin, eds, 'Nuovi problemi della logica e della filosofia della scienza', Vol. II, CLUEB, Bologna, Italy. Proceedings of the conference with the same name, Viareggio, Italy, January 1990.
- Girard, J.-Y., Lafont, Y. & Taylor, P. (1988), *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press.
- Guerrini, S. (1999), Correctness of multiplicative proof nets is linear, *in* 'Fourteenth Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Science Society, pp. 454–263.
- Hepple, M. (1990), The Grammar and Processing of Order and Dependency: A Categorical Approach, PhD thesis, Centre for Cognitive Science, University of Edinburgh.
- Hepple, M. (1994a), Comments on multimodal systems, *in* M. Moortgat, ed., 'Lambek Calculus. Multimodal and Polymorphic Extensions', OTS, Utrecht, pp. 37–44. DYANA Report R1.1.B.
- Hepple, M. (1994b), Labelled deduction and discontinuous constituency, *in* V. M. Abrusci, C. Cassadia & M. Moortgat, eds, 'Linear Logic and Lambek Calculus', ILLC, Amsterdam, pp. 123–150.
- Heylen, D. (1999), Types and Sorts: Resource Logic for Feature Checking, PhD thesis, Utrecht Institute of Linguistics OTS, Utrecht University.
- Hodas, J. S. (1992), Specifying filler-gap dependency parsers in a linear-logic programming language, *in* K. Apt, ed., 'Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming', MIT Press.

- Hodas, J. S. (1996), A linear logic treatment of phrase structure grammars for unbounded dependencies, in A. Lecomte, F. Lamarche & G. Perrier, eds, 'Proceedings of the Workshop on Logical Aspects of Computational Linguistics'.
- Jäger, G. (2001), Anaphora and quantification in categorial grammar, in M. Moortgat, ed., 'Logical Aspects of Computational Linguistics', Vol. 2014 of *Lecture Notes in Computer Science*, Springer, pp. 70–90.
- Johnson, M. (1998), 'Proof nets and the complexity of processing center-embedded constructions', *Journal of Logic, Language and Information* 7(4), 443–447.
- Joshi, A. (1994), Tree-adjointing grammars, in R. E. Asher, ed., 'The Encyclopedia of Language and Linguistics', Pergamon Press, Oxford, UK.
- Joshi, A. & Kulick, S. (1997), Partial proof trees, resource sensitive logics, and syntactic constraints, in C. Retoré, ed., 'Logical Aspects of Computational Linguistics, Selected Papers', Vol. 1328 of *Lecture Notes in Computer Science*, Springer, pp. 21–42.
- Joshi, A., Kulick, S. & Kurtonina, N. (2001), An LTAG perspective on categorial inference, in M. Moortgat, ed., 'Logical Aspects of Computational Linguistics', Vol. 2014 of *Lecture Notes in Computer Science*, Springer, pp. 90–105.
- Joshi, A., Levi, L. S. & Takahashi, M. (1975), 'Tree adjunct grammars', *Journal of Computer and System Science* 10, 136–163.
- Joshi, A. & Schabes, Y. (1996), Tree-adjointing grammars, in G. Rosenberg & A. Salomaa, eds, 'Handbook of Formal Languages', Vol. 3, Springer, New York, pp. 69–123.
- Kamp, H. & Reyle, U. (1993), *From Discourse to Logic*, Kluwer Academic Publishers, Dordrecht.
- Kanovich, M. (1991), The multiplicative fragment of linear logic is NP-complete, Technical report, University of Amsterdam. ITLI Prepublication Series X-91-13.
- Kaplan, R. & Bresnan, J. (1982), Lexical-functional grammar. A formal system for grammatical representation, in J. Bresnan, ed., 'The Mental Representation of Grammatical Relations', MIT Press, pp. 173–281.
- Karp, R. (1972), Reducibility among combinatorial problems, in R. Mille & J. Thatcher, eds, 'Complexity of Computer Computations', Plenum Press, New York, pp. 85–104.
- Koroda, S.-Y. (1964), 'Classes of languages and linear-bounded automata', *Information and Control* 7, 207–223.

- Kurtonina, N. (1995), Frames and Labels. A Modal Analysis of Categorical Inference, PhD thesis, OTS Utrecht, ILLC Amsterdam.
- Kurtonina, N. & Moortgat, M. (1997), Structural control, *in* P. Blackburn & M. de Rijke, eds, 'Specifying Syntactic Structures', CSLI, Stanford, pp. 75–113.
- Lafont, Y. (1995), From proof nets to interaction nets, *in* J.-Y. Girard, Y. Lafont & L. Regnier, eds, 'Advances in Linear Logic', Cambridge University Press, pp. 225–247.
- Lamarche, F. (1994), Proof nets for intuitionistic linear logic I: Essential nets, Technical report, Imperial College.
- Lamarche, F. & Retoré, C. (1996), Proof nets for the lambek calculus — an overview, *in* V. M. Abrusci & C. Casadio, eds, 'Proofs and Linguistic Categories', CLUEB, Bologna, pp. 241–262.
- Lambek, J. (1958), 'The mathematics of sentence structure', *American Mathematical Monthly* **65**, 154–170.
- Lambek, J. (1961), On the calculus of syntactic types, *in* R. Jacobson, ed., 'Structure of Language and its Mathematical Aspects, Proceedings of the Symposia in Applied Mathematics', Vol. XII, American Mathematical Society, pp. 166–178.
- Lecomte, A. & Retoré, C. (1998), Words as modules: a lexicalised grammar in the framework of linear logic proof nets, *in* C. Martin-Vide, ed., 'Mathematical and Computational Analysis of natural Language', Vol. 45 of *Studies in Functional and Structural Linguistics*, John Benjamins, pp. 129–144. Selected papers of ICML'96.
- Lincoln, P. (1995), Deciding provability of linear logic formulas, *in* J.-Y. Girard, Y. Lafont & L. Regnier, eds, 'Advances in Linear Logic', Cambridge University Press, pp. 109–122.
- Lincoln, P., Mitchell, J., Scedrov, A. & Shankar, N. (1990), Decision problems in linear logic, *in* 'Proceedings of the 31st Annual Symposium on Foundations of Computer Science', IEEE Computer Society Press, pp. 662–671.
- Montague, R. (1974), The proper treatment of quantification in ordinary English, *in* R. Thomason, ed., 'Formal Philosophy. Selected Papers of Richard Montague', Yale University Press, New Haven.
- Moortgat, M. (1990), Unambiguous proof representations for the Lambek calculus, *in* 'Proceedings 7th Amsterdam Colloquium'.
- Moortgat, M. (1996a), In situ binding: A modal analysis, *in* P. Dekker & M. Stokhof, eds, 'Proceedings 10th Amsterdam Colloquium', ILLC, Amsterdam, pp. 539–549.

- Moortgat, M. (1996b), 'Multimodal linguistic inference', *Journal of Logic, Language and Information* 5(3–4), 349–385.
- Moortgat, M. (1997), Categorical type logics, in J. van Benthem & A. ter Meulen, eds, 'Handbook of Logic and Language', Elsevier/MIT Press, chapter 2.
- 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. & Piazza, M. (2001), 'Linguistic applications of first order multiplicative linear logic', *Journal of Logic, Language and Information* 10(2), 211–232.
- Moot, R. & Puite, Q. (1999), Proof nets for multimodal categorial grammars, in G.-J. M. Kruijff & R. T. Oehrle, eds, 'Proceedings of Formal Grammar 1999', pp. 103–114.
- Moot, R. & Puite, Q. (2001), Proof nets for the multimodal Lambek calculus, in W. Buszkowski & M. Moortgat, eds, 'Studia Logica', Kluwer Academic Publishers. Special Issue Dedicated to Joachim Lambek.
- Morrill, G. (1994), *Type Logical Grammar*, Kluwer Academic Publishers, Dordrecht.
- Morrill, G. (1995), Clausal proofs and discontinuity, in R. Kempson, ed., 'Bulletin of the IGPL 3(2,3)', IGPL, pp. 403–417. Special Issue on Deduction and Language.
- Morrill, G. (1998), Incremental processing and acceptability, Technical Report LSI-98-46-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.
- Morrill, G. (1999), Relational interpretation and linguistic form, in V. M. Abrusci & C. Casadio, eds, 'Dynamic Perspectives in Logic and Linguistics', Bulzoni Editore, Roma.
- Morrill, G. (2000), Type-logical anaphora, Technical Report LSI-00-77-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.

- Morrill, G., Leslie, N., Hepple, M. & Barry, G. (1990), Categorical deductions and structural operations, in G. Barry & G. Morrill, eds, 'Studies in Categorical Grammar', Vol. 5 of *Edinburgh Working Papers in Cognitive Science*, Centre for Cognitive Science, pp. 1–21.
- Murawski, A. S. & Ong, C.-H. L. (2000), Dominator trees and fast verification of proof nets, in 'Logic in Computer Science', pp. 181–191.
- Muskens, R. (1994), Categorical grammar and discourse representation theory, in 'Proceedings COLING 94', Kyoto, pp. 508–514.
- Oehrle, R. T. (1994), 'Term-labeled categorial type systems', *Linguistics & Philosophy* 17(6), 633–678.
- Pareschi, R. (1988), A definite clause version of categorial grammar, in 'Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics', pp. 270–277.
- Pentus, M. (1995), Lambek grammars are context free, in 'Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science', Montreal, Canada, pp. 429–433.
- Pentus, M. (1997), 'Product-free Lambek calculus and context-free grammars', *Journal of Symbolic Logic* 62, 648–660.
- Pereira, F. & Shieber, S. (1987), *Prolog and Natural Language Analysis*, CSLI, Stanford.
- Pereira, F. & Warren, D. (1980), 'Definite clause grammars for language analysis. A survey of the formalism and a comparison with augmented transition networks', *Artificial Intelligence* 13, 231–278.
- Pereira, F. & Warren, D. (1983), Parsing as deduction, in '21st Annual Meeting of the Association for Computational Linguistics', Cambridge, Massachusetts, pp. 137–144.
- Perrier, G. (1999), 'A PSPACE-complete fragment of second order linear logic', *Theoretical Computer Science* 222(1–2), 267–289.
- Pollard, C. & Sag, I. (1994), *Head-Driven Phrase Structure Grammar*, CSLI, Chicago.
- Puite, Q. (1998), Proof nets with explicit negation for multiplicative linear logic, Technical report, Department of Mathematics, Utrecht University. Preprint 1079.
- Puite, Q. (2001), Sequents and Link Graphs: Contraction Criteria for Refinements of Multiplicative Linear Logic, PhD thesis, Department of Mathematics, Utrecht University.
- Puite, Q. & Moot, R. (1999), Proof nets for the multimodal Lambek calculus, Technical Report 1096, Department of Mathematics, Utrecht University.

- Rambow, O. (1994), Formal and Computational Aspects of Natural Language Syntax, PhD thesis, University of Pennsylvania.
- Roorda, D. (1991), Resource Logics: A Proof-theoretical Study, PhD thesis, University of Amsterdam.
- Schabes, Y. & Joshi, A. (1988), An Earley-type parsing algorithm for tree adjoining grammars, in '26th Meeting of the Association for Computational Linguistics', ACL.
- Schabes, Y. & Vijay-Shanker, K. (1990), Deterministic left to right parsing of tree adjoining languages, in '28th Annual Meeting of the Association for Computational Linguistics', ACL, pp. 276–283.
- Seki, H., Matsumura, T., Fujii, M. & Kasami, T. (1991), 'On multiple context-free grammars', *Theoretical Computer Science* **88**, 191–229.
- Shieber, S., Schabes, Y. & Pereira, F. (1995), 'Principles and implementation of deductive parsing', *Journal of Logic Programming* **1–2(24)**, 3–36.
- Stabler, E. (1997), Derivational minimalism, in A. Lecomte, ed., 'LACL97', Vol. 1582 of *Lecture Notes in Computer Science*, Springer.
- Tärnlund, S.-A. (1977), 'Horn clause computability', *BIT* **2**, 215–226.
- Troelstra, A. S. (1992), *Lectures on Linear Logic*, CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California.
- van Benthem, J. (1986), Categorical grammar, in 'Essays in Logical Semantics', Reidel, Dordrecht, chapter 7, pp. 123–150.
- van Benthem, J. (1987), Categorical grammar and lambda calculus, in D. Sko-rdev, ed., 'Mathematical Logic and Its Applications', Plenum Press, New York, pp. 39–60.
- van Benthem, J. (1995), *Language in Action: Categories, Lambdas and Dynamic Logic*, MIT Press, Cambridge, Massachusetts.
- Versmissen, K. (1996), Grammatical Composition: Modes, Models, Modalities, PhD thesis, Research Institute for Language and Speech, Utrecht University.
- Vijay-Shanker, K. & Joshi, A. (1985), Some computational properties of tree adjoining grammars, in '23th Meeting of the Association for Computational Linguistics', ACL, pp. 82–93.
- Weir, D. (1988), Characterizing Mildly Context Sensitive Grammar Formalisms, PhD thesis, University of Pennsylvania.

Symbols

$(\cdot)^\perp$ *see* negation, linear
 $?$ *see* why not
 $\&$ *see* with
 $!$ *see* bang
 \multimap *see* implication, linear
 \oplus *see* plus
 \otimes *see* tensor
 \wp *see* par

A

AB 154
 abstract proof structure . . . *see* proof structure, abstract
 additives 8–10
 linguistic applications . 21–23
 sequent rules 9
 adjunction 175
 null 193, 195
 obligatory 193–194
 selective 193–195
 ambiguity
 derivational 50
 lexical 21
 spurious 50
 aps . . . *see* proof structure, abstract
 associativity . . *see* structural rules, associativity
 automated deduction 89, 133–151, 203
 automated theorem proving . . . *see* automated deduction

B

bang 10
 block 121–122
 bottom up 166

C

comb 126
 commutativity *see* structural rules, commutativity
 compilation 135
 completeness 1, 46
 complexity
 linear logic 28–29
 multimodal Lambek calculus 153–172
 component 119–121, 141–145
 active 141
 completed 143
 waiting 141
 conclusion 53, 107
 confluence 62
 consistency 1, 11–13, 20, 32
 constraint programming 136
 contraction . . . *see* structural rules, contraction
 graph 61–63, 111, 201
 conversion 19
 graph 111–112
 structural 111–112
 correction graph 53, 127, 138
 first order multiplicative linear logic 72–73

count check 136
 Curry-Howard homomorphism 43
 Curry-Howard isomorphism . 17–
 20, 26–27
 cut elimination 8, 11–13, 32, 58–61,
 71, 119–122

D

de Morgan laws 8, 14, 66, 100–101
 definite clause 74
 definite clause grammar 2, 172
 derivation
 TAG 176
 difference list 74
 discourse representation theory 26
 divide and conquer 136

E

early failure 136, 139, 143, 236
 ellipsis 25
 essential net... *see* graph, dynamic
 eta expanded 58, 133
 evaluation
 eager 143, 146
 lazy 143, 237
 exponentials 10–11
 linguistic applications . 23–24
 sequent rules 10
 EXPSPACE 28

F

features 22–24, 40
 focusing 144–145
 foot 174
 formula
 axiomatic 108
 cut 108
 flow 108, 181
 multimodal 35, 39
 tensor only 178

G

grail 203–238
 download 204
 lexicon 204–210, 229–233
 license 204
 postulates .. 217–221, 233–236

reference guide 221–238
 tutorial 204–221

grammar

context free 154, 174
 context sensitive 156
 lexicalized 157
 type 0 166–172
 ϵ free 167
 type 1 .. *see* grammar, context
 sensitive

graph

dynamic 94–97
 correct 94

graph contraction. *see* contraction,
 graph

H

HPSG 172
 hypothesis 57, 107
 hypothesis comb 127–128, 145
 hypothesis tree 110–111

I

implication

linear 9, 20, 21, 101–102

index *see* mode

K

Kripke frame 45, 149

L

L *see* Lambek calculus, associative

label conversion

logical 97
 structural 96

labeled deduction 89–98, 201

Lambek calculus 31–46
 associative 31–33, 73–76,
 124–128

multimodal 35–46

non-associative . 33–34, 77–79

non-associative with permuta-
 tion 34,
 79

with empty antecedent ... 66

with permutation 26, 34,
 76–77

lexical rules 2
 LFG 172
 linear logic *see* logic, linear
 link 107
 definition 107
 Lambek calculus 67
 MILL 65
 MILL1 71
 MLL 52
 $NL \diamond_{\mathcal{R}}$ 108
 par
 active 141
 consecutive 144
 waiting 141
 link graph 129
 literal selection 141
 locality constraints 86
 logic
 classical 7
 intensional 26
 intuitionistic 15, 17
 linear 7–29
 complexity . *see* complexity,
 linear logic
 cyclic 65, 166
 fragments 13–14
 intuitionistic 14–20
 linguistic applications . 20–
 27
 non-commutative 65
 modal 38, 39, 45
 S4 10, 38
 long distance dependencies 21

M

mode 35
 continuous 145, 237
 external 134, 184, 236
 internal 134, 236
 transparent 237
 model theory 45–46, 149
 module 180, 182
 multiplicatives 8–9
 linguistic applications . 20–21
 sequent rules 9

N

natural deduction
 Lambek calculus 42–45
 linear logic 15–20
 negation
 linear 8, 66, 100–101
 NEXPTIME 28
 NL *see* Lambek
 calculus, non-associative
 $NL \diamond_{\mathcal{R}}$ *see* Lambek calculus,
 multimodal
 $NL \diamond_{\mathcal{R}^+}$ 166
 $NL \diamond_{\mathcal{R}^-}$ 155–166
 $NL_{\mathcal{R}}$ *see* Lambek calculus,
 multimodal
 normal form 20
 NP 28, 69, 71, 154, 197

P

par 8, 21
 parallel computation 136
 parasitic gapping 25, 28
 parsing as deduction 2
 partial execution 135
 Petri nets 28
 pied piping 85
 plus 8, 9, 22
 polymorphism 24, 28
 pro drop 23
 processing 141
 Prolog 2, 203
 proof net 49–199
 first order multiplicative linear
 logic 69–87
 inductive definition 50
 multiplicative intuitionistic li-
 near logic 63–65
 multiplicative linear logic 49–
 68
 non-commutative 65–68
 two sided 99–129
 proof structure 52
 abstract 109–112
 auxiliary 185
 initial 185
 saturated 185
 unsaturated 185

- first order multiplicative linear logic 71–72
 - two sided 107–109
- proof tree
 - partial 173
- PSPACE 28, 155, 166
- psycholinguistics 141
- Q**
- quantifiers 11
 - first order
 - sequent rules 11
 - generalized . . . 26, 81–83, 208, 210
 - linguistic applications . 24–25, 80–86
 - second order
 - sequent rules 11
- R**
- reduction 19
- resolution 141
- S**
- S4** *see* logic, modal, **S4**
- semantics
 - model theoretic *see* model theory
 - natural language 26–27, 42–45
 - shortest move 135
 - soundness 1, 46
 - spine 174
 - structural label 91, 93, 122–124
 - normal 97
 - reducible 97
- structural rules
 - associativity 26
 - classical logic 7
 - commutativity 25
 - contraction 7, 25
 - linear 35
 - linear logic 10
 - linguistic applications . 25–26
 - multimodal Lambek calculus 35–37
 - non-expanding 155
 - weak Sahlqvist 45–46
 - weakening 7
- structural variable 90, 93
- subformula property 13, 20, 32
- subject extraction 23
- substitution 175
- switching 53, 127, 138, 201
- syntax-semantics interfaces *see* Curry-Howard isomorphism
- T**
- TclTk 203
- tensor 8, 13, 21
- top down 166
- tree
 - auxiliary 174
 - derived 174
 - elementary 174
 - initial 174
- tree adjoining grammars . 173–199
 - adjoining constraints 193–195
 - definition 174
 - lexicalized 173–199
 - multi component 196–199
 - simultaneous 196
- U**
- units
 - additive
 - sequent rules 9
 - linguistic applications 23
 - multiplicative
 - sequent rules 9
- V**
- verb raising 37, 41
- W**
- weakening *see* structural rules, weakening
- wh extraction 104–106
- why not 10
- with 9, 17, 22

SAMENVATTING IN HET NEDERLANDS

IN deze dissertatie worden verschillende taalkundige toepassingen van bewijsnetten onderzocht. Bewijsnetten zijn geïntroduceerd voor lineaire logica door Girard (1987). In tegenstelling tot andere bewijssystemen, zoals natuurlijke deductie en sequentencalculus, hebben ze het voordeel dat ze vrij zijn van redundantie, met andere woorden, bewijsnetten abstraheren over de volgorde van regelapplicaties waar deze irrelevant zijn. Behalve dat dit conceptueel aantrekkelijk is, geeft dit ook computationele voordelen aangezien we de verschillende bewijzen voor een stelling elk maar één keer vinden.

Hoewel bewijsnetten voor lineaire logica al een tijd bestaan, was er voor de multimodale Lambek calculus $NL\Diamond_{\mathcal{R}}$ van Moortgat (1997) nog geen bewijsnetcalculus die bewijsbaar correct en volledig was. In hoofdstuk 7 van dit proefschrift geven we een correcte en volledige bewijsnet calculus voor $NL\Diamond_{\mathcal{R}}$ en onderzoeken we de praktische en computationele implicaties van deze calculus.

Dit proefschrift is opgedeeld in drie delen.

Het eerste deel van dit boek bestaat uit een inleiding in lineaire logica en Lambek calculi en geeft de lezer een beknopt overzicht van deze logica's en hun gebruik voor natuurlijke taalanalyse.

Het tweede deel behandelt de taalkundige toepassingen van bewijsnetten voor verschillende logica's.

Allereerst introduceren we bewijsnetten voor **MLL**, het multiplicatieve fragment van lineaire logica in hoofdstuk 4, en schetsen we het bewijs van correctheid voor dit systeem. We laten ook zien hoe we het intuitionistische en het noncommutatieve fragment verkrijgen door een restrictie op de vorm van de formules en door een restrictie tot planaire verbindingen respectievelijk.

In hoofdstuk 5 geven we taalkundige toepassingen van bewijsnetten voor het eerste orde multiplicatieve fragment van lineaire logica (Girard 1991),

allereerst door een inbeddingsstelling van de Lambek calculus te bewijzen en in de tweede plaats door aan te tonen hoe verschijnselen die problematisch zijn voor de standaard Lambek calculus eenvoudig behandeld kunnen worden in het eerste orde fragment.

Hoofdstuk 6 laat zien hoe labeling, zoals geïntroduceerd door Moortgat (1997), ons een methode geeft om de bewijsnetten voor lineaire logica van term labels te voorzien en zo, via condities op die term labels, een bewijsnetcalculus voor de multimodale Lambek calculus te krijgen. Nadeel van deze methode is echter het ontbreken van een correct- en volledigheidstelling.

In hoofdstuk 7 gebruiken we de twee-zijdige bewijsnetcalculus van Puite (1998) aan om zo een twee-zijdige bewijsnetcalculus voor de multimodale Lambek calculus te formuleren, met een correctie criterium in de stijl van Danos's (1990) contractie criterium. We bewijzen correctheid en volledigheid van deze calculus met betrekken tot de sequenten formulering, geven een bewijs van snede eliminatie voor deze bewijsnetten en wijzen op de relatie met de gelabelde calculus uit hoofdstuk 6.

In het derde deel onderzoeken we computationele aspecten van de bewijsnetten voor $NL\Diamond_{\mathcal{R}}$, zoals geïntroduceerd in hoofdstuk 7, en relateren we deze bewijsnetten aan andere formele grammatica's.

Hoofdstuk 8 geeft een algoritme voor automatische deductie dat gebruik maakt van bewijsnetten en suggereert verschillende strategieën om de efficiëntie van dit algoritme te verbeteren. Veel van deze verbeteringen worden ook in de in de appendix beschreven stellingsbewijzer toegepast.

Hoofdstuk 9 geeft een complexiteitsanalyse van het zoekprobleem naar bewijsnetten voor een gegeven stelling in $NL\Diamond_{\mathcal{R}}$ en laat zien dat dit probleem equivalent is aan het beslissingsprobleem voor context gevoelige grammatica's, om daarmee een PSPACE complexiteitsresultaat te geven.

In hoofdstuk 10 laten we zien hoe de bewijsnetten uit hoofdstuk 7 gerelateerd zijn aan LTAGs, een grammaticaformalisme geïntroduceerd door Joshi et al. (1975), door een fragment van $NL\Diamond_{\mathcal{R}}$ te geven dat equivalent is aan een LTAG grammatica. Een interessant gevolg van deze vertaalslag is dat dit ons een fragment van $NL\Diamond_{\mathcal{R}}$ geeft dat in polynomiale tijd beslisbaar is.

De appendix, tenslotte, introduceert de Grail automatische stellingsbewijzer, een praktische applicatie voor het ontwikkelen van grammatica's voor $NL\Diamond_{\mathcal{R}}$ die als onderdeel van dit onderzoek geschreven is.

CURRICULUM VITAE

I was born on Christmas Eve of 1972 in Alkmaar, the Netherlands. From 1985 to 1991 I studied VWO at the 'Bonhoeffer College' in Casticum.

Starting in 1991, I studied Cognitive Science and Artificial Intelligence at Utrecht University, specializing in Computational Linguistics and Logic, graduating on 27 September 1996 and writing a master's thesis 'Proof Nets and Labeling for Categorical Grammar Logics' under supervision of Michael Moortgat.

From 1997 to 2001, I was a PhD student at the Utrecht Institute of Linguistics OTS, researching the linguistic and computational applications of proof nets for Lambek calculi, again under the supervision of Michael Moortgat, resulting in this PhD thesis.

In addition to my work as a PhD student, I was editor of the Colibri electronic newsletter, junior lecturer teaching the course 'Logic Programming and Natural Language Analysis' and researcher working on annotation tools for the syntactic annotation part of the 'Corpus Gesproken Nederlands', a 10 million word corpus of spoken Dutch funded by NWO and Nederlandse Taalunie.