Type-logical investigations: proof-theoretic, computational and linguistic aspects of modern type-logical grammars

By

Richard Moot

Habilitation à diriger des recherches

Montpellier University

Defended 23 November 2021

Type-logical investigations

Proof-theoretic, computational and linguistic aspects of modern type-logical grammars

Richard Moot

CNRS, LIRMM, Université de Montpellier

1 Introduction

Logic textbooks often begin by discussing the translation of natural language expressions such as “some freshmen are intelligent” to the corresponding logical formulas such as $\exists x.[\mathit{freshman}(x) \wedge \mathit{intelligent}(x)]$ (from (**?**, p. 49)). Even for simple sentences such as this, the logical meaning is not always obvious: does the plural “some freshmen” imply there must be at least *two* intelligent freshmen for the sentence to be true, or is one enough? A translation into logic forces us to be very precise about what a sentence means.

Since the work of **?**, work in formal semantics has had as one of its main goals to *automate* the assignment of logical formulas to natural language expressions (even though a lot of work today is more concerned with describing the required input/output relations of such a system than with actually producing working, computational grammar fragments). In formal semantics, the basic data are natural language strings and sets of logical formulas representing possible meanings (*readings*) of these strings. There are of course many questions about what set of logical primitives and indeed what *logic* is best for representing natural language meaning. I will not have many things to say about these questions. **?** used higher-order logic with an identity predicate, and added connectives from modal and temporal logic, but many alternatives exists (**?**, **?**). A fully adequate logical theory of natural language semantics will need to be able to correctly model many types of reasoning (*deontic*, dealing with permission and obligation, *alethic*, dealing with possibility and necessity, *temporal*, dealing with time and temporal ordering, etc.) but also the *interactions* between them. In addition, there are many other topics in natural language semantics. Among many others, these include: imperatives

(**?**), presupposition and implicature (**?**, **?**), generics (**?**, **?**), plurals (**?**), aspect (**?**), vagueness (**?**), irony (**?**) and metaphors (**?**). In general, there are two types of problems to be solved for each of these:

1. *automated detection*, which is easy for plurals and (direct) imperatives[1] but hard for generics and for irony, and

2. *formal modelling*, which consists of modelling the syntax-semantics interface and of providing the logical primitives (with the logically and linguistically desired properties) in the semantic language.

This is not intended to be a textbook in formal semantics. The different semantic phenomena which were listed above were only intended to give an idea of the breadth and the complexity that a full logical theory of natural language semantics will have to cover. I refer the reader interested in any of these topics in semantics to the cited references. However, we will encounter a number of interesting phenomena on the syntax-semantics interface in the rest of this chapter (and further on in this book).

Type-logical grammars are a systematic way to set up theories of string-meaning relations, essentially compelling us to develop a formal theory of syntax and a formal theory of semantics in parallel. Type-logical grammars are a family of grammar formalisms aiming to provide a theory of natural language, and in particular its syntax-semantics interface, based on logic and type theory.

1.1 Motivations: grammar, meaning and logic

At this point, the reader might be skeptical about the usefulness of logic both for grammar design and for meaning representation. Might these be cases of someone with only a hammer seeing every problem as a nail? There is, of course, a trivial sense in which we can always translate any sufficiently precise and mechanisable theory of syntax or of meaning into first-order logic (given the Church-Turing thesis and the possibility of coding Turing machines into first-order logic). This, however, is not what interests us. We want to find a way of representing natural language grammar and natural language meaning into logic which is insightful, which has non-trivial consequences, and which makes non-trivial predictions. In short, we want to use logic to develop good scientific theories.

1.1.1 Logic for grammar design

Many grammar formalisms can be cast as logical theories (**?**, **?**, **?**), and this is true even for the more linguistically sophisticated ones, like HPSG, which has a standard logical interpretation (**?**, Section 1.2), and LFG, which uses

[1] That is, we can identify "close the window" and "pass me the salt" as imperatives based on their morphosyntactic properties, but identifying "it's a bit cold in here" and "can you pass me the salt?" as polite versions of the two previous phrases is much harder.

linear logic at the level of the syntax-semantics interface (?) but which can be usefully interpreted as a type-logical grammar as well (?, ?).

What distinguishes type-logical grammars from these other formalisms is that type-logical grammars are not logical *theories* but rather *logics*. In other words, while other formalisms stipulate grammatical principles as non-logical axioms (which can be tweaked in different ways, when required), in type-logical grammars the logic and the grammatical formalisms are the same object[2]. This means that we can prove important meta-theoretical properties such as decidability and computational complexity once and for all.

The reader may be skeptical about how how useful it is to use logic for modelling syntactic composition. Before the end of this chapter, I hope I will have shown enough examples of complex linguistic phenomena and their treatment both at the level of syntactic composition and at the level of meaning representation to have expelled most of this skepticism.

1.1.2 Logic for meaning representation

One of the main ways to asses the adequacy of a theory of natural language semantics is how well it can model entailment and contradiction. Textual entailment has many applications in natural language processing, notably for question answering and automatic text summarisation (?).

It is important to note that we use logic as a *normative* model of contradiction and entailment and not as a *descriptive* one. In other words, we use logic to model how humans *should* reason, not how they *do* reason. This is of course what logic is designed to do. However, some tasks in natural language processing — notably the entailment and contradiction tasks commonly used in natural language processing (?) — seek to emulate human performance on these tasks. While one of the classic problems in artificial intelligence is how to model common sense, world knowledge, etc. (?), and achieving human-like performance for these problems would be an impressive feat, there is also an important, purely logical part to entailment and contradiction. It is reasonable to demand that automated reasoning systems have perfect (or at least superhuman) scores for Aristolean syllogisms like the following.

1. None of the artists is a beekeeper.
2. All the beekeepers are chemists.
3. There is at least one beekeeper[3].

Humans do extremely poorly for this type of syllogism[4], a majority drawing a wrong conclusion like ‘None of the artists is a chemist’, ‘None of the chemists

[2]There is a strong and a weak form of this claim. The strong version states that *all* linguistic principles must be logical properties. The weak version allows for some extra-logical principles (non-logical axioms, lexical rules, etc.), but only as a last resort.

[3]This last item is only necessary because Artistotle presupposes for every ‘all A are B’ hypothesis that there exists an A.

[4]? reports 27.3% correct responses to this type of syllogism in a first experiment, and 5.3% in a second. ? separated their subjects by performance on an initial syllogism test

is an artist' or 'Some of the artists are not chemists'[5]. Similarly, we should expect perfect entailment/contradiction results for many of the types of examples in the FraCaS test set (**?**). Given the poor human performance on the harder cases for these test sets, we should expect superhuman performance for our automated reasoning systems.

This, of course, presupposes a model of word sense disambiguation to avoid fallacious arguments by equivocation like the following.

1. Exciting books are rare.
2. Rare books are expensive.
3. Exciting books are expensive.

This seems like a valid syllogism, but it uses "rare" in the sense of "few and far between" in the first sentence and in the sense of "precious, valuable" in the second. The following example is similar.

1. Nothing is better than cold beer.
2. Warm beer is better than nothing.
3. Warm beer is better than cold beer.

Here, the first sentence can be paraphrased as something like "there is nothing which is better to drink than cold beer" whereas the second can be paraphrased as "it is better to have a warm beer to drink and it is to have nothing to drink". From the logical point of view, this is more a case of the scope of the negation over an elliptical "to have/drink".

In sum, logic *by design* models correct reasoning, and though it is possible to argue how well this fits with the understanding of the typical person, a system developed for natural language understanding should have a system for correct reasoning at least as a component.

1.2 Precursors of type-logical grammars

Let's make our talk about using logic for natural language syntax more concrete. Before talking about type-logical grammars, we will first look at some of their precursors, which already introduce some of the standard ingredients of type-logical grammars.

? and **?** are generally seen as the earliest references in categorial (but not yet type-logical) grammar. This precursor of type-logical grammar is called AB grammar (after the two authors) and sometimes also classical categorial grammar (classical in the sense of *traditional* rather than as the contrast between classical and intuitionistic logic).

into a top, middle and bottom third (classified respectively as 'expert', 'good' and 'poor' reasoners). For this type of syllogism, they report 29% correct for 'expert' reasoners, 13% for 'good' reasoners, and 0% for 'poor' ones

[5]'The correct conclusion for this syllogisms is 'Some of the chemists are not artists'.

Formulas of the AB grammars are inductively defined. The atomic formulas are

n for common nouns like “book” and “student”

np for noun phrases like “John” and “the book”

s for sentences like “It rained” and “John read the book”

pp for prepositional phrases like “by John” and “in the book”

and possibly a few others, though generally these four atomic formulas are used. It is possible to add some more detail to the formulas, for example by distinguishing between pp_{by} and pp_{in} (to distinguish between prepositional phrases headed by “by” and “in” respectively), or between s_{decl} and s_{ynq} (to distinguish between a declarative sentence and a yes-no question).

Given two formulas A and B (atomic or complex), we can form formulas A/B (pronounced A *over* B) and $B\backslash A$ (pronounced B *under* A)[6].

A formula A/B is looking to its right for an expression of type B (its argument) to form an expression of type A (its result). For example, when we assign the word “the” the formula np/n we indicate it combines with an expression of type n to form an expression of type np, that is, a noun phrase. Therefore, under these assignments “the student” corresponds to $np/n, n$, and according to the meaning of “/”, these formulas combine into an expression of type np.

Similarly, a formula $B\backslash A$ is looking to its left for an expression of type B (its argument) to form an expression of type A (its result). For example, when we assign the word “slept” the formula $np\backslash s$, we indicate it combines with an expression of type np to its left to form an expression of type s, that is, a sentence. Under these assignments “the student slept”, with “the student” analysed as np as above, corresponds to $np, np\backslash s$, which is a sentence s under the meaning of “\”

AB grammars have the following two simplification rules, which are simply a translation of the intuitions behind the connectives into a formal system we can use for calculations of grammaticality.

$$\frac{A/B \quad B}{A}\,[/] \qquad\qquad \frac{B \quad B\backslash A}{A}\,[\backslash]$$

Using these rules, we can show that “the student slept” is a valid sentence as follows (the English words are written above the given formulas using a “Lex” rule).

[6]**?** does not distinguish between the leftward looking and the rightward looking implication. The notation used here was introduced by **?** and was adopted by Bar-Hillel in his later work (**?**).

$$\frac{\Delta \vdash A \bullet B \quad \Gamma, A, B, \Gamma' \vdash C}{\Gamma, \Delta, \Gamma' \vdash C}\ \bullet E \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \bullet B}\ \bullet I$$

$$\frac{\Gamma \vdash A/B \quad \Delta \vdash B}{\Gamma, \Delta \vdash A}\ /E \qquad \frac{\Gamma, B \vdash A}{\Gamma \vdash A/B}\ /I$$

$$\frac{\Gamma \vdash B \quad \Delta \vdash B\backslash A}{\Gamma, \Delta \vdash A}\ \backslash E \qquad \frac{B, \Gamma \vdash A}{\Gamma \vdash B\backslash A}\ \backslash I$$

Table 1.1: Natural deduction for **L**

$$\cfrac{\cfrac{\cfrac{\text{the}}{np/n}\ \mathit{Lex} \quad \cfrac{\text{student}}{n}\ \mathit{Lex}}{np}\ / \qquad \cfrac{\text{slept}}{np\backslash s}\ \mathit{Lex}}{s}\ \backslash$$

This simply amounts to making our previous intuitions precise.

1.3 Lambek calculus and type-logical grammars

Historically, **?** initiated type-logical grammars: he extended the earlier categorial grammars of **?** and **?** into a full logic, which he called the Syntactic Calculus but which is commonly referred to as the Lambek calculus.

With respect to the formulas, the Lambek calculus does not represent a large shift with respect to AB grammars: while Lambek extended the formulas by adding the product — a formula $A \bullet B$ corresponds to the concatenation of an expression of type A to an expression of type B — the product connective has little (if any) concrete linguistic applications in Lambek grammars. It is the connective which corresponds to the comma symbol "," in the antecedent. In other words, the sequence of formulas $np/n, n$ corresponds to the single formula $(np/n) \bullet n$.

The main shift of the Lambek calculus is not in the formulas, but in the rules. From the point of natural deduction proofs, AB grammars have only rules of *use* (elimination rules, which remove a connective; that is, an occurrence of a connective in the premisses of the rule application is eliminated from the conclusion of the rule) but not rules of *proof* (introduction rules, which introduce a connective; that is the conclusion of the rule contains an occurrence of a connective which didn't appear in any of the premisses).

Table 1.1 presents the natural deduction rules for the Lambek calculus. In the rules, Γ, Γ' and Δ correspond to sequences $A_1, \ldots, A_n$ of formulas. The $/I$ and $\backslash I$ rules have the condition that Γ is not the empty sequence (for the other rules, the sequences are allowed to be empty). This condition on the $/I$ and $\backslash I$ rules has the net effect of disallowing derivations with an empty antecedent. For example $\vdash n/n$ is not a theorem.

1.3.1 The Lambek calculus and Lambek grammars

The difference between the Lambek calculus and Lambek *grammars* is only a small step: the addition of a *lexicon* mapping words of the language we want to model to sets of Lambek calculus formulas.

Definition 1.1 *A* Lambek grammar *is a tuple* $\langle \Sigma, Lex, goal \rangle$ *where*

1. Σ *is the set of words in the language*
2. *the lexicon Lex, is a function from* $w \in \Sigma$ *to a (non-empty) set of Lambek calculus formulas*
3. *goal is the set of goal formulas*

Most authors choose the singleton set $\{s\}$ for the set of goal formulas g. This is of course fine from the point of view of formal language theory. However, when we want to model the syntax-semantics interface of different types of expressions — such as declarative sentences, imperatives, yes/no questions, *wh* questions, etc. — it seems a priori desirable to allow for different types of expressions, with different syntactic distributions but also with (potentially) different meanings[7].

In practice, we rarely specify Lambek grammars (and type-logical grammars) in such a formal way. Most of the time, we simply specify the lexicon and leave Σ implicit (as the domain of the *Lex* function; in other words, Σ is the set of those w such that $Lex(w)$ is defined and non-empty), and similarly *goal* is the set of succedent formulas for our given set of examples.

Definition 1.2 *Given a Lambek grammar* $\langle \Sigma, Lex, goal \rangle$*, a sentence* $w_1, \ldots, w_n$ *is* grammatical *if for all* $1 \leq i \leq n$*,* $w_i \in \Sigma$*, there is an* $A_i \in Lex(w_i)$*, there is a* $C \in goal$*, such that* $A_1, \ldots, A_n \vdash C$ *is a Lambek calculus theorem. A sentence is* ungrammatical *otherwise.*

In other words, a sentence is grammatical whenever we can assign to each of its words a formula from the lexicon, and the sequence of formulas corresponding to the sequence of words is a derivable statement for one of the goal formulas. For Lambek grammars, and type-logical grammars in general, the notion of grammaticality and derivability coincide.

We say a grammar *overgenerates* whenever:

1. it allows us to derive ungrammatical sentences, or
2. when it generates an unavailable reading for a grammatical sentence (that is, although the sentence may be grammatical, the grammar assigns it an incorrect semantic interpretation).

A grammar *undergenerates* whenever:

[7]In formal language theory, we generally treat multiple goal formulas $g_1, \ldots, g_n$ by adding n rewrite rules $g_i \rightarrow s$ (for $1 \leq i \leq n$). However, in type-logical grammars we disallow (or at least avoid as much as possible) such non-logical axioms.

1. it fails to derive grammatical sentences, or
2. it fails to generate an available reading for a sentence it derives (that is, it fails to generate at least one of the possible meanings of a sentence).

The notions of undergeneration and overgeneration have both a syntactic aspect (the items 1 above, generating all and only the right sentences) and a semantic aspect (the items 2 above, generating all and only the right meanings for these sentences). When we want to be fully precise, we will sometimes talk about syntactic and semantic under- and overgeneration to distinguish between the two cases.

Not all cases of overgeneration and undergeneration are equally grave: as **?** already notes, it is hard to rule out sentences like (1) and (2) below as ungrammatical while at the same time accepting sentences like (3) and (4).

(1) John works today today.

(2) John works yesterday today.

(3) John works today at lunch.

(4) John works at lunch today.

We therefore generally choose to accept all the sentences (1) to (4) as grammatical, and explain the oddness of sentences (1) and (2) by appealing to semantic/pragmatic notations: the second "today" in (1) is semantically superfluous, whereas the combination of "yesterday" and "today" in sentence (2) is semantically contradictory.

The general point here is that we are willing to accept some types of overgeneration provided that:

1. we can make an appeal to an alternative mechnanism for ruling out these cases (typically one of semantics, pragmatics or processing), and
2. we make a trade-off between the added grammar complexity of treating the phenomenon versus the seriousness of the overgeneration, and judge that the simplicity of the grammar is more important.

A simple Lambek calculus lexicon is shown in Table 1.2. I have adopted the standard convention in type-logical grammars of not using set notation for the lexicon, but instead listing multiple lexical entries for a word separately. This corresponds to treating *Lex* as a non-deterministic function rather than as a set-valued function.

Proper names, such as "Alyssa" and "Emory" are assigned the category np. Common nouns, such as "student" and "exam" are assigned the category n. Adjectives, such as "difficult" or "erratic" are not assigned a basic syntactic category but rather the category n/n, indicating they are looking for a common noun to their right to form a new common noun, so we predict that both "difficult exam" and "exam" can be assigned category n. For more complex entries, "someone" is looking to its right for a verb phrase to produce a sentence, where $np\backslash s$ is the Lambek calculus equivalent of verb

$lex(Alyssa) = np$	$lex(ran) = np\backslash s$
$lex(Emory) = np$	$lex(slept) = np\backslash s$
$lex(logic) = np$	$lex(loves) = (np\backslash s)/np$
$lex(the) = np/n$	$lex(aced) = (np\backslash s)/np$
$lex(difficult) = n/n$	$lex(passionately) = (np\backslash s)\backslash(np\backslash s)$
$lex(erratic) = n/n$	$lex(during) = ((np\backslash s)\backslash(np\backslash s))/np$
$lex(student) = n$	$lex(everyone) = s/(np\backslash s)$
$lex(exam) = n$	$lex(someone) = (s/np)\backslash s$
$lex(who) = (n\backslash n)/(np\backslash s)$	$lex(every) = (s/(np\backslash s))/n$
$lex(whom) = (n\backslash n)/(s/np)$	$lex(some) = ((s/np)\backslash s)/n$

Table 1.2: Lambek calculus lexicon

phrase, whereas "whom" is first looking to its right for a sentence which is itself missing a noun phrase to its right, and then to its left for a noun.

Given the lexicon of Table 1.2, we can already derive some fairly complex sentences, such as the following, and, as we will see in the next section, obtain the correct semantics.

(5) Every student aced some exam.

(6) The student who slept during the exam loves Alyssa.

One of the two derivations of Sentence (5) is shown in Figure 1.1. To improve readability, the figure uses a "sugared" notation: instead of writing the lexical hypothesis corresponding to "exam" as $n \vdash n$, we have written it as $exam \vdash n$. The withdrawn np's corresponding to the object and the subject are given a labels p_0 and q_0 respectively; the introduction rules are coindexed with the withdrawn hypotheses, even though this information can be inferred from the rule instantiation.

We can always uniquely reconstruct the antecedent from the labels. For example, the sugared statement "p_0 aced $q_0 \vdash s$" in the proof corresponds to $np, (np\backslash s)/np, np \vdash s$.

Although it is easy to verify that the proof of Figure 1.1 has correctly applied the rules of the Lambek calculus, finding such a proof from scratch may seem a bit complicated (the key steps at the beginning of the proof involve introducing two np hypotheses and then deriving s/np to allow the object quantifier to take narrow scope). We will defer the question "given a statement $\Gamma \vdash C$, how do we decide whether or not it is derivable?" to Chapter 3. In what follows, we will discuss how this proof corresponds to the following logical formula.

$$
\dfrac{\dfrac{\text{every} \vdash (s/(np\backslash s))/n \quad \text{student} \vdash n}{\text{every student} \vdash s/(np\backslash s)}\,/E \qquad \dfrac{\dfrac{\dfrac{\dfrac{[p_0 \vdash np]^2 \quad \dfrac{\text{aced} \vdash (np\backslash s)/np \quad [q_0 \vdash np]^1}{\text{aced } q_0 \vdash np\backslash s}\,/E}{p_0 \text{ aced } q_0 \vdash s}\,\backslash E}{p_0 \text{ aced} \vdash s/np}\,/I_1 \quad \dfrac{\text{some} \vdash ((s/np)\backslash s)/n \quad \text{exam} \vdash n}{\text{some exam} \vdash (s/np)\backslash s}\,/E}{p_0 \text{ aced some exam} \vdash s}\,\backslash E}{\text{aced some exam} \vdash np\backslash s}\,\backslash I_2}{\text{every student aced some exam} \vdash s}\,/E
$$

Figure 1.1: “Every student aced some exam” with the subject wide scope reading.

$$
\dfrac{\Delta \vdash A \otimes B \quad \Gamma, A, B \vdash C}{\Gamma, \Delta \vdash C}\,\otimes E \qquad \dfrac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,\otimes I
$$

$$
\dfrac{\Gamma \vdash B \quad \Delta \vdash B \multimap A}{\Gamma, \Delta \vdash A}\,\multimap E \qquad \dfrac{\Gamma, B \vdash A}{\Gamma \vdash B \multimap A}\,\multimap I
$$

Table 1.3: Natural deduction for **LP**

$$
\forall x.[\mathit{student}(x) \Rightarrow \exists y.[\mathit{exam}(y) \wedge \mathit{ace}(x, y)]]
$$

1.3.2 Deep structure and semantics

The deep structure of an expression is an intermediate structure which allows us to compute the meaning. The term “deep structure” was introduced in early generative syntax, but has fallen out of favour since. In spite of the fact that authors working with type-logical grammars have chosen to study many different logics, there seems to be a generally agreed upon deep structure, which I will refer to as the standard type-logical deep structure.

The deep structure of a type-logical derivation is a homomorphism from proofs in the logic to proofs in the Lambek-van Benthem calculus **LP**. Since the advent of linear logic, this logic is better known as (multiplicative) intuitionistic linear logic (MILL). The natural deduction rules for **LP** are shown in Table 1.3. The main difference with natural deduction for the Lambek calculus is that antecedents are now multisets instead of lists (in other words, multiplicative linear logic is a commutative logic, whereas the Lambek calculus is a non-commutative one). Under commutativity, the two Lambek calculus implications A/B and $B\backslash A$ become indistinguishable, and there is therefore only a single implication $B \multimap A$ in multiplicative linear logic.

For the Lambek calculus, specifying the homomorphism to multiplicative intuitionistic linear logic is easy: we replace the two implications ‘$\backslash$’ and ‘$/$’ by the linear implication ‘$\multimap$’ and the product ‘$\bullet$’ by the tensor ‘$\otimes$’. In a statement $\Gamma \vdash C$, Γ is now a multiset of formulas instead of a sequence. In other words, the sequent comma ‘,’ is now associative, commutative instead

$$\dfrac{\dfrac{n \multimap ((np \multimap s) \multimap s) \quad n}{(np \multimap s) \multimap s}\multimap E \qquad \dfrac{\dfrac{\dfrac{\dfrac{[np]^2 \quad \dfrac{np \multimap (np \multimap s) \quad [np]^1}{np \multimap s}\multimap E}{s}\multimap E}{np \multimap s}\multimap I_1 \qquad \dfrac{n \multimap ((np \multimap s) \multimap s) \quad n}{(np \multimap s) \multimap s}\multimap E}{s}\multimap E}{np \multimap s}\multimap I_2}{s}\multimap E$$

Figure 1.2: Deep structure of the derivation of Figure 1.1.

of associative, non-commutative. For the proof of Figure 1.1 of the previous section, this mapping gives the proof shown in Figure 1.2.

We have kept the order of the premisses of the rules the same as in Figure 1.1 to allow for an easier comparison. This deep structure still uses the same atomic formulas as the Lambek calculus, it just forgets about the order of the formulas and therefore can no longer distinguish between the leftward looking implication '\' and the rightward looking implication '/'.

To reduce space, Figure 1.2 uses implicit antecedents. We can compute the antecedents explicitly by taking the multiset of undischarged leaves at each node. For example, the $\multimap I$ rule discharging hypothesis one corresponds to the following step with explicit antecedents (the np discharged at the second $\multimap I$ rule is free in this subproof and therefore appears both in the premiss and the conclusion of the rule).

$$\dfrac{np, np \multimap (np \multimap s), np \vdash s}{np, np \multimap (np \multimap s) \vdash np \multimap s} \multimap I_1$$

To obtain a semantics in the tradition of **?**, we use the following mapping from syntactic types to semantic types, using Montague's atomic types e (for *entity*) and t (for *truth value*).

$$\begin{aligned} np^* &= e \\ n^* &= e \to t \\ s^* &= t \\ (A \multimap B)^* &= A^* \to B^* \end{aligned}$$

Applying this mapping to the deep structure proof of Figure 1.2 produces the intuitionistic proof and the corresponding (linear) lambda term as shown in Figure 1.3

The computed term corresponds to the derivational semantics of the proof. To obtain the complete meaning, we need to substitute, for each of $z_0, \ldots, z_4$, the meaning assigned in the lexicon.

For example, "every" has syntactic type $(s/(np\backslash s))/n$ and its semantic type is $(e \to t) \to (e \to t) \to t$. The corresponding lexical lambda

$$\dfrac{\dfrac{z_0^{(e\to t)\to(e\to t)\to t} \quad z_1^{e\to t}}{(z_0\,z_1)^{(e\to t)\to t}}\to E \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x^e]^2 \quad \dfrac{z_2^{e\to(e\to t)} \quad [y^e]^1}{(z_2\,y)^{e\to t}}\to E}{((z_2\,y)\,x)^t}\to E}{\lambda y.((z_2\,y)\,x)^{e\to t}}\to I_1 \qquad \dfrac{z_3^{(e\to t)\to(e\to t)\to t} \quad z_4^{e\to t}}{(z_3\,z_4)^{(e\to t)\to t}}\to E}{((z_3\,z_4)\,\lambda y.((z_2\,y)\,x))^t}\to E}{\lambda x.((z_3\,z_4)\,\lambda y.((z_2\,y)\,x))^{e\to t}}\to I_2}{}}{((z_0\,z_1)\,(\lambda x.((z_3\,z_4)\,\lambda y.((z_2\,y)\,x))))^t}\to E$$

Figure 1.3: Intuitionistic proof and lambda term corresponding to the deep structure of Figure 1.2.

term of this type is $\lambda P^{e\to t}.\lambda Q^{e\to t}.(\forall(\lambda x^e.((\Rightarrow (P\,x))(Q\,x))))$, with '$\forall$' a constant of type $(e \to t) \to t$ and '$\Rightarrow$' a constant of type $t \to (t \to t)$. In the more familiar Montague formulation, this lexical term corresponds to $\lambda P^{e\to t}.\lambda Q^{e\to t}.\forall x.[(P\,x) \Rightarrow (Q\,x)]$, where we can see the formula in higher-order logic we are constructing more clearly. Although the derivational semantics is a linear lambda term, the lexical term assigned to "every" is not, since the variable x has two bound occurrences.

The formula assigned to "some" has the same semantic type but a different term $\lambda P^{e\to t}.\lambda Q^{e\to t}.(\exists(\lambda x^e.((\wedge(P\,x))(Q\,x))))$, corresponding to the more familiar $\lambda P^{e\to t}.\lambda Q^{e\to t}.\exists x.[(P\,x) \wedge (Q\,x)]$).

The other words are simple, "exam" is assigned $\mathit{exam}^{e\to t}$, "student" is assigned $\mathit{student}^{e\to t}$, and "aced" is assigned $\mathit{ace}^{e\to(e\to t)}$ (abstracting away from the past tense information for the moment).

So to compute the meaning, we start with the derivational semantics, repeated below.

$$((z_0\,z_1)\,(\lambda x.((z_3\,z_4)\,\lambda y.((z_2\,y)\,x))))$$

Then we substitute the lexical meanings, for $z_0, \dots, z_4$.

$$\begin{aligned}
z_0 &:= \lambda P^{e\to t}.\lambda Q^{e\to t}.(\forall(\lambda x^e.((\Rightarrow (P\,x))(Q\,x))))\\
z_1 &:= \mathit{student}^{e\to t}\\
z_2 &:= \mathit{ace}^{e\to(e\to t)}\\
z_3 &:= \lambda P^{e\to t}.\lambda Q^{e\to t}.(\exists(\lambda x^e.((\wedge(P\,x))(Q\,x))))\\
z_4 &:= \mathit{exam}^{e\to t}
\end{aligned}$$

This produces the following lambda term.

$$\begin{gathered}
((\lambda P^{e\to t}.\lambda Q^{e\to t}.(\forall(\lambda x^e.((\Rightarrow (P\,x))(Q\,x))))\,\mathit{student}^{e\to t})\\
(\lambda x.((\lambda P^{e\to t}.\lambda Q^{e\to t}.(\exists(\lambda x^e.((\wedge(P\,x))(Q\,x))))\,\mathit{exam}^{e\to t})\\
\lambda y.((\mathit{ace}^{e\to(e\to t)}\,y)\,x))))
\end{gathered}$$

Finally, when we normalise this lambda term, we obtain the following semantics for this sentence.

$$(\forall(\lambda x^e.((\Rightarrow (student^{e\to t}\, x))(\exists(\lambda y^e.((\wedge(exam^{e\to t}\, y))((ace^{e\to(e\to t)}\, y)\, x)))))))$$

This lambda term represents the following more readable first-order logic formula[8].

$$\forall x.[student(x) \Rightarrow \exists y.[exam(y) \wedge ace(x,y)]]$$

Proofs in the Lambek calculus, and in type-logical grammars are subsets of the proofs in intuitionistic (linear) logic and these proofs are compatible with formal semantics in the tradition initiated by **?**.

For the example in this section, we have calculated the semantics of a simple example in 'slow motion': many authors assign a lambda term directly to a proof in their type-logical grammar, leaving the translation to intuitionistic linear logic implicit.

Given a semantic analysis without a corresponding syntactic proof, we can try to reverse engineer the syntactic proof. For example, suppose we want to assign the reflexive "himself" the lambda term $\lambda R^{(e\to e\to t)}\lambda x^e.((R\,x)\,x)$, that is, a term of type $(e \to e \to t) \to e \to t$. Then, using some syntactic reasoning to eliminate implausible candidates like $(np \multimap n) \multimap n$, the only reasonable deep structure formula is $(np \multimap np \multimap s) \multimap (np \multimap s)$ and, reasoning a bit further about which of the implications is left and right, we quickly end up with the quite reasonable (though far from perfect) Lambek calculus formula $((np\backslash s)/np)\backslash(np\backslash s)$.

1.4 Some applications

In spite of its simplicity, the Lambek calculus already gives a fairly reasonable account of a number of non-trivial phenomena on the syntax-semantics interface. Unlike many other formalisms, these accounts follow directly from the logical setup of the Lambek calculus and require no special stipulations. As we will see in the next section, the Lambek calculus also has a number of limitations and a main research goal for type-logical grammars has been to overcome these limitations while sacrificing neither the simplicity nor the good metatheoretical properties of the logical foundations.

In Section 1.3.1, we have already seen some examples of quantification. While limited to peripheral cases, these are still a classic example of how Lambek grammars generate the string-meaning relation with relatively little stipulations. Notably, there is no special rule for quantifier scope.

1.4.1 Relativisers and extraction

The treatment of relative pronouns and *wh* questions is, at least when they occur in peripheral positions, one of the appealing points of Lambek grammars. For example, we can generate sentences like the following without problems.

[8]We have used the standard convention in Montague grammar of writing $(p\,x)$ as $p(x)$ and $((p\,y)\,x)$ as $p(x,y)$, for a predicate symbol p.

(7) The man who just left forgot to pay.

(8) Where did he go?

(9) Irene found the book which Ivan forgot.

The subject noun phrase of Sentence (7) is derived as follows.

$$\dfrac{\dfrac{\text{the}}{np/n}\,Lex \qquad \dfrac{\dfrac{\text{man}}{n}\,Lex \qquad \dfrac{\dfrac{\text{who}}{(n\backslash n)/(np\backslash s)}\,Lex \qquad \dfrac{\dfrac{\text{just}}{(np\backslash s)/(np\backslash s)}\,Lex \qquad \dfrac{\text{left}}{np\backslash s}\,Lex}{np\backslash s}\,/E}{n\backslash n}\,/E}{n}\,\backslash E}{np}\,/E$$

Sentence (8) is only slightly more complicated. For the atomic formulas, *inf* indicates an infinitive, s_q indicates a verb-first question sentence, whereas s_{whq} indicates a sentence starting with a *wh* question (some better treatment using features should probably be used to avoid multiplying the atomic formulas like this). The key word "where" is therefore looking to its right for a verb-initial sentence which itself is missing a prepositional phrase at its rightmost edge. The word "go" forms an infinitival phrase when combined with a prepositional phrase and we can easily complete the proof after combining "go" with the hypothetical *pp* of "where". This gives the following proof.

$$\dfrac{\dfrac{\text{where}}{s_{whq}/(s_q/pp)}\,Lex \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{did}}{(s_q/inf)/np}\,Lex \qquad \dfrac{\text{he}}{np}\,Lex}{s_q/inf}\,/E \qquad \dfrac{\dfrac{\text{go}}{inf/pp}\,Lex \qquad [pp]^1}{inf}\,/E}{s_q}\,/E}{s_q/pp}\,/I_1}{s_{whq}}\,/E$$

Finally, Sentence (9) is derived in a similar manner. The word "which" licenses the hypothetical noun phrase which is the object of "forgot", providing the following proof (only the proof of the object *np* is shown).

$$\dfrac{\dfrac{\text{the}}{np/n}\,Lex \qquad \dfrac{\dfrac{\text{book}}{n}\,Lex \qquad \dfrac{\dfrac{\text{which}}{(n\backslash n)/(s/np)}\,Lex \qquad \dfrac{\dfrac{\dfrac{\text{Barry}}{np}\,Lex \qquad \dfrac{\dfrac{\text{forgot}}{(np\backslash s)/np}\,Lex \qquad [np]^1}{np\backslash s}\,/E}{s}\,\backslash E}{s/np}\,/I_1}{n\backslash n}\,/E}{n}\,\backslash E}{np}\,/E$$

There are a few thing worth mentioning about the analyses above. First, we analyse extraction by just choosing the appropriate formula for the relevant words (the *wh* words “who”, “where” and “which”) keeping all other lexical assignments the same as they would be for declarative (or in the case of Sentence (8) verb-initial) sentences. There is no need to stipulate extraction or movement principles, we simply use the available logical rules. A second point is that these analyses automatically get the meaning right. That is, the lambda terms which correspond to the deep structure linear logic proofs can be combined with lexical lambda terms to obtain the standard meaning for these sentences in formal semantics[9].

1.4.2 Right node raising

Where for the previous examples of quantifiers and extraction our analysis was somewhat limited because of the limitation to peripheral extraction in the Lambek calculus (something which will be addressed in several different ways in the next chapter), there are some other examples where the limitation to the periphery works in our favour. Examples like Sentence (10) are called *right node raising* in the literature.

(10) Howard loved but Geoffrey hated “Syntactic Structures”.

The key property of this sentence we want to capture in our analysis is that the noun phrase “Syntactic Structures” is the object both of the verb “loved” and of the verb “hated”. In other words, we want the meaning of the above sentence to be roughly equivalent to “Howard loved *Syntactic Structures* but Geoffrey hated Syntactic Structures”.

A problem with this type of coordination in many other formalisms is that the two coordinated phrases “Howard loved” and “Geoffrey hated” are not constituents (right node raising is an instance of so-called non-constituent coordination), and these formalisms therefore need to appeal to special mechanisms to ensure the grammaticality of such sentences[10].

However in the Lambek calculus, deriving Sentence (10) is relatively simple, by just assigning “and” one of its standard categories $(X\backslash X)/X$, with $X = s/np$ we obtain the following proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Howard}}{np}\,\mathit{Lex} \quad \dfrac{\dfrac{\text{loved}}{(np\backslash s)/np}\,\mathit{Lex} \quad [np]^1}{np\backslash s}\,/E}{s}\,\backslash E}{s/np}\,/I_1 \quad \dfrac{\dfrac{\text{but}}{((s/np)\backslash(s/np))/(s/np)}\,\mathit{Lex} \quad \dfrac{\dfrac{\dfrac{\text{Geoffrey}}{np}\,\mathit{Lex} \quad \dfrac{\dfrac{\text{hated}}{(np\backslash s)/np}\,\mathit{Lex} \quad [np]^2}{np\backslash s}\,/E}{s}\,\backslash E}{s/np}\,/I_2}{(s/np)\backslash(s/np)}\,/E}{s/np}\,\backslash E \quad \dfrac{\text{Syntactic Structures}}{np}\,\mathit{Lex}}{s}\,/E$$

1.4.3 Argument cluster coordination

A slightly more complicated example is so-called argument cluster coordination. These are examples like the following.

[9]There is some debate about the appropriate meaning for questions, but the lambda term obtained for Sentence (8) is mostly agnostic with respect to these debates.

[10]The term ‘right node raising’ actually refers to one such operation.

(11) Terry gave Robin flowers and Sue a book.

This sentences should mean the same as "Terry gave Robin flowers and *Terry gave* Sue a book". One solution is to use $(X\backslash X)/X$ with $X = np \bullet np$. However, to get the meaning right it is more convenient to use $X = ((s/np)/np)\backslash s$ (in terms of the typed lambda calculus this amounts to defining pairs using implication). This gives the following proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Terry}}{np}\,Lex \quad \dfrac{\dfrac{\dfrac{\text{gave}}{((np\backslash s)/np)/np}\,Lex \quad [np]^1}{(np\backslash s)/np}\,/E \quad [np]^2}{np\backslash s}\,/E}{s}\,\backslash E}{s/np}\,/I_2}{(s/np)/np}\,/I_1 \quad \dfrac{\dfrac{\dfrac{\dfrac{[(s/np)/np]^3 \quad \dfrac{\text{Robin}}{np}\,Lex}{s/np}\,/E \quad \dfrac{\text{flowers}}{np}\,Lex}{s}\,\backslash E}{((s/np)/np)\backslash s}\,\backslash I_3 \quad \dfrac{\dfrac{\text{and}}{(X\backslash X)/X}\,Lex \quad \dfrac{\dfrac{\dfrac{[(s/np)/np]^4 \quad \dfrac{\text{Sue}}{np}\,Lex}{s/np}\,/E \quad \dfrac{\text{a book}}{(s/np)\backslash s}\,Lex}{s}\,\backslash E}{((s/np)/np)\backslash s}\,\backslash I_4}{X\backslash X}\,/E}{((s/np)/np)\backslash s}\,\backslash E}{s}\,\backslash E$$

Similarly, we can conjoin a combination of a noun phrase and an adverb.

(12) Captain Jack served lobster yesterday and bananafish today.

The Lambek calculus treats the above sentence correctly, since both "lobster yesterday" and "bananafish today", are antecedents of the form $np, s\backslash s$, from which we can derive $((np\backslash s)/np)\backslash(np\backslash s)$[11]).

$$\dfrac{\dfrac{\dfrac{\dfrac{[np]^1 \quad \dfrac{[(np\backslash s)/np]^2 \quad \dfrac{\text{lobster}}{np}\,Lex}{np\backslash s}\,/E}{s}\,\backslash E \quad \dfrac{\text{yesterday}}{s\backslash s}\,Lex}{s}\,\backslash E}{np\backslash s}\,\backslash I_1}{((np\backslash s)/np)\backslash(np\backslash s)}\,\backslash I_2$$

What we have seen with these examples is that many interesting linguistic phenomena achieve an at least fairly reasonable analysis in Lambek grammars. Moreover, these analyses follow from simple lexical assignments and give a good account of the meaning of the phrases. All without stipulating any construction specific rules (no gap/trace principles, move conditions, or anything).

1.5 Problems and limitations

We spent the last section giving an overview of some non-trivial facts about the syntax-semantics interface of natural language which Lambek grammars handle at least reasonably well. However, upon closer examination we find that there are quite a few problems with the Lambek calculus as well.

[11]This assignment overgenerates, since it allows for "#Captain Jack served lobster yesterday and bananafish" as well. I believe a more detailed treatment of coordination would need to take into account discourse notions such as parallelism and contrast to accept and reject these cases. For example, **?** argue that the acceptability of a sentence with coordination depends on the similarity of the conjoined phrases.

1.5.1 Formal language

The Lambek calculus generates only context-free languages (**?**). There is good evidence that at least some constructions in natural language require a slightly larger class of languages (**?**). One influential proposal for such a larger class of languages are the *mildly context-sensitive languages* (**?**), characterised by the following properties:

- contains the context-free languages,
- limited cross-serial dependencies; prototypical mildly context-sensitive patterns outside of the context-free class include patterns such as the copy language $\{ww|w \in (a|b)^*\}$, multiple counting dependencies $a^n b^n c^n$ and 'crossed' counting dependencies $a^n b^m c^n d^m$,
- semilinearity (a language is semilinear iff there exists a regular language to which it is equivalent up to permutation),
- polynomial fixed recognition.

The last two items are sometimes stated as the weaker condition 'constant growth' instead of semilinearity and the stronger condition of polynomial parsing instead of polynomial fixed recognition. Since all other properties are properties of formal languages, I prefer the formal language theoretic notion of polynomial fixed recognition.

The class of mildly context-sensitive languages is a very robust class: many independently proposed grammars formalisms turn out to fall into this class (**?**, **?**). There is some further structure in the class of mildly context-sensitive languages: a first parameter gives a measure on the *number* of counting dependencies we can handle (for example, the language $a^n b^n c^n d^n e^n f^n$, with six counting dependencies, requires more a complex formalism than the language $a^n b^n c^n d^n$ with four counting dependencies; context-free languages already handle two counting dependencies, of course), a second parameter is whether the language class is well-nested or not. The class of languages generated by tree adjoining grammars (linear indexed grammars and combinatory categorial grammars generate the same class of languages) is the smallest class of the mildly context-sensitive languages: it is well-nested and handles counting dependencies of up to four.

1.5.2 Syntax-semantics interface

We are not just interested in generating the right string language for a given natural language, we also want to assign grammatical strings a representation of their possible meanings. The Lambek calculus fails to do this even for elementary quantifier scope and medial extraction facts.

In this section, we will briefly look at some of the types of phenomena which have interested researchers in type-logical grammars, and to which different solutions have been proposed. Since a finite list of cases can always be treated by some additional lexical type assignments, we will be interested only in *robust* solutions, that is solutions which generalise beyond the listed examples to more complex cases, and ideally to different *phenomena*.

As we have seen, the Lambek calculus gives a good and simple account of phenomena such as right node raising.

(13) Howard loved but Geoffrey hated "Syntactic Structures".

Example (13), repeated from Example (10), shows right node raising in the Lambek calculus: assuming assignments of $(np \backslash s) / np$ to the transitive verbs and np to "Howard" and "Geoffrey", **L** derives "Howard loved" and "Geoffrey hated" as expressions of type s / np. Although this works correctly, as we have seen in Section 1.4, it crucially depends on the presence of associativity. For example, **NL** no longer allows us to derive sentence (10) unless we add a second lexical assignment $np \backslash (s / np)$ to the transitive verbs.

Medial and other types of extraction

Extraction from non-peripheral positions, as shown in Example (14) below, shows that the Lambek calculus treatment of extraction is insufficiently general.

(14) Peter bought the book which John read yesterday.

Although the sentence above would be derivable without the sentence-final adverb "yesterday", again since "John read" is an expression of type s / np, the presence of "yesterday" blocks this derivation. In other words, the deep structure derivation looks as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{John}}{np}\,Lex \quad \dfrac{\dfrac{\text{read}}{np \multimap (np \multimap s)}\,Lex \quad [np]^1}{np \multimap s}\,\multimap E}{s}\,\multimap E \quad \dfrac{\text{yesterday}}{s \multimap s}\,Lex}{s}\,\multimap E}{np \multimap s}\,\multimap I_1$$

Replacing all "easy" linear logic implications "$\multimap$" by the corresponding Lambek calculus implications (either "/" or "\" depending on whether the corresponding argument is to the right or to the left) gives us the following.

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{John}}{np}\,Lex \quad \dfrac{\dfrac{\text{read}}{(np\backslash s)/np}\,Lex \quad [np]^1}{np\backslash s}\,\multimap E}{s}\,\multimap E \quad \dfrac{\text{yesterday}}{s\backslash s}\,Lex}{s}\,\multimap E}{np \multimap s}\,\multimap I_1$$

The "$\multimap I$" rule can not be turned into a Lambek calculus introduction rule, since the noun phrase hypothesis does not occur in a peripheral position.

Extraction is still more complicated than this. In theoretical linguistics, there are some classic principles such as the so-called coordinate structure constraint and the across-the-board exception (**?**) (**?**, Section 15.6). As usual, an asterisk '*' before a sentence denotes ungrammaticality.

(15) *Peter bough the book which John read "Syntactic Structures" and Mindy liked.

Examples like (15), and related examples, were first discussed as problems for the (associative) Lambek calculus by **?**. The problem with this ungrammatical sentence is that "John read *Syntactic Structures* and Mindy liked" is a sentence missing a noun phrase (at its right edge) and can therefore combine with "which" given its standard Lambek calculus assignment $(n \,\backslash\, n) \,/\, (s \,/\, np)$. The same derivability pattern which helped us for Sentence (10) leads to overgeneration for Sentence (15).

In mainstream linguistics, the coordinate structure constraint disallows extraction from coordinations and therefore blocks examples like (15). This constraint states that extraction out of coordinate structures (that is, phrases of the form "p_1 and/or p_2") is not allowed. It also blocks examples like the following.

(16) a. *This is the student which the principal suspended $[]_{np}$ and Barry.
b. *This is the student which the principal suspended Barry and $[]_{np}$.

Of these examples, only Sentence (16-b) is a problem for the Lambek calculus, since the missing noun phrase occurs at the right edge, just like for Sentence (15). However, once we add an operator allowing medial extraction to our logic, we need to be careful to disallow *all* of the examples (15)-(16).

The coordinate structure constraint has a standard exception: when all the coordinated constituents are missing the same material, then the combination of coordination and extraction is perfectly fine.

(17) This is the student which the principal suspended $[]_{np}$ this morning and the teacher defended $[]_{np}$ this afternoon.

In Sentence (17) above, two sentences with a medial noun phrase gap are conjoined. In the deep structure, this is just a standard coordination of two $np \multimap s$ expressions.

The reader who thinks by this point that we are starting to have rather many stipulated constraints with respect to extraction and coordination, and who would prefer that these facts would all fall out from more general principles is thinking like a type-logical grammarian (or, more generally, a formal linguist, since the goal to derive these constraints from more fundamental principles has played an important role in nearly all grammatical theories, not just type-logical grammars). The challenge in type-logical grammars is therefore to ensure that our treatment of coordination and extraction (whatever they will turn out to be) allows us to generate all and only the right patterns when they interact, but without using any extra-logical principles.

Quantifier scope

Example (18) below illustrates that the Lambek calculus has problems with quantifiers in medial position taking wide scope.

(18) John believes someone left.

This is one of the classic examples in semantics (and philosophy of language). Example (18) has two readings: for the the first, called the "de dicto" reading, the verb "believes" has wider scope than the quantifier "someone" (this can be true when John has heard to door slam and concludes from this weak evidence that someone left; this reading doesn't commit the speaker to believing anyone left), for the second reading, called the "de re" reading, there is a specific person, say Peter, whom John believes has left (this does commit the speaker to believing someone has left).

The narrow scope reading for "someone" is obtained from the following deep structure.

$$\dfrac{\dfrac{\text{John}}{np}\,Lex \qquad \dfrac{\dfrac{\text{believes}}{s \multimap (np \multimap s)}\,Lex \qquad \dfrac{\dfrac{\text{someone}}{(np \multimap s) \multimap s}\,Lex \qquad \dfrac{\text{left}}{np \multimap s}\,Lex}{s}\,\multimap E}{np \multimap s}\,\multimap E}{s}\,\multimap E$$

For the proof above, it is easy to replace the different occurrences of the linear logic implication '$\multimap$' by the Lambek slashes '\' and '/' in order to obtain a Lambek calculus proof. However, the second reading has the following deep structure.

$$\dfrac{\dfrac{\text{someone}}{(np \multimap s) \multimap s}\,Lex \qquad \dfrac{\dfrac{\dfrac{\text{John}}{np}\,Lex \qquad \dfrac{\dfrac{\text{believes}}{s \multimap (np \multimap s)}\,Lex \qquad \dfrac{[np]^1 \qquad \dfrac{\text{left}}{np \multimap s}\,Lex}{s}\,\multimap E}{np \multimap s}\,\multimap E}{s}\,\multimap E}{np \multimap s}\,\multimap I_1}{s}\,\multimap E$$

This deep structure again has a noun phrase in non-peripheral position, and the Lambek calculus lacks a way of withdrawing the *np* from the middle of the sentence (as for the medial extraction case) but also of 'moving' the quantifier "someone" back to the place of the withdrawn noun phrases.

The data are again more complicated (**?**): while it is certainly a relatively good first approximation that quantifiers can scope from anywhere, just like with extraction there are many restrictions. Some of these restrictions are similar to the ones of extraction. For example, we disallow some quantifiers in embedded clauses to takes scope outside of their clause, and therefore Sentence (19) can not mean that for each student there is a different professor who said this student was lazy. The same logical mechanisms used to exclude extraction cases like (15) and (16) would ideally block this unavailable reading as well.

(19) A professor said all students are lazy.

Then there are general restriction as to which type of quantifier can take scope over which other one, or over negation. So for Sentence (20-a), there is both a $\neg\exists$ reading (where Kim didn't pass *any* test) and a $\exists\neg$ reading (where there is a test which Kim didn't pass, but possibly a few others she did pass). Sentence (20-b) has only the $\neg\exists$ reading[12] (equivalent to the $\neg\exists$ reading of (20-a)).

(20) a. Kim didn't pass a test.
b. Kim didn't pass any test.

Dutch verb clusters

Another classic example of a problem with the Lambek calculus is the treatment of Dutch verb clusters (**?**, **?**), illustrated by sentences such as the following.

(21) (dat) Jan Marie de nijlpaarden zag voeren.

(22) (dat) Jan Henk Marie de nijlpaarden zag helpen voeren.

These sentences exhibit the well-known crossed dependencies: in (22) "Henk" is the object of "zag" (saw), "Marie" the object of "helpen" (help) and "de nijlpaarden" (the hippopotami) the object of "voeren" (feed), as shown in the deep structure below.

$$\dfrac{\dfrac{\dfrac{\text{zag}}{np \multimap (inf \multimap (np \multimap s))}\,Lex \quad \dfrac{\text{Henk}}{np}\,Lex}{inf \multimap (np \multimap s)}\,{\multimap}E \qquad \dfrac{\dfrac{\dfrac{\text{Marie}}{np}\,Lex \quad \dfrac{\text{helpen}}{np \multimap (inf \multimap inf)}\,Lex}{inf \multimap inf}\,{\multimap}E \quad \dfrac{\dfrac{\text{de nijlpaarden}}{np}\,Lex \quad \dfrac{\text{voeren}}{np \multimap inf}\,Lex}{inf}\,{\multimap}E}{inf}\,{\multimap}E}{np \multimap s}\,{\multimap}E$$

Although **?** show that such examples can be treated by context-free grammars — and hence by the Lambek calculus — the analysis of mildly context-sensitive formalisms is generally preferred, since it expresses the desired dependencies between objects and verbs. For example, a fairly direct translation

[12]Negative polarity would be one possible explanation. Negative polarity items are words like "any" but also expressions like "lift a finger" which are required to be in the scope a a negation-like operator. This is an important topic in the syntax-semantics which a footnote can not do justice, but about which I will have very little to say.

of the grammar given by **?** into a Lambek grammar gives the following proof for Sentence (22).

$$\dfrac{\dfrac{\text{Henk}}{np}\,Lex \qquad \dfrac{\dfrac{\dfrac{\text{Marie}}{np}\,Lex \qquad \dfrac{\dfrac{\dfrac{\text{de nijlpaarden}}{np}\,Lex \quad \dfrac{\text{zag}}{np\backslash(np\backslash s)}\,Lex}{np\backslash s}\,\backslash E \quad \dfrac{\text{helpen}}{(np\backslash s)\backslash(np\backslash(np\backslash s))}\,Lex}{np\backslash(np\backslash s)}\,\backslash E}{np\backslash s}\,\backslash E \qquad \dfrac{\text{voeren}}{(np\backslash s)\backslash(np\backslash(np\backslash s))}\,Lex}{np\backslash(np\backslash s)}\,\backslash E}{np\backslash s}\,\backslash E$$

The two analyses produce different lambda terms for their derivational semantics, with the semantics of the derivation inspired by **?** shown as 1.1 and the semantics of the deep structure proof inspired by the mildly context-sensitive analysis shown as 1.2.

$$((\mathit{voeren}\,((\mathit{helpen}\,(\mathit{zag}\,\mathit{de\_nijlpaarden}))\;\mathit{Marie}))\;\mathit{Henk}) \tag{1.1}$$
$$((\mathit{zag}\,\mathit{Henk})\,((\mathit{helpen}\,\mathit{Marie})\,(\mathit{voeren}\,\mathit{de\_nijlpaarden}))) \tag{1.2}$$

An obvious advantage of deep structure 1.2 above is that we directly obtain the correct semantics, whereas the Pullum and Gazdar analysis of 1.1 would require us to abandon the simple connection to meaning and add a dedicated 'Dutch verb cluster semantics' component to the syntax-semantics interface.

Gapping, ellipsis, comparatives

A number of other interesting phenomena have been treated in the literature on type-logical grammars. I will give only a brief analysis of a few of them here, because they have been studied in a number of different type-logical frameworks. The reader is referred to (**?**, **?**, **?**, **?**) for more detailed analysis of the phenomena involved. As usual, I will only present the phenomena from the deep structure, linear logic point of view, since this is where most analyses converge. We will see many of these examples again when we investigate different proposals for the surface structure in type-logical grammars.

Gapping is exemplified by Sentence (23) below.

(23) John studies logic and Charles, phonetics.

From the semantic point of view this sentence has the same meaning as "John studies logic and Charles *studies* phonetics"

Ellipsis is another large class of phenomena, but Sentence (24) shows a simple example of verb phrase ellipsis.

(24) John left before Mary did.

This sentence has more or less the same meaning as "John left before Mary *left*" and this is what we want our analysis to find.

A final example are comparatives, such as the following example.

(25) John bought more books than Mary ate bagels.

Sentence (25) means something like "The number of books John bought is greater than the number of bagels Mary ate").

Again the deep structure proofs in multiplicative linear logic for these sentences are fairly simple. For Sentence (23), the deep structure proof looks as follows (where *tv* is short for $np \multimap np \multimap s$ and X is short for $tv \multimap s$, that is, $(np \multimap np \multimap s) \multimap s$).

$$\dfrac{\dfrac{\text{studies}}{tv}\,Lex \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{John}}{np}\,Lex \quad \dfrac{[tv]^1 \quad \dfrac{\text{logic}}{np}\,Lex}{np \multimap s}\multimap E}{s}\multimap E}{tv \multimap s}\multimap I_1 \qquad \dfrac{\dfrac{\text{and}}{X \multimap (X \multimap X)}\,Lex \quad \dfrac{\dfrac{\dfrac{\text{Charles}}{np}\,Lex \quad \dfrac{[tv]^2 \quad \dfrac{\text{phonetics}}{np}\,Lex}{np \multimap s}\multimap E}{s}\multimap E}{tv \multimap s}\multimap I_2}{X \multimap X}\multimap E}{tv \multimap s}\multimap E}{s}\multimap E$$

In the proof, we derive both "John *tv* logic" and "Charles *tv* phonetics" as sentences, then withdraw the two hypothetical transitive verbs, combine the two sentences (missing transitive verbs) using "and", and complete the derivation by combining the resulting $tv \multimap s$ with the transitive verb "studies".

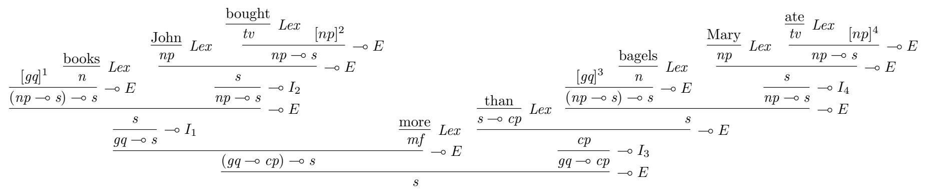
In other words, we have analysed the apparent copying of the transitive verb "studies" by coordinating two sentences missing a transitive verb. We therefore end up with the standard coordination scheme, but of sentences missing transitive verbs. To complete the analysis and obtain the right semantics, we only need to add the lexical semantics $\lambda Q \lambda P \lambda v.(P\, v) \wedge (Q\, v)$ for "and". Note that copying only happens at the level of the meaning: the verb meaning v is copied in the semantic term to fill the role of the missing transitive verb in both sentences.

Sentence (24) is analysed as follows. The formula *vp* abbreviates $np \multimap s$.

$$\dfrac{\dfrac{\text{John}}{np}\,Lex \qquad \dfrac{\dfrac{\text{left}}{vp}\,Lex \qquad \dfrac{\dfrac{\dfrac{\dfrac{[vp]^1 \quad \dfrac{\dfrac{\text{before}}{s \multimap (vp \multimap vp)}\,Lex \quad \dfrac{\dfrac{\text{Mary}}{np}\,Lex \quad [vp]^2}{s}\multimap E}{vp \multimap vp}\multimap E}{vp}\multimap E}{vp \multimap vp}\multimap I_1}{vp \multimap (vp \multimap vp)}\multimap I_2 \quad \dfrac{\text{did}}{(vp \multimap (vp \multimap vp)) \multimap (vp \multimap vp)}\,Lex}{vp \multimap vp}\multimap E}{np \multimap s}\multimap E}{s}\multimap E$$

The above proof is slightly more complicated: we analyse "before Mary" as a verb phrase missing two verb phrases, which is the argument that "did" requires. After combining with "did", we complete the proof by applying the verb phrase "left" (produce a new verb phrase) and finally noun phrase "John" to produce a sentence. Again, the lexical semantic term for "did" completes the analysis. We use $\lambda P \lambda v.(P\, v)\, v)$. Since P is the term corresponding to the meaning of a verb phrase with two missing verb phrases ("*vp* before Mary *vp*" in our case), and v is the meaning of the verb phrase ("left" in our case), this copies the verb phrases meaning to the two places from which a *vp* has been extracted. There is again no need for copying in the syntax, all the required copying is done in the semantic term.

The final sentence is analysed as follows. In the proof, mf abbreviates $(gq \multimap s) \multimap (gq \multimap cp) \multimap s$, with gq abbreviating $n \multimap (np \multimap s) \multimap s$.



In the proof, we have a sentence missing a generalised quantifier "John bought gq books" and a complementiser phrase cp also missing a generalised quantifier "than Mary ate gq bagels". Both serve as arguments to "more" to produce a sentence[13]. Completing the analysis requires providing the semantics term. This is a bit more involved than it was for the other cases. Let the term $\mathcal{I}$ be equal to $\lambda z \lambda P \lambda Q.(P\, z) \wedge (Q\, z)$, let the circumfix operator $|.|$ denote the term of type $(e \rightarrow t) \rightarrow \mathbb{R}$ (with $\mathbb{R}$ the real numbers, of some standard approximation of them, such as floating point numbers), and let $>$ be the usual greater than operator. Then the term assigned to "more" is $\lambda P \lambda Q.|\lambda x.(P\,(\mathcal{I}\, x))| > |\lambda y.(Q\,(\mathcal{I}\, y))|$, "than" is simply assigned the identity function $\lambda x.x$. Let's take a closer look at what's going on with the term assigned to "more". The abstractions over P and Q represent the two sentences each missing a generalised quantifier. The generalised quantifier type is $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$. That is, a function taking (the characteristic function of) two sets as arguments to produce a truth value. For the first sentences these functions are the set of books (with type n) and the set of things John bought (with type $np \multimap s$), for the second the set of bagels and the set of things Mary ate. In both cases, we want to take the intersection of these two sets, then compare the number of items in both intersections. This is what the term for "more" does: $(\mathcal{I}x)$ takes the intersection of two sets (conjoining them as properties of x). Since the term $(\mathcal{I}x)$ is of the generalised quantifier type, terms of this form can be the argument of P and Q, where they will serve as the missing quantifier. We then count the number of x elements in the first intersection and compare it to the y in the second intersection.

The linguistic phenomena described in this section give only a very partial picture of the descriptive work done in the various type-logical frameworks, *and* for which solutions have been proposed (for a more detailed treatment of many more phenomena, I recommend **?**, **?**, **?**). In later chapters, we will see quite a few of these solution-framework combinations. However, I hope this section has at least given an idea of the many challenges addressed by type-logical grammars. It also provides a sort of 'checklist' to evaluate and compare

[13]Some alternative analyses are possible here, using different ways to divide the analysis between "more" and "than". Arguably, "more...than" should be analysed as a single complex lexical entry. **?** give an analysis with a lexical entry for "-er...than", where "-er" is the comparative morpheme (transforming, for example, "good-er" to "better").

different formalisms. This checklist will of course need to be expanded as more and more linguistic phenomena are successfully treated.

1.6 Roadmap

Even though the Lambek calculus is a very simple system which gives a basic account of several interesting phenomena on the syntax-semantics interface, we want a logical system which solves the problems with Lambek grammars while at the same time not making the logic overly complicated. We want a logic which permits a better treatment of extraction, quantifier scope and many other phenomena, as well as of the correct *interactions* between all these phenomena.

From the eighties onwards, many formalisms have been proposed often with the intention to solve all or most of the problems of the Lambek calculus indicated in Section 1.5. Chapter 2 will provide a brief overview of these different calculi. Given that there are many competing formalisms, the current landscape of different type-logical grammars may look confusing. One of the main goals of this book will be to categorise and compare these different formalisms. For this comparison, I will propose two general frameworks, using a proof-theoretic innovation introduced for linear logic called *proof nets*. Chapter 3 will provide the necessary background on proof nets. One of the main research questions of this book is how to adapt the proof nets of linear logic to these different modern type-logical grammars, in a way which facilitates comparison between formalisms using what appear to be a priori rather different logical primitives.

The first general framework for type-logical grammar I propose is first-order multiplicative linear logic (**?**). This is a simple, standard fragment of linear logic. However, several type-logical grammars, including Lambek grammars, lambda grammars, hybrid type-logical grammars, and a large fragment of the Displacement calculus can be translated into first-order linear logic. Fist-order linear logic will be the main topic of Chapter 4.

The second general framework is inspired by the interaction nets of **?** and the multimodal proof nets of **?**. It is a proof net framework based on graph rewriting, which captures *all* current multiplicative type-logical grammar frameworks. These proof-nets-as-graph-rewriting will be the main topic of Chapter 5.

Finally, although the formal properties of our logics are important, we will be interested in theorem proving for these different formalisms. Theorem proving is important in two different ways: firstly, it is important for the grammar designer to test the predictions of his grammar. A number of theorem provers for different type-logical grammars have been developed as part of my research with this goal in mind (**?**, **?**, **?**). A second way this is important is in terms of applications to wide-coverage semantics. Notably, I will be interested in generating logical meaning representations for arbitrary French and Dutch text. While generating formal representations of meaning is only a stepping stone to applications in natural language understanding, I believe it to be an important one. This last question will occupy us in Chapter **??**.

2 Modern type-logical grammars

We have seen in the previous chapter that the Lambek calculus gives a fairly good account of a number of phenomena on the syntax-semantics interface, but fails for many more sophisticated ones. As a result, many variants and extensions of the Lambek calculus have been proposed. In this chapter, we will briefly look at those variants of the Lambek calculus which stay fully in the logical tradition initiated by **?**.

Large parts of this chapter have been taken from the forthcoming Stanford Encyclopedia of Philosophy entry on type-logical grammars (**?**).

2.1 Multimodal type-logical grammars

One of the earliest extensions of the Lambek calculus were so-called *multimodal* type-logical grammars. This idea can be traced back to **?** and to **?**. The basic idea is to have multiple *families* of connectives. Given a set of modes I (decided by the grammar writer), we have for each $i \in I$ a family of connectives $/_i$, $\bullet_i$ and $\backslash_i$. The Lambek calculus **L** is a special case where I is a singleton set (the same holds for its non-associative variant **NL**, and its associative, commutative variant **LP**). For a truly multimodal logic, there is a small set of modes, at least two. In terms of structures, moving from the (associative, commutative) Lambek-van Benthem calculus **LP** to the (associative) Lambek calculus **L** to the non-associative Lambek calculus **NL**, and finally to multimodal categorial grammars can be seen as follows[1].

[1] There is, of course, the possibility to add commutativity but not associativity, giving the logic **NLP**.

$$\frac{\Delta \vdash A \bullet_i B \quad \Gamma[(A \circ_i B)] \vdash C}{\Gamma[\Delta] \vdash C}\ [\bullet E] \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{(\Gamma \circ_i \Delta) \vdash A \bullet_i B}\ [\bullet I]$$

$$\frac{\Gamma \vdash A/_iB \quad \Delta \vdash B}{(\Gamma \circ_i \Delta) \vdash A}\ [/E] \qquad \frac{(\Gamma \circ_i B) \vdash A}{\Gamma \vdash A/_iB}\ [/I]$$

$$\frac{\Gamma \vdash B \quad \Delta \vdash B\backslash_i A}{(\Gamma \circ_i \Delta) \vdash A}\ [\backslash E] \qquad \frac{(B \circ_i \Gamma) \vdash A}{\Gamma \vdash B\backslash_i A}\ [\backslash I]$$

Table 2.1: Multimodal natural deduction rules

$$\frac{\Gamma[\Delta_1 \circ_a (\Delta_2 \circ_a \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_a \Delta_2) \circ_a \Delta_3] \vdash C}\ Ass_1 \qquad \frac{\Gamma[(\Delta_1 \circ_a \Delta_2) \circ_a \Delta_3] \vdash C}{\Gamma[\Delta_1 \circ_a (\Delta_2 \circ_a \Delta_3)] \vdash C}\ Ass_2$$

$$\frac{\Gamma[\Delta_1 \circ_a \Delta_2] \vdash C}{\Gamma[\Delta_1 \circ_n \Delta_2] \vdash C}\ I_{a,n}$$

Table 2.2: The structural rules for associativity for mode a and inclusion between modes a and n in a multimodal logic

Logic	Structural rules	Structure
LP	associativity, commutativity	multiset
L	associativity	list
NL	none	binary tree
multimodal	mode-specific	labeled binary tree

That is, the structure of the antecedent becomes gradually more fine-grained, adding linear order, hierarchical structure and finally labelling on the nodes of the tree. For the natural deduction rules, this changes relatively little. First, to make our antecedents more easily readable, we will write the binary term constructor as an infix $\circ_i$ (for all modes i in the grammar). Compared to using indexed parentheses $(\ldots)^i$ this makes the mode information visible locally at each branch, and allows us to omit outer brackets without loss of information.

The natural deduction rules are shown in Table 2.1. In each rule, the index i is shared between the connective and the structure which is created or removed.

In their simplest form, a multimodal grammar has different modes with different structural rules. For example, we can have a multimodal logic with two modes n and a where n is a non-associative mode and a an associative one.

In such a logic, we have that $a/_ab \circ_a b_ac \vdash a/_ac$ but $a/_nb \circ_n b/_nc \nvdash a/_nc$. This gives us fine-grained lexical control over when to allow associativity. However, in such a setup, there are essentially two isolated logics, an associative one and a non-associative one and there is no possibility of communication

$$\dfrac{\Gamma[\Delta_1 \circ_2 (\Delta_2 \circ_1 \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_2 \Delta_2) \circ_1 \Delta_3] \vdash C}\, MA \qquad \dfrac{\Gamma[\Delta_2 \circ_2 (\Delta_1 \circ_1 \Delta_3)] \vdash C}{\Gamma[\Delta_1 \circ_1 (\Delta_2 \circ_2 \Delta_3)] \vdash C}\, MC$$

Table 2.3: The structural rules for mixed associativity and mixed commuatativity for modes 1 and 2

$$\dfrac{\Gamma[\Delta_1 \circ_a (\Delta_2 \circ_a \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_n \Delta_3) \circ_w \Delta_2] \vdash C}\, Wr_{-1} \qquad \dfrac{\Gamma[(\Delta_1 \circ_n \Delta_3) \circ_w \Delta_2] \vdash C}{\Gamma[\Delta_1 \circ_a (\Delta_2 \circ_a \Delta_3)] \vdash C}\, Wr$$

Table 2.4: The wrap/split structural rules

between the two. Interaction postulates are structural rules which mention more than one distinct mode. The simplest type of interaction postulate is an inclusion postulate, such as the inclusion between the associative and non-associate mode shown in Table 2.2[2].

One influential type of structural rule are the so-called mixed associativity and mixed commutativity rules of **?**. Table 2.3 shows these rules for a mode 1 and a mode 2.

The mixed commutativity rule provides an important step towards the analysis of verb clusters in Dutch subordinate clauses. For example, it allows us to derive "(dat) Marie boeken wil lezen" (*that Marie wants to read books*) as follows.

$$\dfrac{\dfrac{\text{Marie} \vdash np \qquad \dfrac{\text{wil} \vdash (np\backslash_1 s)/_1 inf \qquad \dfrac{\text{boeken} \vdash np \qquad \text{lezen} \vdash np\backslash_2 inf}{\text{boeken} \circ_2 \text{lezen} \vdash inf}\,\backslash E}{\text{wil} \circ_1 (\text{boeken} \circ_2 \text{lezen}) \vdash np\backslash s}\,/E}{\text{Marie} \circ_1 (\text{wil} \circ_1 (\text{boeken} \circ_2 \text{lezen})) \vdash s}\,\backslash E}{\text{Marie} \circ_1 (\text{boeken} \circ_2 (\text{wil} \circ_1 \text{lezen})) \vdash s}\, MC$$

However, this analysis still needs to be improved to prevent overgeneration: we need a mechanism to ensure the verbs appear together at the end of the phrase (**?**), which the analysis above fails to do. **?** and **?** provide a much more detailed multimodal analysis of Dutch verb clusters, which includes the unary connectives of the next section.

As a final example of an influential set of multimodal structural rules, Table 2.4 lists the 'wrap' structural rules from **?**. In these rules a is a standard associative mode (with an identity element ϵ), and n is a standard non-associative mode. The Wr_{-1} rule allows us to move a structure Δ_2 outside

[2]There have historically been differences of opinion with respect to the direction of the inclusion between more permissive modes with respect to more restrictives ones. Many authors prefer the inverse inclusion relations (**?**, **?**). From the point of view of linear logic, one way to see this difference is whether our desired interpretation of $A \otimes B$ is something like $(A \bullet B) \& (B \bullet A)$ (the interpretation of **?**, **?**) or more as $(A \bullet B) \oplus (B \bullet A)$ (the interpretation of **?**).

as the rightmost argument of a wrap mode w, but combining the remnant structures Δ_1 and Δ_3 together using the non-associative mode n. This fixes the point from which Δ_2 was removed, guaranteeing that when we move Δ_2 back in using the Wr rule, we return it to the same place (this is, of course, primarily interesting when Δ_2 has changed due to to other logical rules).

The wrap/split rules provide a solution to quantifier scope using the formula $s/_w(np\backslash_w s)$ for words like "someone". The following derivation illustrates this, using the "someone" wide scope reading of "John believes someone left" as an example.

$$
\begin{prooftree}
\AxiomC{$\text{John} \vdash np$}
\AxiomC{$\text{believes} \vdash (np\backslash_a s)/_a s$}
\AxiomC{}
\RightLabel{Ax}
\UnaryInfC{$np \vdash np$}
\AxiomC{$\text{left} \vdash np\backslash_a s$}
\RightLabel{$\backslash E$}
\BinaryInfC{$np \circ_a \text{left} \vdash s$}
\RightLabel{$/E$}
\BinaryInfC{$\text{believes} \circ_a np \circ_a \text{left} \vdash np\backslash_a s$}
\RightLabel{$\backslash E$}
\BinaryInfC{$\text{John} \circ_a \text{believes} \circ_a np \circ_a \text{left} \vdash s$}
\RightLabel{Wr_{-1}}
\UnaryInfC{$((\text{John} \circ_a \text{believes}) \circ_n \text{left}) \circ_w np \vdash s$}
\RightLabel{$/I$}
\UnaryInfC{$((\text{John} \circ_a \text{believes}) \circ_n \text{left}) \vdash s/_w np$}
\AxiomC{$\text{someone} \vdash (s/_w np)\backslash_w s$}
\RightLabel{$\backslash E$}
\BinaryInfC{$((\text{John} \circ_a \text{believes}) \circ_n \text{left}) \circ_w \text{someone} \vdash s$}
\RightLabel{Wr}
\UnaryInfC{$\text{John} \circ_a \text{believes} \circ_a \text{someone} \circ_a \text{left} \vdash s$}
\end{prooftree}
$$

To keep the proof simple, we have left associativity of a implicit by not placing any brackets. The proof starts deriving "John believes np left" as a sentence. Then, the Wr_{-1} rule is used to move the np formula to the rightmost position with a wrap mode w. The pair of strings "John believes" and "left" joined by non-associative mode n marks the place of the np formula by separating the two strings to its left and right (that is, non-associativity fixes the point of extraction). We withdraw the np hypothesis, then combine the resulting proof with the lexical entry for "someone" using the $\backslash E$ rule. We complete the proof by using the wrap rule Wr to move "someone" back to the position from which we removed the noun phrase.

To avoid overgeneration, we require that our output cannot contain mode w. Otherwise, we could have ended the proof before the last Wr rule to obtain "John believes left someone" as an alternative, valid word order for this reading. It is standard practice in multimodal grammars to allow only a subset of the modes of a grammar to appear in the endsequent.

2.1.1 Unary residuated connectives

The earliest unary connectives added to type-logical grammars were inspired by the linear logic exponential '!'. Whereas the linear logic exponential marked formulas as specifically permitted to be weakened and contracted, the Lambek calculus exponential marked formulas as allowing the structural rule of commutativity. The left and right rule were the standard promotion and dereliction rule (**?**, **?**), essentially the sequent calculus rules for the modal logic **S4**.

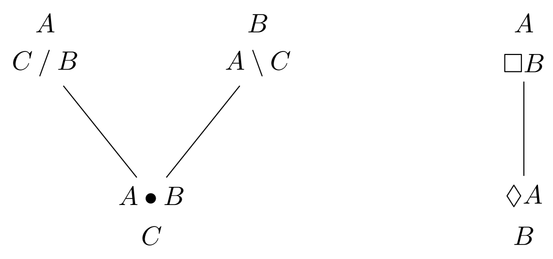


Figure 2.1: Visual representation binary and unary residuation

$$\frac{\Delta \vdash \Diamond_j A \quad \Gamma[\langle A\rangle^j] \vdash C}{\Gamma[\Delta] \vdash C}\ \Diamond E \qquad \frac{\Gamma \vdash A}{\langle\Gamma\rangle^j \vdash \Diamond_j A}\ \Diamond I$$

$$\frac{\Gamma \vdash \Box_j A}{\langle\Gamma\rangle^j \vdash A}\ \Box E \qquad \frac{\langle\Gamma\rangle^j \vdash A}{\Gamma \vdash \Box_j A}\ \Box I$$

Table 2.5: Natural deduction rules for the unary connectives

? proposed an alternative way of adding unary connectives to type-logical grammars[3]. Whereas the binary connectives '\', '•' and '/' are a residuated triple, the unary connectives '□' and '◊' are a residuated pair, as indicated by the following equations.

$$A \to C/B \quad \Leftrightarrow \quad A \bullet B \to C \quad \Leftrightarrow \quad B \to A\backslash C \tag{2.1}$$

$$A \to \Box B \quad \Leftrightarrow \quad \Diamond A \to B \tag{2.2}$$

The advantage of this setup is that, from the point of view of linear logic, we stay inside the multiplicative fragment and avoid the more complicated proof machinery required for the exponentials. This is especially useful for the proof nets which will be introduced in Section 5.1.3 and which will play an important role in the rest of this book.

We can display residuation visually as shown in Figure 2.1 (**?**, Section 4.2.1). In the same way each node in the binary branch corresponds to one of the three binary connectives, each node in the unary branch corresponds to one of the unary connectives, with $\Diamond$ a unary version of $\bullet$ and $\Box$ a unary version of the implications.

Just like we can have multiple families $i \in I$ of binary connectives, we can allow multiple families $j \in J$ of unary connectives, both for small sets I of binary families and J of unary families. From the point of view of the allowed structures, the unary connectives add (labeled) unary branches to the

[3]The bracket/antibracket operators of **?** are a similar proposal, but the setup of **?** is cleaner and has the required proofs of cut-elimination, and of soundness and completeness with respect to the Kripke models.

$$\frac{\Gamma[\Delta_1 \circ (\Delta_2 \circ \langle \Delta_3 \rangle)] \vdash C}{\Gamma[(\Delta_1 \circ \Delta_2) \circ \langle \Delta_3 \rangle] \vdash C} MA \qquad \frac{\Gamma[(\Delta_1 \circ \langle \Delta_3 \rangle) \circ \Delta_2] \vdash C}{\Gamma[(\Delta_1 \circ \Delta_2) \circ \langle \Delta_3 \rangle] \vdash C} MC$$

Table 2.6: Version of the mixed associativity and mixed commutativity rules using unary branches

binary branches of previous multimodal grammars. We write unary branches as $\langle \ldots \rangle^j$, where the angular brackets should remind us these brackets are a structural reflection of the connective $\Diamond_j$ (just like $\circ_i$ reflects $\bullet_i$). Table 2.5 lists the natural deduction rules for the unary connectives.

? show that the unary connectives allow us to start in a very restricted logic, such as **NL**, and use the unary connectives to license the structural rules of associativity and commutativity. On the other hand, they also allow us to start in a very free logic, such as **LP**, and use the unary connectives to block associativity and commutativity.

The unary connectives also have a number of interesting derivability patterns, the simplest being the following.

$$\Diamond\Box A \vdash A \qquad\qquad A \vdash \Box\Diamond A \tag{2.3}$$

The first of these patters can be derived as follows.

$$\frac{\overline{\Diamond\Box A \vdash \Diamond\Box A}\ Ax \qquad \dfrac{\overline{\Box A \vdash \Box A}\ Ax}{\langle \Box A \rangle \vdash A}\ \Box E}{\Diamond\Box A \vdash A}\ \Diamond E$$

One standard way of implementing medial extraction in multimodal type-logical grammars is by assigning the relativiser the formula $(n\backslash n)/(s/\Diamond\Box np)$ (we are using only a single unary and binary mode here, and therefore suppress all indices, in a more sophisticated grammar, additional mode information will need to be added). This exploits that fact that, according to 2.3, $\Diamond\Box np \vdash np$ which allows us to use the 'marked' noun phrase as a normal noun phrase. However, we use versions of the mixed associativity and mixed commutativity structural rules, listed in Table 2.6, which (read from premiss to conclusion) allow us to move constituents marked with a unary branch outwards.

Taken together, this allows us to derive "John read yesterday" as being of type $s/\Diamond\Box np$. We can extend this proof "book which John read yesterday" as of type n using the assignment n to "book" and $(n\backslash n)/(s/\Diamond\Box np)$ to "which".

$$
\dfrac{\dfrac{}{\Diamond\Box np \vdash \Diamond\Box np}\,Ax \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{John} \vdash np \qquad \dfrac{\text{saw} \vdash (np\backslash s)/np \qquad \dfrac{\dfrac{}{\Box np \vdash \Box np}\,Ax}{\langle \Box np \rangle \vdash np}\,\Box E}{\text{read} \circ \langle \Box np \rangle \vdash np\backslash s}\,/E}{\text{John} \circ (\text{read} \circ \langle \Box np \rangle) \vdash s}\,\backslash E \qquad \text{yesterday} \vdash s\backslash s}{(\text{John} \circ (\text{read} \circ \langle \Box np \rangle)) \circ \text{yesterday} \vdash s}\,\backslash E}{((\text{John} \circ \text{read}) \circ \langle \Box np \rangle) \circ \text{yesterday} \vdash s}\,MA}{((\text{John} \circ \text{read}) \circ \text{yesterday}) \circ \langle \Box np \rangle \vdash s}\,MC}{\dfrac{((\text{John} \circ \text{read}) \circ \text{yesterday}) \circ \Diamond\Box np \vdash s}{(\text{John} \circ \text{read}) \circ \text{yesterday} \vdash s/\Diamond\Box np}\,/I}\,\Diamond E
$$

This provides an implementation of medial extraction in multimodal categorial grammars. **?** and **?** note that some interesting differences between Dutch and English follow from the choice between extraction from right branches, as is done here for English in Table 2.6, and the left-right symmetric choice of extraction from left branches for the analysis of Dutch.

An additional simple but important application of the unary connectives can be obtained by assigning the coordinator "and" the following formula $\forall X.(X\backslash\Box_0 X)/X$ (**?**, **?**). Combined with the previous solution to medial extraction, this would solve most of the problematic interactions between extraction and coordination we have seen before (as sentences (15) to (17) in Section 1.5[4]).

2.1.2 Discussion

Multimodal type-logical grammars are important in the history of type-logical grammars in that they allowed us to give a purely logical account of many proposed extensions of categorial grammars. Everything from proof theory to model theory works as it should, and a number of interesting linguistic phenomena have been treated in multimodal grammars (**?**, **?**, **?**, **?**, **?**). In addition, the multimodal structural rules are rather similar to 'movement' rules of mainstream syntactic theory (**?**, **?**, **?**).

There has also been some effort to cast some other type-logical grammars, notably the Displacement calculus (**?**) and $\mathbf{NL}_\lambda$ (**?**, Chapter 17), as instances of multimodal grammars. While such translations are always important and establish some useful points of comparison, our goal will be to provide general frameworks allowing many more of these points of comparison.

A problem with the multimodal calculus is that the set of allowed structural rules is extremely free, and this makes it hard to falsify multimodal type-logical grammars. It is always possible to add an extra mode, or some new structural rules. While a lot has certainly been done with relatively few structural rules — essentially the mixed associativity and commutativity rules we have seen here — there has been a move to logics with fixed sets of structural rules (or no structural rules at all), logics which do not fit neatly within the multimodal framework of this section.

[4]We need to be careful here that we use a different mode for licensing extraction than we do for blocking it.

In the remainder of this chapter, we'll take a brief tour of these different logics.

2.2 The Displacement calculus

The Displacement calculus is a discontinuous version of Lambek calculus that has been developed by Morrill and co-workers (**?**), building upon earlier discontinuous Lambek calculi (**?**, **?**) The Displacement calculus extends the associative Lambek calculus **L**. We have seen that **L** is the logic of strings composed by concatenation. The discontinuous calculi enrich the ontology with a notion of *split* strings: expressions consisting of detached parts, as in the idiom "take — to task". To build the phrase "take someone to task", one *wraps* the discontinuous expression around its object. In this particular example, there is a single point of discontinuity, but one can also think of cases with more than one split point. In the Displacement calculus, we call the number of split points in an expression (or the denotation of a formula) its *sort*.

The vocabulary of **D** (Displacement calculus) consists of residuated families of unary and binary type-forming operations. Some of the key connectives are given below. For the binary case, in addition to the concatenation product of **L** and the residual slash operations, we have a discontinuous (wrapping) product $\odot$, with residual infixation $\downarrow$ and extraction $\uparrow$ operations. For cases with multiple split points, the discontinuous type-forming operations have an indexed form $\uparrow_k, \odot_k, \downarrow_k$ explicitly referring to the k-th split point of their interpretations. The function of the unary operations is to control the creation and removal of split points.

The central equation of the Displacement calculus is the following, where 'W' denotes the binary wrap operation, '+' denotes string concatenation, '$\mathbf{1}$' a split point[5].

$$W(\alpha_1 + \mathbf{1} + \alpha_2, \beta) \Leftrightarrow \alpha_1 + \beta + \alpha_2 \tag{2.4}$$

It states that the wrap operator '$W(.,.)$' interacts with a structure containing a split point '$\mathbf{1}$' by replacing this split point by β (the wrapped argument). Returning to our previous example, given a term $W(\mathit{take}{+}\mathbf{1}{+}\mathit{to}{+}\mathit{task}, \mathit{someone})$, we rewrite using Equation 2.4 (with $\alpha_1 = \mathit{take}$, $\alpha_2 = \mathit{to} + \mathit{task}$ and $\beta = \mathit{someone}$) to $\mathit{take} + \mathit{someone} + \mathit{to} + \mathit{task}$.

In Equation 2.4, we allow α_1 and/or α_2 to be the empty string ϵ (that is, when the split point is the leftmost or the rightmost element of a structure, the wrap operation applies normally). The equation can be used in both directions. When there are multiple insertion points, the $W_k(.,.)$ operation requires α_1 to contain $k-1$ insertion points (making the insertion point shown in the equation the kth one). An alternative to $W_k(.,.)$ are the leftmost wrap $W_>(.,.)$ which requires α_1 to be without insertion points (that is, it replaces

[5]It is important to note that this use of '**1**' is very different from its use in linear logic, where **1** denotes the empty antecedent.

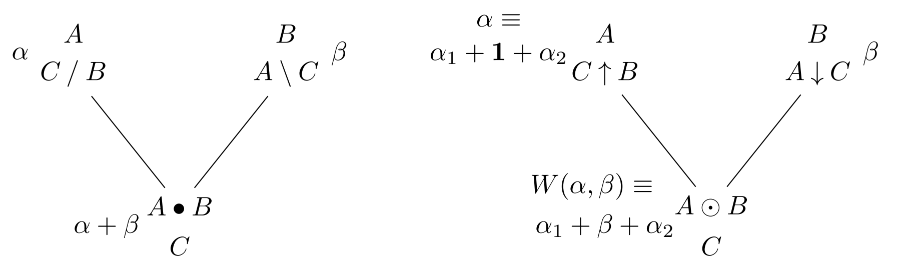

α A C / B
B $A \backslash C$ β
$\alpha + \beta$ $A \bullet B$ C
$\alpha \equiv \alpha_1 + \mathbf{1} + \alpha_2$ A $C \uparrow B$
B $A \downarrow C$ β
$W(\alpha, \beta) \equiv \alpha_1 + \beta + \alpha_2$ $A \odot B$ C

Figure 2.2: Visual representation of residuation for the Displacement calculus

the first insertion point by β), and rightmost wrap $W_<(.,.)$ which requires α_2 to be without insertion points (that is, it replaces the last insertion point by β).

One useful way to see the Displacement calculus as a logic with two triples of residuated connectives: the Lambek calculus connectives, defined with respect to concatenation '+' and the discontinuous connectives, defined with respect to $W(.,.)$. Figure 2.2 shows the two residuated triples side by side. However, for the operation $W(\alpha, \beta)$ to be well-defined, α needs to be of the form $\alpha_1 + \mathbf{1} + \alpha_2$. But if $\alpha \equiv \alpha_1 + \mathbf{1} + \alpha_2$, then $W(\alpha, \beta) \equiv W(\alpha_1 + \mathbf{1} + \alpha_2, \beta)$ and we can further simplify this, according to Equation 2.4, to $\alpha_1 + \beta + \alpha_2$. This means that we can 'compile away' the explicit applications of the wrap operation and formulate the logic as a labeled natural deduction calculus as shown in Table 2.7.

Note that this requires the A and $C \uparrow B$ hypotheses to be assigned a complex label containing a split point $\mathbf{1}$ (this amounts to spelling out variables α of sort k as complex terms $a_1 + \mathbf{1} + \ldots + \mathbf{1} + a_{k+1}$ with all variables a_i of sort 0).

The unary connectives $^\wedge$ and $^\vee$ add and remove split points. We can see them as defined by the operator '$W(\alpha, \epsilon)$', replacing a split point in α by the empty string ϵ.

The $^\wedge$ connective is used in the analysis of extraction and similar cases where we need to remove the split point left by hypothetical material. **?** assign a relative pronoun such as "that" the formula $(n \backslash n)/^\wedge(s \uparrow np)$.

$$\dfrac{\dfrac{\begin{array}{c} a : np \\ \vdots \\ \text{John} + \text{read} + a + \text{yesterday} : s \end{array}}{\text{John} + \text{read} + \mathbf{1} + \text{yesterday} : s \uparrow np} \uparrow I}{\text{John} + \text{read} + \text{yesterday} : {}^\wedge(s \uparrow np)} {}^\wedge I$$

This provides a solution for the medial extraction cases like sentence (14) from Section 1.5.

The $^\vee$ connective introduces a type of non-determinism of the position of the split point: the $^\vee E$ rule removes a split point, but the $^\vee I$ rule can then

$$\dfrac{\delta : A \odot B \quad \begin{array}{c}[\alpha_1 + \mathbf{1} + \alpha_2 : A]^i \quad [\beta : B]^i \\ \vdots \\ \gamma_1 + \alpha_1 + \beta + \alpha_2 + \gamma_2 : C\end{array}}{\gamma_1 + \delta + \gamma_2 : C}\ \odot E_i \qquad \dfrac{\alpha_1 + \mathbf{1} + \alpha_2 : A \quad \beta : B}{\alpha_1 + \beta + \alpha_2 : A \odot B}\ \odot I$$

$$\dfrac{\alpha_1 + \mathbf{1} + \alpha_2 : C \uparrow B \quad \beta : B}{\alpha_1 + \beta + \alpha_2 : C}\ \uparrow E \qquad \dfrac{\begin{array}{c}[\beta : B]^i \\ \vdots \\ \alpha_1 + \beta + \alpha_2 : C\end{array}}{\alpha_1 + \mathbf{1} + \alpha_2 : C \uparrow B}\ \uparrow I_i$$

$$\dfrac{\alpha_1 + \mathbf{1} + \alpha_2 : A \quad \beta : A \downarrow C}{\alpha_1 + \beta + \alpha_2 : C}\ \downarrow E \qquad \dfrac{\begin{array}{c}[\alpha_1 + \mathbf{1} + \alpha_2 : A]^i \\ \vdots \\ \alpha_1 + \beta + \alpha_2 : C\end{array}}{\beta : A \downarrow C}\ \downarrow I_i$$

$$\dfrac{\delta : {}^{\wedge}A \quad \begin{array}{c}[\alpha_1 + \mathbf{1} + \alpha_2 : A]^i \\ \vdots \\ \gamma_1 + \alpha_1 + \alpha_2 + \gamma_2 : C\end{array}}{\gamma_1 + \delta + \gamma_2 : C}\ {}^{\wedge}E_i \qquad \dfrac{\alpha_1 + \mathbf{1} + \alpha_2 : A}{\alpha_1 + \alpha_2 : {}^{\wedge}A}\ {}^{\wedge}I$$

$$\dfrac{\alpha_1 + \mathbf{1} + \alpha_2 : {}^{\vee}A}{\alpha_1 + \alpha_2 : A}\ {}^{\vee}E \qquad \dfrac{\alpha_1 + \alpha_2 : A}{\alpha_1 + \mathbf{1} + \alpha_2 : {}^{\vee}A}\ {}^{\vee}I$$

Table 2.7: Labeled natural deduction rules for the discontinuous connectives of the Displacement calculus

insert a split point at any place (for the deterministic version, only the position between two other split points is determined). **?** use this expressivity for the treatment of words allowing relatively free placement, such as parentheticals.

For the mapping from the syntactic source calculus **D** to the semantic target calculus **LP**, the unary type-forming operations are considered inert: the inference rules for these connectives, consequently, leave no trace in the **LP** proof term associated with a derivation in the syntactic source calculus. The continuous and discontinuous families, for the rest, are treated exactly alike. Specifically, the infixation and extraction operations are mapped to **LP** function types, like the slashes.

2.2.1 Illustration

D has been successfully applied to a great number of discontinuous dependencies. The operations of sort 1 (with a single split point) are used in the analysis of non-peripheral extraction, discontinuous idioms, gapping and ellipsis, quantifier scope construal, reflexivisation, pied-piping and the Dutch cross-serial dependencies, among others.

As an example, consider quantifier scope construal. **D** provides a uniform type assignment to generalized quantifier expressions such as "everyone", "someone": $(s \uparrow np) \downarrow s$. In the syntactic source calculus, this type assignment allows a quantifier phrase QP to occupy any position that could be occupied by a regular non-quantificational noun phrase. Semantically, the image of the $\uparrow$ Introduction rule at the level of the semantic target calculus **LP** binds an np type hypothesis at the position that was occupied by the quantifier phrase (the anp premise, with a a structural variable for the np hypothesis). The image of the $\downarrow$ Elimination rule applies the term representing the QP meaning to this abstract. Scope ambiguities arise from derivational ambiguity in the source calculus **D**. The derivation below results in a non-local reading 'there is a particular x such that Mary thinks x left'. Looking upward from the conclusion, the last rule applied is $\downarrow$ Elimination, which means the quantifier phrase takes scope at the main clause level. An alternative derivation, producing the local scope reading, would have the / Elimination rule for "thinks": $(np\backslash s)/s$ as the last step.

$$\dfrac{\text{someone} : (s \uparrow np) \downarrow s \quad \dfrac{\begin{array}{c} a : np \\ \vdots \\ \text{Mary} + \text{thinks} + a + \text{left} : s \end{array}}{\text{Mary} + \text{thinks} + \mathbf{1} + \text{left} : s \uparrow np}\uparrow I}{\text{Mary} + \text{thinks} + \text{someone} + \text{left} : s}\downarrow E$$

2.3 Lambda grammars

The origins of lambda grammars can be traced back to the work of **?** and **?**, with Dick Oehrle the first to present a simple treatment of quantifier scope in the system. The work of **?** and **?** caused a renewed interest in the system. Lambda grammars are also called abstract categorial grammars (technically a lambda grammar corresponds to a *pair* of abstract categorial grammars when definitions are harmonized (**?**)) or linear grammars (**?**). Their defining feature is the use of the lambda calculus not only for the semantic terms representing the meaning of phrases but also for *prosodic* terms representing the string.

In other words, the syntactic logic for lambda grammars is simply **LP**, the homomorphism from syntactic to semantic types remains as it was, but there is now a second homomorphism from syntactic to prosodic types, translating all atomic types to type s for string.

semantic types: $(np)' = e$, $(s)' = t$, $(n)' = e \to t$, $(A \multimap B)' = A' \to B'$

prosodic types: $(np)' = s$, $(s)' = s$, $(n)' = s$, $(A \multimap B)' = A' \to B'$

Given a type of $np \multimap s$ for an intransitive verb like "left", this means the corresponding prosodic term is $s \to s$, a function from strings to strings. As a term of this type, we can use $\lambda y.(y + \text{left})$, where '+' represents the concatenation operation as before. The table below gives some more lexical entries.

Word	syntactic type	prosodic type	prosodic term
Mary	np	s	Mary
thinks	$s \multimap np \multimap s$	$s \rightarrow s \rightarrow s$	$\lambda y \lambda x.(x + \text{thinks} + y)$
someone	$(np \multimap s) \multimap s$	$(s \rightarrow s) \rightarrow s$	$\lambda P.(P\,\text{someone})$
left	$np \multimap s$	$s \rightarrow s$	$\lambda z.(z + \text{left})$

The lexical entries for "left" and "thinks" correspond to the Lambek calculus formulas $np\backslash s$ and $(np\backslash s)/s$ respectively, with the word order information moved from the logical formula to the prosodic term. The interesting case is the assignment to the quantifier: syntactically it takes a sentence missing a noun phrase to produce a sentence, while prosodically, it transforms a string with a hole (corresponding syntactically to the missing noun phrase) into a string where this hole is filled by the constant 'someone'.

The rules for term assignment to syntactic proofs are the same as those we have seen for semantic term assignment.

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B} \multimap I \qquad \frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash (M\ N) : B} \multimap E$$

One of the possible syntactic proofs given these lexical entries is the following.

$$\dfrac{w_3 : (np \multimap s) \multimap s \qquad \dfrac{\dfrac{w_1 : np \qquad \dfrac{w_2 : s \multimap np \multimap s \qquad \dfrac{v : np \quad w_4 : np \multimap s}{(w_4\ v) : s} \multimap E}{(w_2\ (w_4\ v)) : np \multimap s} \multimap E}{((w_2\ (w_4\ v))\ w_1) : s} \multimap E}{\lambda v.((w_2\ (w_4\ v))\ w_1) : np \multimap s} \multimap I}{(w_3\ \lambda v.((w_2\ (w_4\ v))\ w_1)) : s} \multimap E$$

After substitution of the prosodic terms for the different words in the lexicon, the computed term $w_3\ \lambda v.((w_2\ (w_4\ v))\ w_1)$ for this proof becomes

$$(\lambda P.(P\,\text{someone}))\ \lambda v.(((\lambda y \lambda x.x + \text{thinks} + y)\ ((\lambda z.z + \text{left})\ v))\ \text{Mary})$$

which reduces to 'Mary + thinks + someone + left' by β reduction.

Discussion As a consequence of its logical setup, lambda grammars make the strong claim that every **LP** proof corresponds to a sequence of words. This in contrast to other type-logical grammars where the set of proofs is generally a proper subset of the **LP** proofs. Where in other typelogical grammars word order is determined by the logic itself, in lambda grammars the prosodic lambda terms in the lexicon determine the word order.

In lambda grammars, as in the lambda calculus, the syntactic constructors are application (for the elimination rule) and abstraction (for the introduction rule). However, the introduction rule is universal: once there is an occurrence of x we can abstract over this occurrence without regard for whether it is the leftmost or rightmost undischarged hypothesis as we would in a Lambek calculus proof.

$$\frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash (M\ N) : B} \multimap E \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B} \multimap I$$

$$\frac{\Gamma \vdash P : B \quad \Delta \vdash Q : B\backslash A}{\Gamma, \Delta \vdash P + Q : A} \backslash E \qquad \frac{w : B, \Gamma \vdash w + P : A}{\Gamma \vdash P : B\backslash A} \backslash I$$

$$\frac{\Gamma \vdash P : A/B \quad \Delta \vdash Q : B}{\Gamma, \Delta \vdash P + Q : A} /E \qquad \frac{\Gamma, w : B \vdash P + w : A}{\Gamma \vdash P : A/B} /I$$

HTLG : natural deduction

Table 2.8: Natural deduction rules for hybrid type-logical grammars

2.4 Hybrid type-logical grammars

Lambda grammars and Lambek grammars have opposite 'periphery problems', with Lambek grammars unable to handle *non-peripheral* extraction and lambda grammars unable to handle *peripheral* extraction. In a series of papers, Kubota and Levine (**?**, **?**) present a direct *combination* of lambda grammars with the Lambek calculus, which they call hybrid type-logical grammars (HTLG).

To ensure the soundness of a combination of lambda grammars and the Lambek calculus into a single system, there needs to be a sensible way of combining function application and abstraction with string concatenation and its residuals. For example, when f and g are both functions from strings to strings, it is not clear what their concatenation would be, since concatenation is an operation on strings and not on functions.

The solution of Kubota and Levine is to require that Lambek calculus formulas only operate on objects of the string type. In other words, the lambda grammar linear implication '$\multimap$' can never occur within the scope of a Lambek calculus implication. As a consequence, $(np \multimap (np\backslash s)) \multimap (np\backslash s)$ and $((np\backslash s)/np) \multimap s$ are valid HTLG formulas, but $(n\backslash n)/(np \multimap s)$ and $((np \multimap s) \multimap s)/n$ are not.

The natural deduction rules for HTLG are as shown in Table 2.8.

All variables and terms in the Lambek calculus rules are of type s (string) and '+' is string concatenation. The Lambek calculus elimination rules take terms P and Q of the two premisses of the rule, both representing strings, and assigns the concatenation of these two string to the rule conclusion. Similarly, the $\backslash I$ rule verifies that the string variable w (corresponding to hypothesis B in the premiss of the rule) is the leftmost premiss of the computed string, whereas the $/I$ rule verifies it is the rightmost premiss. The $\multimap E$ and $\multimap I$ rule are unchanged from their lambda grammar counterparts.

Given these logical rules and the standard lambda grammar assignment of $\lambda P.(P\, e) : (np \multimap s) \multimap s$ to a quantifier like "everyone", we derive the

Lambek calculus subject quantifier $s/(np\backslash s)$ as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : np \quad y : np\backslash s}{x + y : s}\,\backslash E}{\lambda x.(x + y) : np \multimap s}\multimap I \qquad \lambda P.(P\,e) : (np \multimap s) \multimap s}{(\lambda P.(P\,e))(\lambda x.(x + y)) : s}\multimap E}{e + y : s}\equiv_\beta}{e : s/(np\backslash s)}\,/I$$

This derivation depends crucially both on working modulo β equivalence and on substitution of the lexical lambda term $\lambda P.(P\,e)$ into the proof, as indicated by $\equiv_\beta$ step in the proof, and the complex term assigned to the hypothesis $(np \multimap s) \multimap s$ respectively. Together, these allow us to compute the string term $e+y$ indicating that the hypothesis y corresponding to the formula $np\backslash s$ is the rightmost hypothesis of the proof, thereby allowing us to perform the $/I$ rule to withdraw this y hypothesis.

Although it may seem unusual to allow the success of a derivation to depend on the lexical term, the lambda term recipe given by the lexicon is crucial for determining the left- and rightmost positions of a string term. For example, choosing $\lambda P.(P\ \epsilon) + e$ (where ϵ denotes the empty string) would produce $y + e$ after the β reduction step, which would make the final $/I$ step in the proof invalid (but which would allow a $\backslash I$ inference as the last step). In other words, although hybrid type-logical grammars have a less clean separation of the lexicon from the logic than other type-logical grammars, this is a consequence of ensuring the Lambek calculus implications behave correctly.

2.5 The logic of scope

The logic of scope $\mathbf{NL}_\lambda$ was introduced by Chris Barker and Chung-Chieh Shan (**?**) as a logic based on the non-associative Lambek calculus **NL**. It is a basic logic with two non-associative modes. In a standard multimodal setup this would produce sequents where the antecedents are binary branching trees, with formulas at the leaves and the internal nodes labeled with one of the two modes. The particularity of $\mathbf{NL}_\lambda$ is that it introduces what it calls *gapped structures* where the gaps are indicated in the antecedents by a notation based on lambda abstraction. The key component of $\mathbf{NL}_\lambda$ and its main departure from other type-logical grammars are the structural rules below.

$$\dfrac{\Xi[\Gamma[\Delta]] \Rightarrow C}{\Xi[\Delta \circ_w \lambda x.\Gamma[x]] \Rightarrow C}\,\lambda \qquad \dfrac{\Xi[\Delta \circ_w \lambda x.\Gamma[x]] \Rightarrow C}{\Xi[\Gamma[\Delta]] \Rightarrow C}\,\lambda^{-1}$$

The λ rule has the side condition that the variable x must be unique to the proof. The λ rule can move out *any* substructure Δ from a structure Γ (in any larger context Ξ), producing a structure with the wrapping mode '$\circ_w$' where Δ is the left sister of the structure Γ where the variable x, which is abstracted by a λ operator, marks the original position of Δ. The side condition of the λ

rule demands that each λ rule uses a distinct variable. The inverse rule λ^{-1} moves Δ back to its original position.

Although the notation is borrowed from the lambda calculus and some of the properties of the lambda constructor are reminiscent of the linear lambda calculus, an $\mathbf{NL}_\lambda$ antecedent with a number of lambda operators is in many ways closer to an antecedent with the same number of split points in the discontinuous Lambek calculus, with the important difference that $\mathbf{NL}_\lambda$ operates in a non-associative calculus and that in $\mathbf{NL}_\lambda$ insertion occurs on the outermost abstracted variable whereas in the discontinuous Lambek calculus insertion occurs with respect to the relative linear order of the split points.

As an example of how $\mathbf{NL}_\lambda$ works in practice, we return to the example sentence "Mary thinks someone left" with the wide scope reading for "someone". Given the standard formula assignments of np to "Mary", $(np \backslash s) / s$ to "thinks" and $np \backslash s$ to "left", one of the proofs is the following.

$$\dfrac{\dfrac{\text{someone} \vdash s /_w (np \backslash_w s) \qquad \dfrac{\dfrac{\dfrac{\vdots}{\text{Mary} \circ (\text{thinks} \circ (np \circ \text{left})) \vdash s}}{np \circ_w \lambda x.(\text{Mary} \circ (\text{thinks} \circ (x \circ \text{left}))) \vdash s}\,\lambda}{\lambda x.(\text{Mary} \circ (\text{thinks} \circ (x \circ \text{left}) \vdash np \backslash_w s}\,\backslash_w I}{\text{someone} \circ_w \lambda x.(\text{Mary} \circ (\text{thinks} \circ (x \circ \text{left}))) \vdash s}\,/_w E}{\text{Mary} \circ (\text{thinks} \circ (\text{someone} \circ \text{left})) \vdash s}\,\lambda^{-1}$$

For the interesting part of the proof, reading from the conclusion to the premisses, we use the λ^{-1} rule to move "someone" to the leftmost position and mark its previous position by the abstracted variable x. The introduction rule and the λ rule then move the np back to the original position of "someone". We can then derive "Mary thinks np left" of type s as before.

The proof is structurally similar to the $\mathbf{D}$ proof of the same reading, with the combination of λ and $\backslash_w I$ corresponding to the $\uparrow I$ rule and the combination of λ^{-1} and $/_w E$ corresponding to the $\downarrow E$ rule.

2.5.1 Discussion

Although the λ structural rules are somewhat unusual, $\mathbf{NL}_\lambda$ gives us a fairly simple treatment of quantifier scope and many other phenomena. **?** present another calculus, $\mathbf{NL}_{cl}$, which has a standard set of multimodal structural rules. They present a translation from $\mathbf{NL}_\lambda$ proofs to $\mathbf{NL}_{cl}$ proofs and vice versa. In an indirect way, this also provides standard Kripke models for $\mathbf{NL}_\lambda$ via its translations to $\mathbf{NL}_{cl}$, although **?** also provides a way to obtain Kripke models directly.

While nothing in the setup of $\mathbf{NL}_\lambda$ itself hinges upon non-associativity of '$\circ$', the translation from $\mathbf{NL}_\lambda$ to $\mathbf{NL}_{cl}$ *does* crucially depend on the non-associativity of the target logic. Therefore, adding associativity of '$\circ$' (or even a separate associative mode in addition to the two other modes) to $\mathbf{NL}_\lambda$ would require a more delicate translation to a standard multimodal logic.

2.6 Lambek-Grishin

In addition to the logical perspective of the Lambek calculus, Lambek (?, ?) also presented an algebraic perspective where the three connectives of the Lambek calculus form a residuate triple as indicated by Equation 2.5 below. Grishin extended the Lambek calculus by adding dual residuation principles to the Lambek calculus' residuation principle. So where the Lambek calculus has three connectives satisfying Equation 2.5, the Lambek-Grishin calculus has three additional, dual residuated connectives satisfying Equation 2.6.

$$B \to A\backslash C \quad \Leftrightarrow \quad A \bullet B \to C \quad \Leftrightarrow \quad A \to C/B \tag{2.5}$$

$$A \oslash\!\!\!\!\!\!\;\backslash\, C \to B \quad \Leftrightarrow \quad C \to A \circledast B \quad \Leftrightarrow \quad C \oslash B \to A \tag{2.6}$$

Just like the algebraic/combinatorial representation of the non-associative Lambek calculus, we need only add reflexivity and transitivity of '→' to obtain the 'basic' Lambek-Grishin calculus.

The three Grishin connectives '⦸', '⊛' and '⊘' are related to each other in a way which is perfectly symmetric to the relation between '\', '•', and '/'. Grishin also investigated the addition of four sets of interaction principles to the calculus. One of these sets — shown below as Equations 2.7 and 2.8 — is the most interesting for our purposes: they combine the Lambek and Grishin connectives in a way which (at least with perfect hindsight) resembles mixed associativity and commutativity in multimodal grammars.

$$(A \obslash B) \bullet C \to A \obslash (B \bullet C) \qquad A \bullet (B \oslash C) \to (A \bullet B) \oslash C \tag{2.7}$$

$$B \bullet (A \obslash C) \to A \obslash (B \bullet C) \qquad (A \oslash C) \bullet B \to (A \bullet B) \oslash C \tag{2.8}$$

The postulates for ⦸ allow us to move the A argument of the connective out of Lambek structures, essentially allowing it to select its argument at a distance, provided it passes only through Lambek contexts '•'. Linguistic applications and formal properties of **LG** have been investigated by Bernardi, Moortgat and colleagues (?, ?, ?), but the linguistic applications have not been as well-developed compared to many other type-logical grammars.

2.7 Discussion

Table 2.9 gives an executive summary of the different logics which serve as the basis of modern type-logical grammars. For each logic, we present its logical connectives, the way its logical statements are structured, and the operations on these structures.

For multimodal type-logical grammars, the formulas are structured as labeled 1-2 trees (that is, trees with both unary and binary branches) and we allow tree rewrites which can refer to these labels, but cannot copy or delete subtrees. With the exception of lambda grammars, which only have linear implication as a logical connective, all type-logical grammars take either **NL** or **L** as their base logic, to which they add different connectives. On the structure/operation side, a number of formalisms allow some form of (linear)

Logic	Connectives	Structure	Operations
L	$/, \bullet, \backslash$	list	—
NL	$/, \bullet, \backslash$	binary tree	—
Multimodal	$/_i, \bullet_i, \backslash_i$	labeled binary tree	tree rewrites
	$\Diamond_j, \Box_j$	labeled 1-2 tree	tree rewrites
D	$\mathbf{L} + \uparrow_k, \odot_k, \downarrow_k$		
	$^\wedge, {}^\vee$	tuple of lists	wrap
Lambda	$\multimap$	lambda term	β reduction
Hybrid	$\mathbf{L} + \multimap$	lambda term (list)	β reduction
$\mathbf{NL}_\lambda$	$\mathbf{NL} + /\!\!/, \circledcirc, \backslash\!\!\backslash$	lambda term (tree)	β reduction/expansion
LG	$\mathbf{NL} + \oslash, \circledast, \obslash$	free tree	graph rewrites

Table 2.9: Executive summary of different type-logical grammars

Formalism	RNR	Ex	q	$\wedge$	gap	Language	Complexity
NL	$-$	$-$	$-$	$+$	$-$	$=$ CFL	P
L	$+$	$-$	$-$	$+$	$-$	$=$ CFL	NP-complete
D	$+$	$+$	$+$	$+$	$+$	$\supseteq \text{MCFL}_{wn}$	NP-complete
Lambda	$-$	$+$	$(+)$	$-$	$-$	$\supseteq$ MCFL	NP-complete
LG	$-$	$-$	$+$	$+$	$-$	$\supseteq$ MCFL	NP-complete
$\mathbf{NL}_\lambda$	$-$	$+$	$+$	$+$	$+$	$\supseteq$ MCFL	NP
Hybrid	$+$	$+$	$+$	$+$	$+$	$\supseteq$ MCFL	NP-complete
MILL1	$+$	$+$	$+$	$+$	$+$	$\supseteq$ MCFL	NP-complete
Multimodal	$+$	$+$	$+$	$+$	$+$	$=$ CSL	PSPACE-complete

Table 2.10: Scorecard for the different type-logical grammars

lambda abstraction and β reduction, but these have a different status in different systems: in $\mathbf{NL}_\lambda$, only structural rules can produce or remove lambda terms, whereas in lambda grammars and hybrid grammars, lambda terms appear both in the lexicon and (by the Curry-Howard isomorphism) as the result of the rules for linear implication '$\multimap$'.

Compared to the Lambek calculus, which is a logic of strings or of lists of formulas, the Displacement calculus **D** can be seen as a logic of *tuples* of strings or lists of lists of formulas. An expressions of sort k, that is with k separator symbols, corresponds to a $k+1$-tuple, and the wrap operation $W_k(\alpha, \beta)$ replaces the kth separator symbol in α by β.

The Lambek-Grishin calculus **LG** adds up-down symmetric connectives to the non-associative Lambek calculus, which produces acyclic connected graphs (free trees, to be contrasted with the rooted trees of other type-logical grammars) with two types of binary branches, branching either upward or downward. The interaction rules proposed by **?**, which reconnect binary branches of the two families, extend the expressivity of the system.

Table 2.10 presents a 'scorecard' for the different type-logical grammars discussed here. The first 5 columns represent classes of linguistic phenomena, corresponding essentially to our problem cases in Section 1.5.2:

1. *right-node raising* (RNR); this includes related forms of non-constituent coordination,
2. *extraction* (Ex), including medial extraction,
3. *quantifier scope* (q), including related phenomena handled by **?** q operator; the '(+)' indication for lambda grammars indicates that in this formalism q works for quantifiers, but not for related constructions such as reflexives,
4. *coordination* ($\wedge$) represents the treatment of coordination,
5. *gapping* (gap) represents the treatment of gapping.

The final two columns indicate the known results for formal language classes[6] (for most formalisms, only a lower bound is known) and complexity.

2.8 Conclusion

We have seen quite a number of logical systems in this chapter, all adhering to the logical view of grammars introduced by **?**. Even though there are many different logical primitives used, there is a large agreement among formalisms as to the underling deep structure of the analysis of many phenomena[7]. In addition, there seems to be a sort of family resemblance between the treatments of medial extraction and quantifier scope in these different systems:. However, the differences at the level of surface structure make it difficult to give detailed comparisons between analyses in different formalisms. Surface structure comparisons would make it easier to translate analyses between different formalisms, but also to show where analyses differ. The main goal of Chapters 4 and 5 is to make such comparisons possible by providing two general logical formalisms which together capture all modern type-logical grammars. But before that, the next chapter presents some of the necessary background in proof theory.

[6]CFL denotes the context-free languages, MCFL$_{wn}$ and MCFL the (well-nested) multiple context free languages (**?**), and CSL the context-sensitive languages.

[7]The Lambek-Grishin calculus is an exception here in that we can not directly 'read off' the deep structure derivation from the surface structure one. Its syntax-semantics interface is more complex, but also more flexible (**?**, **?**).

3 Proof Nets

3.1 Proof systems

The Lambek calculus, like many other logics, can be formulated in a number of different ways. Although these different formulations are easily shown to be equivalent, each proof system has its own set of advantages and disadvantages. The executive summary of these advantages is as follows.

Natural deduction has the advantage of a direct link to formal semantics by means of the Curry-Howard isomorphism. Even though this is clear and simple for the implications, this straightforward link to semantics is complicated by the $\bullet E$ rule[1],

Sequent calculus has the advantage of clearly showing the logical symmetry of the calculus. It is also easy (although inefficient) to use for proof search. Sequent calculus is also the system of choice for inductive proofs about the Lambek calculus: some results, such as the subformula property, are most easily shown for the sequent calculus. Unlike natural deduction, different proofs do not necessarily correspond to different semantic readings (this is generally called the problem of *spurious ambiguity*: the sequent calculus generates many different proofs of the same reading). Intuitively, this is because the sequent calculus is overly bureaucratic.

[1]In intuitionistic logic, adding the conjunction rules don't represent such a complication, but this is due to the fact that $\wedge E$ is additive from the linear logic point of view, whereas $\bullet E$ is multiplicative.

Combinator systems have the advantage of a very simple rule set, which makes them the system of choice for proving soundness and completeness with respect to models. For the Lambek calculus (and its multimodal versions) the combinatorial representation also forms an interpretation of the logic in category theory. Unlike natural deduction and sequent calculus, combinator systems generally do not have the subformula property, and doing parsing/proof search directly in combinator systems is quite hard,

Proof nets combine the good properties of natural deduction and the sequent calculus: they have the logical symmetry of the sequent calculus and the correspondence between proofs and lambda terms of natural deduction. Because of this combination, they are a convenient formalism for proof search. On the other hand, inductive proofs requiring us to decompose a proof net into its valid substructures tend to be harder than for sequent proofs: unlike for sequent proofs, where it is easy to remove the last rule, proof nets require us to be a bit more careful.

Because of the constructive intertranslatability results, these advantages are all relative.

3.1.1 Natural deduction

Natural deduction is the proof system we have seen in Chapter 1. Natural deduction can be presented in two different ways, which are easily shown to be equivalent. The easiest way to present natural deduction is in the tree like form shown in Table 3.1. This representation is sometimes called Prawitz style or Prawitz-Gentzen style.

This representation has the advantage that we do not have to manage the hypotheses explicitly: the leaves of the tree — or at least those leaves which have not been discharged — are the hypotheses. Given that the Lambek calculus is a non-commutative logic, the yield of the tree represents the list of the hypotheses. This allows us to specify that the $\backslash I$ rule discharges the leftmost (undischarged) hypothesis of the proof, and the $/I$ the rightmost one. The $/I$ and $\backslash I$ rules have the additional condition that there must be at least one other undischarged hypothesis besides B (the prevents empty antecedent derivations such as $\vdash n/n$). The $\bullet E$ rule is a bit complicated in this respect: we require that the two discharged A and B hypotheses are adjacent, but the hypotheses used to derive $A \bullet B$ appear between those to the left of A and those to the right of B. In other words, the discharged hypotheses A and B are replaced by the hypotheses used to derive $A \bullet B$ with respect to the yield.

Table 3.2 shows the version of natural deduction with explicit hypotheses. While the $\bullet E$ rule is still complicated in this format, the explicit management of hypotheses allows the rule to state directly (and without tortuous prose) that Δ occurs between Γ and Γ' in the conclusion of the rule. The $/I$ and $\backslash I$ rules have the condition that Γ cannot be empty, but we can state explic-

$$\dfrac{A \bullet B \qquad \begin{array}{c}\dots [A]^i [B]^i \dots \\ \vdots \\ C\end{array}}{C}\ \bullet E_i \qquad\qquad \dfrac{A \quad B}{A \bullet B}\ \bullet I$$

$$\dfrac{A/B \quad B}{A}\ /E \qquad\qquad \dfrac{\begin{array}{c}\dots [B]^i \\ \vdots \\ A\end{array}}{A/B}\ /I_i$$

$$\dfrac{B \quad B\backslash A}{A}\ \backslash E \qquad\qquad \dfrac{\begin{array}{c}[B]^i \dots \\ \vdots \\ A\end{array}}{B\backslash A}\ \backslash I_i$$

Table 3.1: Natural deduction for **L**, Prawitz style. The A and B hypotheses must be adjacent for the $\bullet E$ rule, B must be the leftmost undischarged hypothesis for $/I$ and the rightmost for $\backslash I$.

$$\dfrac{\Delta \vdash A \bullet B \quad \Gamma, A, B, \Gamma' \vdash C}{\Gamma, \Delta, \Gamma' \vdash C}\ \bullet E \qquad \dfrac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \bullet B}\ \bullet I$$

$$\dfrac{\Gamma \vdash A/B \quad \Delta \vdash B}{\Gamma, \Delta \vdash A}\ /E \qquad \dfrac{\Gamma, B \vdash A}{\Gamma \vdash A/B}\ /I$$

$$\dfrac{\Gamma \vdash B \quad \Delta \vdash B\backslash A}{\Gamma, \Delta \vdash A}\ \backslash E \qquad \dfrac{B, \Gamma \vdash A}{\Gamma \vdash B\backslash A}\ \backslash I$$

Table 3.2: Natural deduction for **L**, sequent style

itly that B must be the leftmost (respectively the rightmost) undischarged hypothesis.

The sequent style version of natural deduction results in proofs which are a bit more complicated because of the explicit hypothesis management, but this also gives us added flexibility. Many rules can not easily be stated directly on trees (at least not without bending the definition of tree beyond recognition) but can be stated without problem in sequent style natural deduction. These include the linear logic additives, but also the rules for the non-associative Lambek calculus and the rules for the unary residuated connectives $\Diamond$ and $\Box$.

In natural deduction, the connectives have an elimination rule E, which tells us how to *use* a formula with this connective, and an introduction rule I, which tells us how to *prove* a formula with this connective. As a consequence, for the elimination rules, the formula with the principal connective appears as a premiss of the rule (and disappears in its conclusion), whereas for the introduction rule, for formula with the principal connective appears only in the conclusion.

For natural deduction calculi, we generally want to prove a form of normalisation. Normalisation is a way of showing that the introduction and elimination rules are duals or inverses and that their combinations 'cancel out'. For example, a normalisation step for / looks as follows.

$$\dfrac{\dfrac{\begin{array}{c}\Gamma\quad [B]^i\\ \vdots\ \delta_1\\ A\end{array}}{A/B}\,/I_i \qquad \begin{array}{c}\Delta\\ \vdots\ \delta_2\\ B\end{array}}{A}\,/E \qquad \rightsquigarrow \qquad \begin{array}{c}\Delta\\ \vdots\ \delta_2\\ \Gamma\quad B\\ \vdots\ \delta_1\\ A\end{array}$$

On the left hand side, we have an introduction rule producing a formula A/B, which is then immediately combined with B for form an A again. This proof contains a 'detour' of a kind: if we can prove B using the proof δ_2, then instead of the withdrawn hypothesis B, we can use this proof δ_2 directly, then continue the proof δ_1 as before to produce A. This produces the proof shown above on the right. This operation preserves the order on the undischarged hypotheses Γ, Δ. On the term side, this corresponds to replacing a term $((\lambda x.M)N)$ by $M[x := N]$, that is to a beta reduction on the terms.

3.1.2 Sequent calculus

Table 3.3 shows the sequent calculus version of the Lambek calculus. In the sequent calculus the $L\bullet$ rule is just a normal rule like the others — unlike the $\bullet E$ rule in natural deduction to which it corresponds.

Compared to natural deduction, the sequent calculus is considerably more bureaucratic. While the natural deduction calculus (like the sequent calculus) keeps track of the hypotheses of a proof, in a natural deduction proof nearly all the 'action' happens on the right hand side of the turnstile. The left rules of the sequent calculus, on the other hand, operate (as their name suggests) entirely on the left hand side. For example for the $L/$ rule, there is non-determinism in the choice of the main formula of the rule (there can be alternative left or right rules which can apply depending on Γ, Δ, Γ' and C) and additional nondeterminism in splitting up the context to the right of the formula A/B into Δ and Γ' (the same non-determinism applies, up to symmetry, to the $L\backslash$ rules). As a consequence one natural deduction proof generally corresponds to many different sequent calculus proofs because of inessential rule permutations — a problem sometimes called the *spurious ambiguity* problem in the type-logical grammar literature. As discussed in Section 1.3.1, we are interested in different proofs only when they produce (at least potentially) different meanings. More precisely, we are interested in different proofs only when the lambda terms corresponding to these proofs (by the Curry-Howard isomorphism) are different. In other words, we want our proof search to enumerate (normal form) natural deduction proofs, or at least proof objects in 1-1 correspondence with them.

Identity

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

Logical Rules

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

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

$$\frac{\Delta \vdash B \quad \Gamma, A, \Gamma' \vdash C}{\Gamma, \Delta, B\backslash A, \Gamma' \vdash C}\ L\backslash \qquad \frac{B, \Gamma \vdash A}{\Gamma \vdash B\backslash A}\ R\backslash$$

Table 3.3: The sequent calculus **L**

The sequent calculus has left L and right R rules for the logical connectives (instead of the introduction and elimination rules of natural deduction) depending on whether the main formula of the rule occurs on the left or on the right of the turnstile. Whereas natural deduction is based on a symmetry of between introduction and elimination rules for a connective, with normalisation the standard verification the system is logically well-behaved, sequent calculus is based on a symmetry between left and right rules, and cut elimination is the standard verification that the system is well-behaved.

Proving cut elimination is a bit complicated because of the syntactic setup of the sequent calculus. Many (indeed most) of the cases in a cut elimination proof consist of changing the order of two rules until we arrive at the case where there is a cut rule of a formula A which is the main formula of both rules producing the premisses of the cut rule. The proof below shows the case for a cut formula A/B which is the main formula of both rules providing the premisses of the cut rule.

$$\dfrac{\dfrac{\overset{\vdots\ \delta_1}{\Delta, B \vdash A}}{\Delta \vdash A/B}\ R/ \qquad \dfrac{\overset{\vdots\ \delta_2}{\Delta' \vdash B} \quad \overset{\vdots\ \delta_3}{\Gamma, A, \Gamma' \vdash C}}{\Gamma, A/B, \Delta', \Gamma' \vdash C}\ L/}{\Gamma, \Delta, \Delta', \Gamma'}\ Cut$$

We can replace this cut by two cuts on the immediate subformulas of A/B as follows.

$$\dfrac{\overset{\vdots\ \delta_2}{\Delta' \vdash B} \qquad \dfrac{\overset{\vdots\ \delta_1}{\Delta, B \vdash A} \quad \overset{\vdots\ \delta_3}{\Gamma, A, \Gamma' \vdash C}}{\Gamma, \Delta, B, \Gamma' \vdash C}\ Cut}{\Gamma, \Delta, \Delta', \Gamma'}\ Cut$$

? already proved cut elimination in his original article on the Lambek calculus including the many cases of rule permutations (**?**, Section 2.7, also provide a full proof). When a new type-logical formalism is introduced, normal a cut elimination proof is one of the first things to prove. For example, **?** prove cut elimination proof for the hybrid type-logical grammars of **?**.

Cut elimination has important consequences. Firstly, it ensures the subformula property. That is, to prove a statement we only need to consider formulas which are subformulas of the initial statement. Although the subformula property also holds for natural deduction proofs, it is a bit harder to prove: the C formula of the $\bullet E$ rule complicates the proof.

A second advantage of cut elimination, is that we can use the (cut-free) sequent calculus for backward chaining proof search. In the absence of the cut rule, there is only a finite number of ways to instantiate the rules, and each of them only produces smaller subproofs (since, at the very least, the premisses have one less connective). **?** used cut elimination as a way to prove decidability of his logic. For variants and extensions of the Lambek calculus, establishing decidability can be a bit more involved, but still generally passes by cut elimination.

However, while cut elimination shows it is certainly possible to use the sequent calculus for proof search, it is not a very efficient: the same rule permutations which complicate the cut elimination proof also complicate proof search, and while it is possible use intelligent proof search strategies for the sequent calculus of the Lambek calculus (**?**), I believe it is worthwhile to look for proof systems which inherently avoid the spurious ambiguity problem. Ideally, each distinct proof object would correspond to at least a potentially distinct reading of a sentence.

3.1.3 Combinatorial calculus

Lambek's classic paper (**?**) starts with a combinatorial representation of the Lambek calculus based on the algebraic principle of residuation. Table 3.4 shows the rules of this calculus.

In the Lambek calculus, the standard interpretation of the product '$\bullet$' is as a type of concatenation, with the implications '$\backslash$' and '$/$' its residuals. Using the residuation calculus, we can derive standard cancellation schemes such as the following.

$$\dfrac{\dfrac{}{C / B \to C / B}\,\mathit{Refl}}{(C / B) \bullet B \to C}\,\mathit{Res}_{/,\bullet} \qquad\qquad \dfrac{\dfrac{}{A \backslash C \to A \backslash C}\,\mathit{Refl}}{A \bullet (A \backslash C) \to C}\,\mathit{Res}_{\backslash,\bullet} \qquad\qquad \mathit{units}$$

Showing us that when we compose C / B with a B to its right, we produce a C, and that when we compose $A \backslash C$ with an A to its left, we produce a C.

The residuation presentation of the Lambek calculus naturally forms a category. This not only gives the Lambek calculus a category theoretic foundation — something **?** argues is an important, deeper level of meaning for logics — but it can also play the role of an alternative type of natural language semantics for the Lambek calculus (**?**, **?**), to be contrasted with the

Identity

$$\frac{}{A \to A}\ \mathit{Refl} \qquad \frac{A \to B \quad B \to C}{A \to C}\ \mathit{Trans}$$

Residuation

$$\frac{A \bullet B \to C}{A \to C \,/\, B}\ \mathit{Res}_{\bullet,/} \qquad \frac{A \bullet B \to C}{B \to A \setminus C}\ \mathit{Res}_{\bullet,\setminus}$$

$$\frac{A \to C \,/\, B}{A \bullet B \to C}\ \mathit{Res}_{/,\bullet} \qquad \frac{B \to A \setminus C}{A \bullet B \to C}\ \mathit{Res}_{\setminus,\bullet}$$

Associativity

$$\frac{}{A \bullet (B \bullet C) \to (A \bullet B) \bullet C}\ \mathit{Ass}_1 \qquad \frac{}{(A \bullet B) \bullet C \to A \bullet (B \bullet C)}\ \mathit{Ass}_2$$

Table 3.4: Residuation-based presentation of the Lambek calculus

Identity

$$\frac{}{A \to A}\ \mathit{Refl} \qquad \frac{A \to B \quad B \to C}{A \to C}\ \mathit{Trans}$$

Application

$$\frac{}{A \bullet (A \setminus B) \to B}\ \mathit{Appl}\backslash \qquad \frac{}{(B \,/\, A) \bullet A \to B}\ \mathit{Appl}/$$

Co-Application

$$\frac{}{A \to B \setminus (B \bullet A)}\ \mathit{Coappl}\backslash \qquad \frac{}{A \to (A \bullet B) \,/\, B}\ \mathit{Coappl}/$$

Monotonicity

$$\frac{A \to B \quad C \to D}{B \setminus C \to A \setminus D}\ \mathit{Mon}_{\setminus} \qquad \frac{A \to B \quad C \to D}{A \bullet C \to B \bullet D}\ \mathit{Mon}_{\bullet} \qquad \frac{A \to B \quad C \to D}{C \,/\, B \to D \,/\, A}\ \mathit{Mon}_{/}$$

Associativity

$$\frac{}{A \bullet (B \bullet C) \to (A \bullet B) \bullet C}\ \mathit{Ass}_1 \qquad \frac{}{(A \bullet B) \bullet C \to A \bullet (B \bullet C)}\ \mathit{Ass}_2$$

Table 3.5: Došen's presentation of the Lambek calculus

more standard semantics for type-logical grammars in the tradition of **?** we have seen before.

An alternative combinatorial representation of residuation is found in Table 3.5. This presentation uses the two application principles we have derived above as axioms, and adds two additional principles of co-application, easily obtained from the identity on the product formulas together with a residuation step.

$$\dfrac{\dfrac{}{A \bullet B \to A \bullet B}\ \mathit{Refl}}{A \to (A \bullet B) \mathbin{/} B}\ \mathit{Res}_{\bullet,/} \qquad \dfrac{\dfrac{}{B \bullet A \to B \bullet A}\ \mathit{Refl}}{A \to B \mathbin{\backslash} (B \bullet A)}\ \mathit{Res}_{\bullet,\backslash} \qquad \mathit{units}$$

The advantage of this presentation is that, besides transitivity, the only recursive rules are the monotonicity principles for the three connectives. This makes this presentation especially convenient for some types of inductive proofs. For example, **?** uses this presentation to show that the Lambek calculus is sound and complete for Kripke models, and these results extend to the multimodal calculi we have seen in Section 2.1 (**?**, **?**, **?**).

Compared to sequent calculus and natural deduction, the combinatorial calculi are not well-suited for proof search. We do not have the subformula property, and, contrary to the cut rule in the sequent calculus, the transitivity rule is essential to prove many theorems. However, the connections to algebra, category theory and model theory make the combinatorial representation a valuable alternative presentation of the Lambek calculus[2].

3.1.4 Towards proof nets

With so many proof systems available for type-logical grammars, the reader may be justified to wonder about the need for yet another one. First, let's emphasise that having many different proof systems is a good thing: having multiple, equivalent characterisations of the same class of provable statements is a type of evidence that this class is mathematically a 'natural' class. When proving a meta-theorem about the logic, having multiple equivalent proof systems allows us to pick and choose the formalism most convenient for proving the theorem: sequent calculus for the subformula property, natural deduction for the Curry-Howard isomorphism and the combinatorial system for soundness and completeness with respect to Krikpe models.

As noted before, when we use type-logical grammars as a tool for linguistic modelling we want to verify its predictions using a systematic way to enumerate all different proofs for a sentence in our grammar. This way, we verify our grammars generate all and only the right readings for each grammatical sentence.

With this purpose in mind, natural deduction proofs are *almost* the right proof system. The implication only fragment is very well-behaved and can be used for proof search (**?**, Section 2.6.2) (**?**, Section 2.2). However, as we have seen, the problem with natural deduction is the $\bullet E$ rule. Now, while the $\bullet E$ rule — to say the very least — does not play a very important part in the linguistic applications of type-logical grammars, there are connectives

[2]While all type-logical grammars discussed in Chapter 2 have a sequent and natural deduction calculus, not all type-logical grammars have a combinatorial representation of this type. For example, hybrid type-logical grammars and lambda grammars do not appear to have a way to summarise all grammar combinatorics using this type of axiomatisation.

of type-logical grammars, notably $\Diamond$ and $\exists$, for which the natural deduction elimination rules have the same bad properties as the $\bullet E$ rule, but whose linguistic applications are much more convincing.

In the literature on Lambek calculus proof search, many authors have studied ways of restricting the form of proofs in the sequent calculus in such a way that only one proof is found for each different reading (**?**, **?**, **?**). However, these systems were generally restricted to the implicational fragment. With the advent of linear logic and a better understanding of the logical symmetries of the sequent calculus, **?** presented a system of focused sequent proof search. Focusing is an important topic in its own right, but I will little more to say about it beyond a few brief remarks.

Proof nets are hypergraph-based representations of proofs. They can be seen as a sort of parallelised sequent calculus or, alternatively, as a multi-conclusion natural deduction. Like natural deduction, different proof nets correspond to proofs which are different for interesting reasons. Like the sequent calculus the $\bullet E/\bullet L$ rule requires no special treatment. Because there are no inessential rule permutation to consider, cut elimination for proof nets is trivial (**?**, Appendix B.4).

Proof nets were originally introduced for linear logic (**?**), but soon adapted to the Lambek calculus (**?**). One of the important questions which will concern us throughout the rest of this book is how to adapt proof nets to the various types of modern type-logical grammars we have seen in Chapter 2. For multimodal categorial grammars extended with unary connectives, **?** already provide a proof net calculus which we will briefly discuss in Section 5.1.3.

Whereas focusing is a type of backward chaining sequent proof search, with restrictions on the allowed rule applications, proof search with proof nets is better seen as a type of forward chaining proof search[3]. This has the advantage that it directly computes the structure for calculi with structured sequents. This is not that relevant for the Lambek calculus, where we are given the sequence of input formulas, but many other calculi (even the non-associative Lambek calculus) require us to compute the structure of the input formulas (e.g., a binary branching tree for **NL**). Proof nets also do not suffer from the sequent calculus problem (which applies equally to focused proofs) of deciding how to partition the antecedent formulas among the premisses of a binary rule. In addition, proof net proof search has more degrees of freedom, which at least potentially allows it to better detect certain types of early failure. While these disadvantages of focused proof search can at least be partially solved by making proof search slightly more complicated, and while the sequents for which proof nets perform better are almost certainly balanced by other types of sequents for which focused sequents perform better, we will leave a detailed performance comparison to future research.

[3]Given that we use linear logic, many of the standard drawbacks of forward chaining (**?**, Section 9.3) do not apply. For example, we never need to worry about 'irrelevant' formulas, since all our formulas must be used in the proof.

$$\frac{}{\vdash A, A^{\perp}}\ Ax \qquad \frac{\vdash \Gamma, A \quad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}\ Cut$$

$$\frac{\vdash \Gamma, A \quad \vdash B, \Delta}{\vdash \Gamma, A \otimes B, \Delta}\ \otimes \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr$$

Table 3.6: One-sided sequent calculus rules for multiplicative linear logic

3.2 Multiplicative proof nets

Multiplicative linear logic contains only the connective for conjunction, '$\otimes$' called *tensor*, the connective for disjunction, '$\parr$' called *par*, and negation, written as a '$\perp$' superscript. Negation can be restricted to atomic formulas by using the linear logic versions of the standard de Morgan rules and double negation elimination to remove negations over complex formulas — the right-hand column lists the standard de Morgan rules for comparison.

$$(A \otimes B)^{\perp} = B^{\perp} \parr A^{\perp} \qquad \neg(A \wedge B) = \neg A \vee \neg B \tag{3.1}$$

$$(A \parr B)^{\perp} = B^{\perp} \otimes A^{\perp} \qquad \neg(A \vee B) = \neg A \wedge \neg B \tag{3.2}$$

$$(A^{\perp})^{\perp} = A \qquad \neg\neg A = A \tag{3.3}$$

Formulas are then formed by literals (atoms and their negations) and the tensor/conjunction and par/disjunction connectives. Linear implication '$\multimap$' is defined in terms of disjunction and negation, with the standard definition $A \multimap B \equiv_{def} A^{\perp} \parr B$ (corresponding to $A \Rightarrow B \equiv_{def} \neg A \vee B$ in classical logic).

The advantage of setting up the logic in this way is that we can formulate a very simple one-sided sequent calculus for it, translating two-sided sequents $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ into one-sided sequents $\vdash A_1^{\perp}, \ldots, A_n^{\perp}, B_1, \ldots, B_m$ and with the logical rules shown in Table 3.6. The sequent calculus comma is implicitly assumed to be associative and commutative.

As an example, let's take the intuitionistic sequent of Equation 3.4 below.

$$(A \multimap B) \otimes (B \multimap C) \vdash A \multimap C \tag{3.4}$$

It is the reflection in intuitionistic linear logic of either of the two Lambek calculus sequents of Equation 3.5.

$$(A\backslash B) \bullet (B\backslash C) \vdash A\backslash C \qquad (C/B) \bullet (B/A) \vdash C/A \tag{3.5}$$

Equation 3.6 shows the corresponding classical one-sided sequent.

$$\vdash (A \otimes B^{\perp}) \parr (B \otimes C^{\perp}), A^{\perp} \parr C \tag{3.6}$$

One possible proof of this sequent is the following.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash B, B^\perp} \quad \dfrac{}{\vdash A, A^\perp}}{A \otimes B^\perp, B, A^\perp}\otimes \quad \dfrac{}{\vdash C, C^\perp}Ax}{\vdash A \otimes B^\perp, B \otimes C^\perp, A^\perp, C}\otimes}{\vdash A \otimes B^\perp, B \otimes C^\perp, A^\perp \parr C}\parr}{\vdash (A \otimes B^\perp) \parr (B \otimes C^\perp), A^\perp \parr C}\parr$$

There are three other cut-free proofs of this same sequent: the order of the two par rules and of the two tensor rules with respect to each other can be reversed. However, these rule orderings are inessential, just an artefact of the very explicit way in which the sequent calculus manages contexts. There is only one proof net corresponding to these four sequent proofs. When proof nets differ, they differ for interesting reasons. In the intuitionistic case we have one (cut-free) proof net for each lambda term (modulo equivalence).

In this particular case, the problem is not resolved when moving to (intuitionistic) natural deduction: as shown below, the last rule in the natural deduction proof of $(A \multimap B) \otimes (B \multimap C) \vdash A \multimap C$ can be either $\otimes E$ or $\multimap I$ and these two proofs are equivalent (by a commutative conversion).

$$\dfrac{(A \multimap B) \otimes (B \multimap C) \qquad \dfrac{\dfrac{\dfrac{[A]^2 \quad [A \multimap B]^1}{B}\multimap E \quad [B \multimap C]^1}{C}\multimap E}{A \multimap C}\multimap I_2}{A \multimap C}\otimes E_1$$

$$\dfrac{\dfrac{(A \multimap B) \otimes (B \multimap C) \qquad \dfrac{\dfrac{[A]^2 \quad [A \multimap B]^1}{B}\multimap E \quad [B \multimap C]^1}{C}\multimap E}{C}\otimes E_1}{A \multimap C}\multimap I_2$$

Proof nets are a type of (hyper)graph where the nodes are formulas and the (hyper)edges are called *links*. The links for multiplicative linear logic are shown in Table 3.7. The *conclusions* of a link are those formulas connected from the bottom of the link to the top of the formula; the *premisses* of a link are the formulas attached (from their bottom) to the top of the link. The axiom link has no premisses and two conclusions A and $A^\perp$ (the order of the two conclusions doesn't matter). The cut link has two premisses A and $A^\perp$ and no conclusions (the order of the two premisses doesn't matter). The tensor and par link have the same premisses A and B (in that order) and as conclusion $A \otimes B$ and $A \parr B$ respectively. The par link is distinguished visually by using a connected pair of dotted lines.

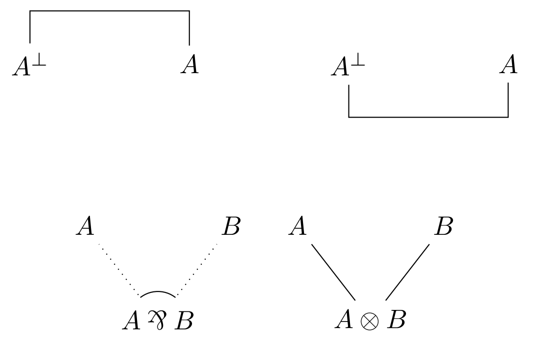


Table 3.7: Links for proof nets in multiplicative linear logic

Definition 3.1 *We define proof nets inductively as follows.*

Axiom *Axiom links are proof nets with conclusions $A, A^\perp$.*

Par *If Π is a proof net with conclusions Γ, A, B, we can connect A and B as the premisses of a new par link with conclusion $A \parr B$ to form a proof net of $\Gamma, A \parr B$.*

Tensor *If Π_1 and P_2 are two disjoint proof nets with conclusions Γ, A and B, Δ respectively, we can connect A and B as the premisses of a new tensor link with conclusion $A \otimes B$ to form a proof net with conclusions $\Gamma, A \otimes B, \Delta$.*

Cut *If Π_1 and P_2 are two disjoint proof nets with conclusions Γ, A and $A^\perp, \Delta$ respectively, we can connect A and $A^\perp$ as the premisses of a new cut link to form a proof net with conclusions Γ, Δ.*

The inductive rules for proof net construction follow the rules of the sequent calculus very closely, and this makes it easy prove the systems are equivalent with respect to derivable sequents (up to a 1-1 matching between the rules of the sequent proof and the links of the proof nets).

The sequent proof of $\vdash (A \otimes B^\perp) \parr (B \otimes C^\perp), A^\perp \parr C$ translates into the proof net shown in Figure 3.1.

Although the four possible proofs differ in the order of application of the inductive rules for proof nets — as in the sequent calculus, we can inverse the order of the two tensor rules, and similarly for the two par rules — these all produce the proof net shown in Figure 3.1.

3.2.1 Proof structures, modules and components

The inductive definition of proof nets has the drawback that it is not immediately obvious whether of a graph which looks like a proof net actually is one. That is, given a *proof structure* — a graph composed of formulas and links

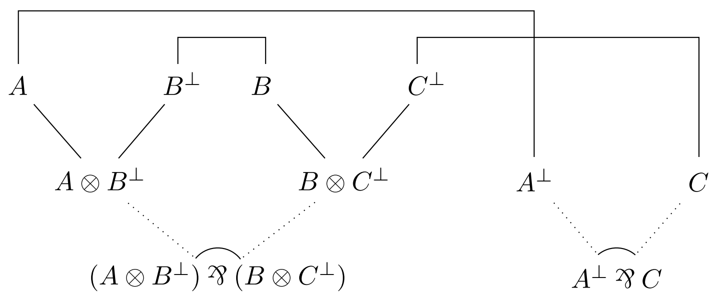


Figure 3.1: Proof net of $(A \otimes B^\perp) \parr (B \otimes C^\perp), A^\perp \parr C$

just like proof net — we want to determine whether or not it corresponds in some way to the inductive definition of proof nets.

Definition 3.2 *A* proof structure *is a set of formula occurrences connected by the links of Table 3.7 such that:*

- *each formula is the premiss of at most one link,*
- *each formula is the conclusion exactly one link.*

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

In other words, proof structures have two types of formulas, *internal formulas*, which are attached to links both from above and from below, and *conclusions*, which are attached only from above. A proof structure with conclusions $A_1, \ldots, A_n$ corresponds to the sequent $\vdash A_1, \ldots, A_n$.

The definition of proof structure has an asymmetry with respect to premisses and conclusions: we can have formulas which are not the premiss of any link (the conclusions of the proof structure) but not formulas which are not the conclusion of any link. In the context of proof search using proof structures, it is often useful to consider slightly more general structures called *modules*.

Definition 3.3 *A* module *is a set of formula occurrences connected by the links of Table 3.7 such that:*

- *each formula is the premiss of at most one link,*
- *each formula is the conclusion of at most one link.*

Formulas which are not the premiss of any link are the conclusions *of the module. Formulas which are not the conclusion of any link are the* hypotheses *of the module.*

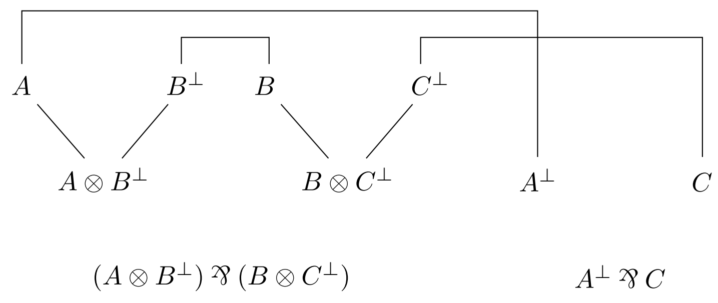


Figure 3.2: Components of the proof net from Figure 3.1

Modules can have hypotheses as well as conclusions. A module without axiom links and only atomic hypotheses is sometimes called a *proof frame* (**?**) and it represents the initial state of proof search using proof structures. In the general case, we can represent stages of proof search as modules where all hypotheses are atomic. More specifically, we add axiom links to a proof frame until we obtain a proof structure.

We also want to refer to substructures of proof structures (and, similarly to subnets of proof nets). There is one type of substructure, the component, which will be particularly useful.

Definition 3.4 *A* component *is a maximal, connected submodule of a module which does not contain any par links.*

From a proof structure (or a module) we obtain its components by simply removing all par links (but keeping its premiss and conclusion formulas). The components are the connected substructures of the resulting structure. Figure 3.2 shows the components of the proof net from Figure 3.1. The proof net has three components. At least for proof *nets*, a proof net with p par links has $p + 1$ components, and, moreover, each component must be acyclic (it is connected by definition, of course). Two of the components shown in Figure 3.2 consist of only a single vertex.

3.2.2 Correctness conditions

Not all proof structures are proof nets. Figure 3.3 shows some example proof structures which are not proof nets. Proof structure (a) is not a proof net because the tensor case of the inductive definition requires us to combine two disjoint proof nets whereas we have only a single A axiom. Similarly, (b) and (c) are not a proof nets, because the par case requires us to attach the link to a single proof net, whereas we have two disjoint axioms. We can also show these are not proof nets by showing the sequents (a) $\vdash A \otimes A^{\perp}$, (b) $\vdash (A^{\perp} ⅋ B) \otimes B^{\perp}, A$, and (c) $\vdash A, A^{\perp} ⅋ B, B$ are underivable. However, neither of these options is very attractive. We want a method to identify proof

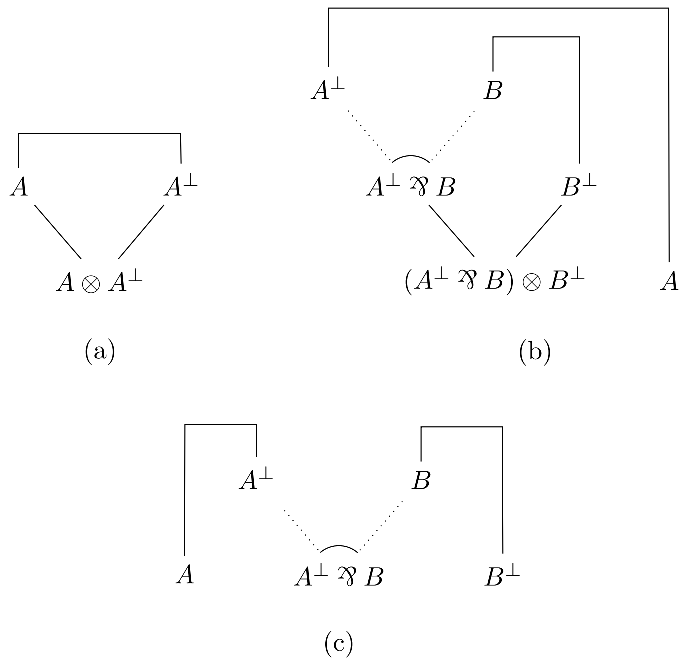


Figure 3.3: Proof structures which are not proof nets

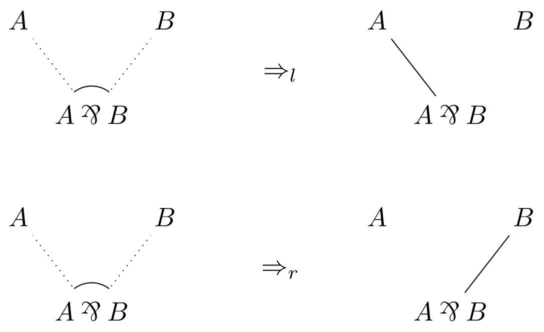


Table 3.8: Switchings for par links

nets, using only properties of the graphs. A *correctness condition* provides a way of identifying proof nets among proof structures. Several correctness conditions exist (**?**, **?**, **?**).

Acyclicity and connectedness

The best-known correctness condition is the switching condition of **?**. We replace each par link in one of the ways shown in Table 3.8. A switching transforms a proof structure into a standard graph, with formula occurrences as vertices.

Theorem 3.5 *A proof structure is a proof net when all its switchings are acyclic and connected.*

? prove that proof nets defined by this correctness condition are exactly those which correspond to derivable sequents. In other words, we now have two definitions of proof nets: 1) an inductive definition, and 2) proof structures satisfying a correctness condition, but these two definitions refer to the same class of structures.

Given that each par link has two possible switchings, it would appear that this correctness condition is hard to verify: for a proof structure with p par links, there are 2^p possible switchings. We can do a bit better: with respect to acyclicity and and connectedness, when the left and right switching connect to the same component, we only have to consider one of the two switchings. For the proof net of Figure 3.1 we therefore have to consider only one switching, since the premisses of both par links all connect to the same component.

The acyclicity and connectedness condition is very convenient for showing a proof structure is *not* a proof net, since a single switching serves as a counterexample. Figure 3.4 shows how the proof structures of Figure 3.3 are not proof structures. Proof structure (a) is cyclic, proof structure (b) has a switching which is cyclic and disconnected shown in the figure, and proof structure (c) has a disconnected switching. It is worth noting that the alternative switching for (b) *is* acyclic and connected, and that, in general, all but one of the switchings of an incorrect proof structure can by acyclic and connected. However, there are very efficient (linear time) algorithms for verifying a proof structure is a proof net (**?**, **?**).

Graph contractions

There is an alternative correctness condition based on graph contractions. As we will see, this criterion is particularly convenient for proof search and we will see it return — with some variations — in the sections and chapters which follow. Table 3.9 presents the contraction rules due to **?** for which he proves the following theorem.

Theorem 3.6 *A proof structure is a proof net if and only if it contracts to a single vertex using the contractions of Table 3.9.*

Since variants of this contraction criterion are important in many of the later chapters, I believe it is worthwhile to perform an example contraction of a proof structure to a single vertex in full detail and, in addition, give some

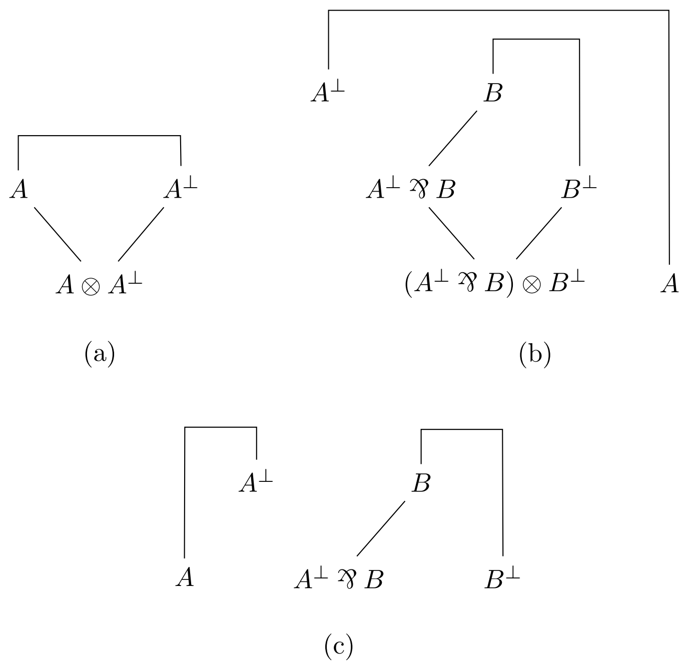


Figure 3.4: Switchings for the incorrect proof structures from Figure 3.3 exhibiting cycles or disconnectedness.

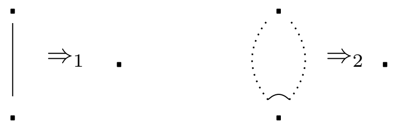


Table 3.9: Contractions for multiplicative linear logic. The contractions can only be applied when the two vertices on the left hand side are distinct. Contraction 2 requires that the dotted links are paired, as indicated by the connecting link.

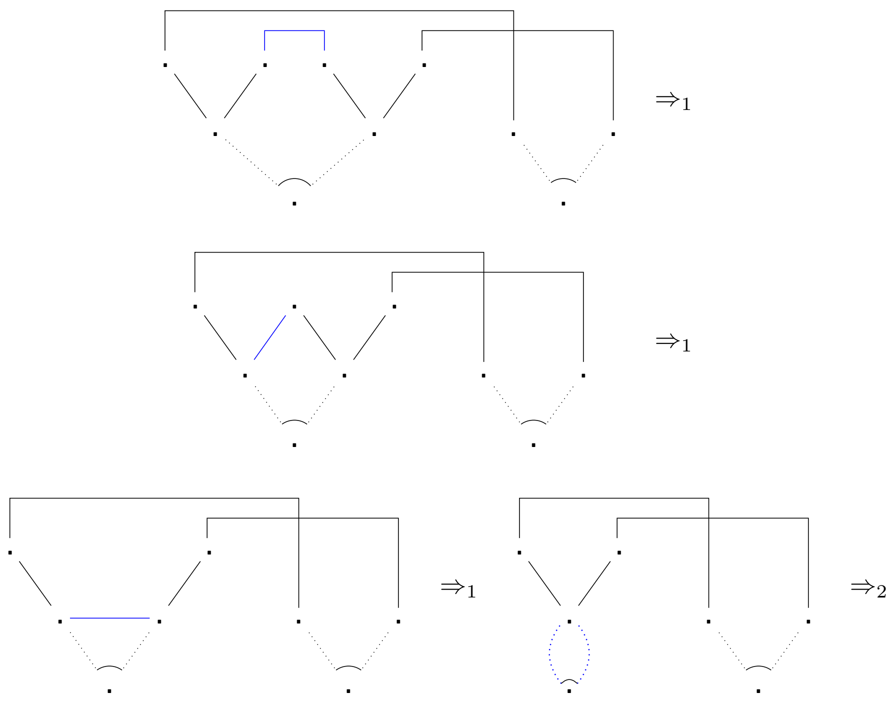


Figure 3.5: Start of the conversion sequence

examples of where the contraction condition fails for proof structures which are not proof nets.

Figure 3.5 shows, at the top of the figure, the proof structure of Figure 3.1 with all formula labels removed. The first edge to be contracted is drawn in blue. Removing the blue edge and identifying the two vertices produces the structure shown on the middle of Figure 3.5. Contracting the blue edge of this structure produces the structure shown at the bottom left of Figure 3.5 and another contraction of the blue edge produces the structure shown on the bottom right.

From the point of view of the leftmost par link, what we have done so far is reduce the path between the two linked premisses of the link from a three link path until the two premisses of the par link have 'joined' into a single vertex. This is the correct configuration for the 2 rewrite, and the result is shown on the top left of Figure 3.6. Figure 3.6 shows that four more 1 rewrites also reduce the length of the path between the two premisses of the second par link to zero. We complete the conversion sequence by a 2 contraction, showing the proof structure is a proof net.

It is equally important that the contraction condition fails for structures which are not proof nets. Figure 3.7 shows the proof structures of Figure 3.3 where all applicable contractions have been applied. None of these structures

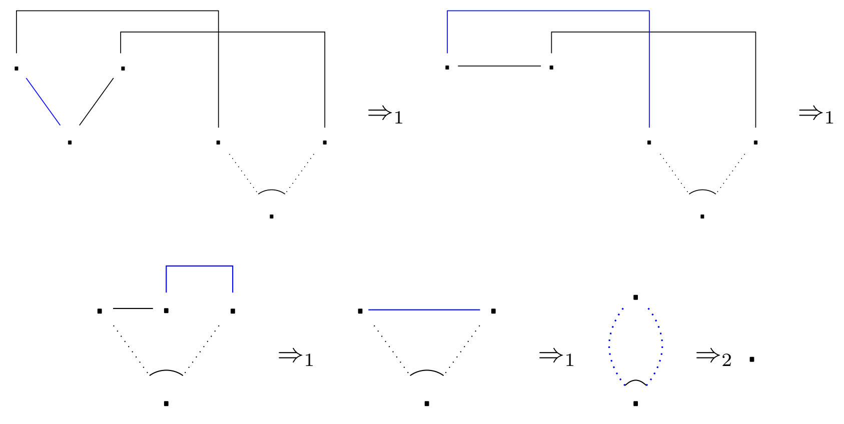


Figure 3.6: End of the conversion sequence of Figure 3.5

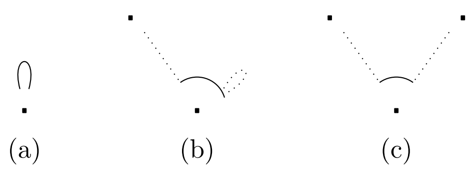


Figure 3.7: Contraction failure for the incorrect proof structures of Figure 3.3

are single vertices, a no contractions can be applied to any of them: (a) has a loop, and therefore fails the condition of the 1 contraction; for (b) and (c) neither graph is a single vertex, and no contractions apply.

3.2.3 Intuitionism

From the point of view of linear logic, intuitionism can bee seen as a restriction on the allowed formulas. We distinguish between negative formulas $\mathcal{N}$ and positive formulas $\mathcal{P}$.

$$\mathcal{N} ::= p^{\perp} \mid \mathcal{N} \parr \mathcal{N} \mid \mathcal{N} \otimes \mathcal{P} \mid \mathcal{P} \otimes \mathcal{N} \tag{3.7}$$

$$\mathcal{P} ::= p \quad \mid \mathcal{P} \otimes \mathcal{P} \mid \mathcal{P} \parr \mathcal{N} \mid \mathcal{N} \parr \mathcal{P} \tag{3.8}$$

In a commutative setting we need only one of the pair $\mathcal{N} \otimes \mathcal{P}$ and $\mathcal{P} \otimes \mathcal{N}$ (we will keep only the second). Similarly, we can restrict ourselves to one of $\mathcal{P} \parr \mathcal{N}$ and $\mathcal{N} \parr \mathcal{P}$ (we will again keep the second).

Given the formula restrictions for positive and negative formulas, we can see which combinations are excluded: par cannot combine two positive formulas, whereas tensor cannot combine two negative formulas. It is this property which allows us to prove following property.

Proposition 3.7 *All derivable sequents consisting only of positive and negative formulas have exactly one positive formula.*

Proof This is easy to show by induction, using the inductive definition of proof nets. □

Given that negative formulas correspond to formulas on the left hand side of the turnstile and positive formulas to formulas on the right hand side, Proposition 3.7 shows that the formula restriction produces intuitionistic (that is, single conclusion) sequents.

The correspondence between the restricted fragment of linear logic and intuitionism is even more clear when we consider the formula restriction as follows.

$$\mathcal{N} ::= p^{\perp} \mid \mathcal{N} \parr \mathcal{N} \mid \mathcal{P} \otimes \mathcal{N} \qquad \mathcal{N} ::= \overset{-}{p} \mid \mathcal{N} \overset{-}{\otimes} \mathcal{N} \mid \mathcal{P} \overset{-}{\multimap} \mathcal{N} \tag{3.9}$$

$$\mathcal{P} ::= p \quad \mid \mathcal{P} \otimes \mathcal{P} \mid \mathcal{N} \parr \mathcal{P} \qquad \mathcal{P} ::= \overset{+}{p} \mid \mathcal{P} \overset{+}{\otimes} \mathcal{P} \mid \mathcal{N} \overset{+}{\multimap} \mathcal{P} \tag{3.10}$$

The column on the right presents the intuitionistic formulas using $\otimes$ and $\multimap$ and uses $+$ and $-$ to distinguish between positive and negative formulas.

Given all this, it is easy to produce the links for intuitionistic proof nets. Table 3.10 presents the logical links for multiplicative intuitionistic linear logic proof nets. The distinction between par links and tensor links is determined by the corresponding classical formula according to Equations 3.9 and 3.10. Therefore, the negative $\otimes$ and positive $\multimap$ links are par links, and the positive $\otimes$ and negative $\multimap$ links are tensor links.

To try and prove a sequent $A_1, \ldots, A_n \vdash C$ we decompose the formulas according to the links of the figure, starting with

$$\overset{-}{A_1} \ldots \overset{-}{A_n} \overset{+}{C}$$

and then we continue as for classical multiplicative linear logic: we continue unfolding the formulas until we reach the atomic subformulas. As before, the logical links essentially produce a subformula tree with the additional marking of the polarity of each subformula and marking the distinction between conjunctive (tensor) and disjunctive (par) nodes.

Figure 3.8 shows the proof net from Figure 3.1 translated to its intuitionistic version. The important point is that only the formula labels at the vertices have changed (and if we want, we can translate the intuitionistic formulas back to their classical equivalents). This means that the correctness conditions are unchanged from the classical versions.

3.2.4 Non-commutativity

A number of authors have studied proof nets for non-commutative (or cyclic) versions of linear logic. One easy way to obtain non-commutativity is to split linear implication into a left and right version, and to require that the axiom links are planar. Intuitionism combined with planar axiom links gives

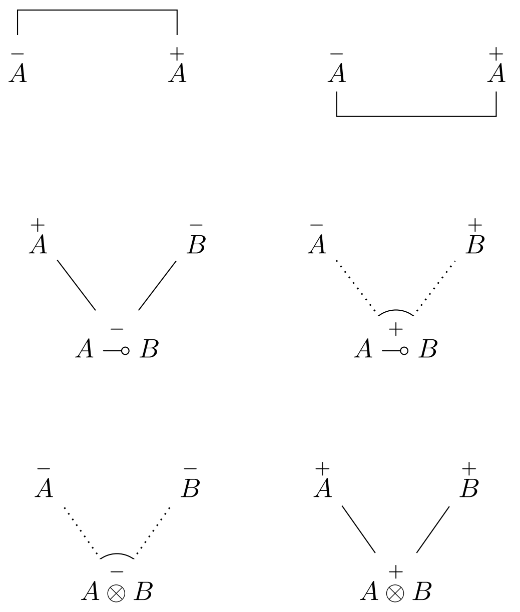


Table 3.10: Logical links for MILL proof structures

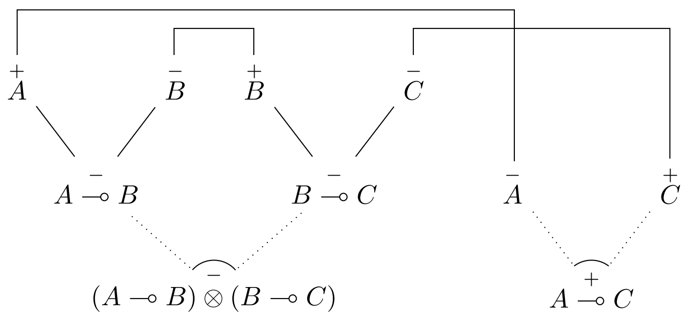

$\overset{+}{A}$
$\bar{B}$
$\overset{+}{B}$
$\bar{C}$
$A \overset{-}{\multimap} B$
$B \overset{-}{\multimap} C$
$\bar{A}$
$\overset{+}{C}$
$(A \multimap B) \overset{-}{\otimes} (B \multimap C)$
$A \overset{+}{\multimap} C$

Figure 3.8: Intuitionistic proof net of $(A \multimap B) \otimes (B \multimap C) \vdash A \multimap C$

us proof nets for the Lambek calculus (?). There are other ways of enforcing non-commutativity. For example, ? extend the correctness criterion of ? to the non-commutative case.

A problem with planarity is that it is an all-or-nothing criterion: either the axiom links of a proof net are planar, or they are not. However, for the applications we have seen, we want a way of combining a commutative logic with a non-commutative one, that is, to have a logic which is only partially commutative, relaxing non-commutativity only when required for the linguistic applications we have seen in Section 1.5.

Some partially commutative linear logics have been proposed in the literature. De Groote (?) proposes a logic which combines (commutative) multiplicative intuitionistic linear logic with the (non-commutative) Lambek calculus. ? show how this logic can translate syntactic analyses in the style of ?. However, in the terms of Section 2.1, this logic is simply a multimodal logic using an associative, commutative mode and an associative, non-commutative mode. It therefore poses no particular challenges for a proof net implementation as used for similar multimodal logics (see Section 5.1.3 below).

Another partially commutative logic is the pomset logic of ?. This logic adds a non-commutative connective '<' (before) to multiplicative linear logic. The formulas in the sequents of this logic form a partially ordered multiset (a pomset), and the proof net calculus for the logic is a simple extension of the multiplicative proof nets of ?.

One important logic in this respect is the non-commutative logic of ?, which combines the standard ⅋ and $\otimes$ connectives of commutative multiplicative linear logic with a non-commutative conjunction $\odot$ (corresponding to the Lambek calculus $\bullet$) and a non-commutative disjunction ∇. This allows us to define the two non-commutative implications as $B\backslash A \equiv B^{\perp}\nabla A$ and $A/B \equiv A\nabla B^{\perp}$. The logic has both a simple sequent calculus, and a simple proof net calculus (?), using a correctness criterion which is a variant of ? trip condition.

It is unclear to what extent the logics and proof net calculi of ? or ? can be used to solve the problems with the Lambek calculus described in Section 1.5: I conjecture that some (like medial extraction) will not pose too much of a problem, but that others, such as the Dutch verb clusters and quantifier scope will be more problematic. But I will leave these questions for future research.

With respect to partially enforcing non-commutativity, we will consider two options in what follows:

1. in Chapter 4, we will use first-order variables to ensure non-commutativity without requiring planarity,

2. in Chapter 5, we will use proof nets for multimodal categorial grammars (and extensions thereof) which can implement the multimodal strategy of ?; that is, we take a non-associative base logic, and provide controlled access to the desired structural rules.

3.3 Conclusions

This chapter has introduced the motivations and basic definitions for proof nets. As an inherently redundancy-free representation of proofs, they appear to be a good candidate for the enumeration of proofs in different type-logical grammars. However, the proof nets we have seen so far are for multiplicative linear logic, which is a fully commutative logic. Our goal, in the next two chapters will be to provide proof nets for all different type-logical grammars of the previous chapter.

This first framework, first order linear logic, is simple and standard. Yet, as we will see in Chapter 4, it will prove to be powerful enough to serve as a theorem prover for many type-logical grammars. Even for calculi which do not have a direct translation into first-order linear logic, we can often use it for approximation, as a way of quickly discovering underivable statement (for example, because they are underivable in intuitionistic linear logic). So even though first-order linear logic cannot handle all type-logical grammars, it still operates somewhere under the surface for most of our theorem provers (**?**, **?**, **?**).

The second framework, based on graph rewriting, is more general. The components of its proof nets represent fully structured antecedents which makes it easy to represent arbitrary structural rules as tree rewrites (provided they do not use copying or duplication). As we will see in Chapter 5, we will be able to accommodate all modern type-logical grammar in this second proof net framework by using suitable variations on the allowed structures and links. This extra expressivity comes at a price, though, and makes proof search more complicated.

4 First-order linear logic

4.1 Introduction

This chapter will focus on multiplicative intuitionistic first-order linear logic — hereafter simply first-order linear logic or MILL1 — as a first general computational framework for type-logical grammars, with the next chapter devoted to the more flexible graph rewriting framework. First-order linear logic has a number of good properties which make it suited as a framework for computational linguistics:

1. first-order linear logic is a well-understood fragment of linear logic,
2. has a tight connection to formal language theory (for the Horn clause fragment),
3. embeds the Lambek calculus (**?**),
4. is NP complete, that is, no increase in the complexity of proof search with respect to the Lambek calculus,
5. has a simple proof net calculus.

First-order linear logic also embeds a number of other type-logical grammars, namely lambda grammars, (an important fragment of) the Displacement calculus and hybrid type-logical grammars (**?**, **?**). In addition, it has a dedicated theorem prover based on proof nets, with facilities for designing grammars in lambda grammars, hybrid type-logical grammars and displacement grammars (**?**).

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

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

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

$$\frac{\Gamma, A \vdash C}{\Gamma, \exists x.A \vdash C}\ [L\exists^*] \qquad \frac{\Gamma \vdash A[x := t]}{\Gamma \vdash \exists x.A}\ [R\exists]$$

$$\frac{\Gamma, A[x := t] \vdash C}{\Gamma, \forall x.A \vdash C}\ [L\forall] \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\ [R\forall^*]$$

Table 4.1: The sequent calculus for first-order intuitionistic multiplicative linear logic.

4.2 Proof theory

A *sequent* or a *statement* is an expression of the form $A_1, \ldots, A_n \vdash C$ (for some $n \geq 0$), which we will often shorten to $\Gamma \vdash C$. We call Γ the *antecedent*, formulas A_i in Γ *antecedent formulas*, and C the *succedent* of the statement. We assume the sequent comma is both associative and commutative and treat statements which differ only with respect to the order of the antecedent formulas to be equal. Table 4.1 shows the sequent calculus rules for first-order multiplicative intuitionistic linear logic. The $R\forall$ and $L\exists$ rule have the standard side condition that there are no free occurrences of x in Γ and C.

Cut-free proof search for the sequent calculus is decidable (the decision problem is NP complete (**?**)), and sequent proof search can be used as a practical decision procedure (**?**). Decidability presupposes both cut elimination (which, as usual, is a simple enough proof even though there are many rule permutations to verify) and a restriction on the choice of t for the $L\forall$ and $R\exists$ rules. A standard solution is to use unification for this purpose, effectively delaying the choice of t to the most general term required by the axioms in backward chaining cut-free proof search. This of course requires us to verify the eigenvariable conditions for the $R\forall$ and $L\exists$ rules are still satisfied after unification. We can see this in action in the following failed attempt to prove $\forall y[a \otimes b(y)] \vdash a \otimes \forall x.b(x)$ (the reader can easily verify all other proof attempts fail as well).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \vdash a}\ Ax \quad \dfrac{\dfrac{Y = x}{b(Y) \vdash b(x)}\ Ax}{b(Y) \vdash \forall x.b(x)}\ \forall R*}{a, b(Y) \vdash a \otimes \forall x.b(x)}\ R\otimes}{a \otimes b(Y) \vdash a \otimes \forall x.b(x)}\ L\otimes}{\forall y.[a \otimes b(y)] \vdash a \otimes \forall x.b(x)}\ L\forall$$

Tracing the proof from the endsequent upwards to the axioms, we start by replacing y by a fresh metavariable Y to be unified later, then follow the proof upwards to the axioms. For the b predicates, we compute the most general unifier of x and Y, which is x. But then, the antecedent of the $\forall R$ rule becomes $b(x)$, which fails to respect the eigenvariable condition for x. We can improve on the sequent proof procedure for first-order linear logic, even exploiting some of the rule permutabilities (**?**). However, in the next section (Section 4.3) we will present a proof net calculus for first order linear logic, following **?**, which *intrinsically* avoids the efficiency problems caused by rule permutations.

Before we do so, however, we will briefly recall how we can use first-order linear logic for modelling natural languages.

First-order linear logic and natural language grammars For type-logical grammars, a *lexicon* is a mapping from words to formulas in the corresponding logic. In first-order linear logic, this mapping is parametric for two position variables L and R, corresponding respectively to the left and right position of the string segment corresponding to the word. In general, for a sentence with n words, we assign the formula of word w_i (for $1 \leq i \leq n$) the string positions $i-1$ and i. This simply follows the fairly standard convention in the parsing literature to represent substrings of the input string by pairs of integers.

As noted by **?**, we can translate Lambek calculus formulas to first-order linear logic formulas as follows.

$$\|p\|^{x,y} = p(x,y) \tag{4.1}$$
$$\|A \bullet B\|^{x,z} = \exists y. \|A\|^{x,y} \otimes \|B\|^{y,z} \tag{4.2}$$
$$\|A \setminus C\|^{y,z} = \forall x. \|A\|^{x,y} \multimap \|C\|^{x,z} \tag{4.3}$$
$$\|C \,/\, B\|^{x,y} = \forall z. \|B\|^{y,z} \multimap \|C\|^{x,z} \tag{4.4}$$

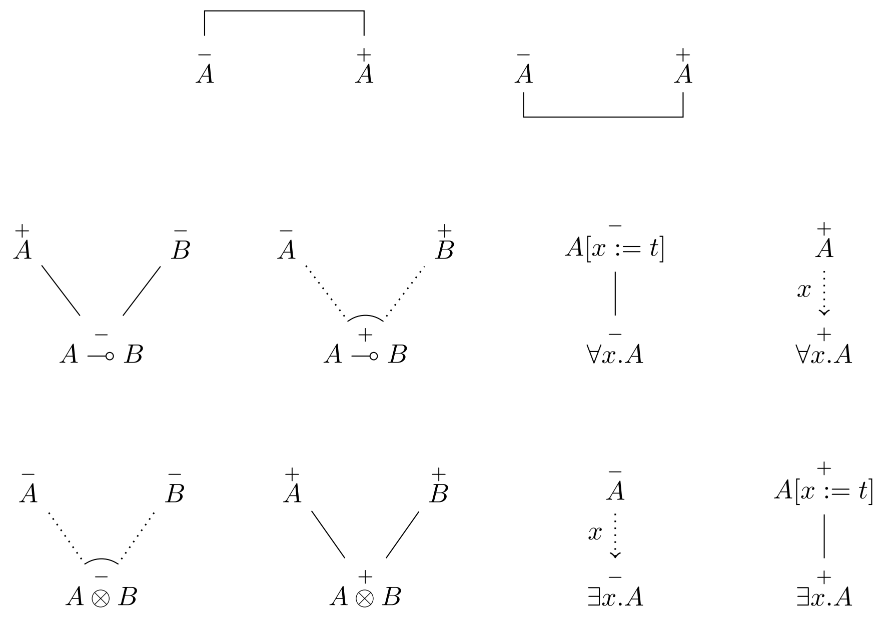
Equation 4.4 states that when C/B is a formula spanning string x, y (that is, having x as its left edge and y as its right edge), that means combining it with a formula B having y as its left edge and any z as its right edge will produce a formula C starting at x (the left edge of C/B) and ending at z (the right edge of B).

4.3 MILL1 proof nets

To change from multiplicative intuitionistic linear logic to first-order multiplicative intuitionistic linear logic (for brevity, I will call this system MILL1, or simply first-order linear logic) we need to change relatively little.

We will present proof nets for first-order linear logic from the perspective of using the calculus for theorem proving. To try and prove a sequent $A_1, \ldots, A_n \vdash C$ we decompose the formulas starting with

$$\overset{-}{A_1} \ldots \overset{-}{A_n}\, \overset{+}{C}$$



$\overset{-}{A}$
$\overset{+}{A}$
$\overset{+}{A}$
$\overset{-}{B}$
$A \overset{-}{\multimap} B$
$\overset{-}{A}$
$\overset{+}{B}$
$A \overset{+}{\multimap} B$
$A[x \overset{-}{:=} t]$
$\overset{-}{\forall x.A}$
x
$\overset{+}{\forall x.A}$
$A \overset{-}{\otimes} B$
$A \overset{+}{\otimes} B$
$\overset{-}{\exists x.A}$
$A[x \overset{+}{:=} t]$
$\overset{+}{\exists x.A}$

Table 4.2: Logical links for MILL1 proof structures

and continuing the unfolding until we reach the atomic subformulas using links of Table 4.2. This produces a formula decomposition tree with some annotations: we distinguish between positive and negative subformulas (formulas of negative polarity correspond to antecedent formulas in the sequent calculus, and those of positive polarity to succedent formulas), and some links are dotted.

Even though there are eight different logical links, these are divided into four different types:

1. the binary links drawn with solid lines are *tensor* links,
2. the binary links drawn with dotted lines are *par* links,
3. the unary links drawn with solid lines are *existential* links,
4. the unary links drawn with dotted links are *universal* links, universal links are labeled with the name of the eigenvariable and have an arrow pointing towards the conclusion of the link.

The tensor and par links are the same as those for multiplicative intuitionistic linear logic, only the quantifier links are new. We use the standard convention that each quantifier in a sequent uses a different eigenvariable (**?**, **?**).

After this unfolding step, we connect atomic formulas of opposite polarity using the axiom link (Table 4.2, top left). When all atomic formulas have been

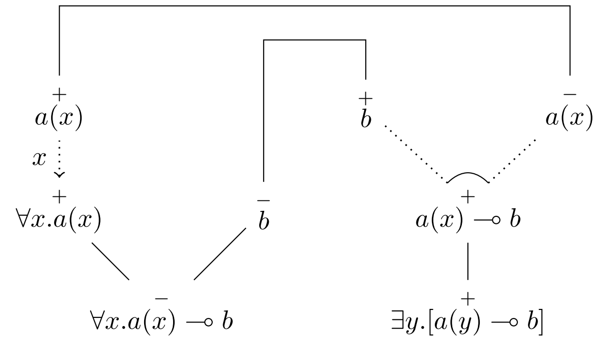

$\overset{+}{a(x)}$
x
$\overset{+}{\forall x.a(x)}$
$\overset{-}{b}$
$\overset{-}{\forall x.a(x) \multimap b}$
$\overset{+}{b}$
$\overset{-}{a(x)}$
$\overset{+}{a(x) \multimap b}$
$\overset{+}{\exists y.[a(y) \multimap b]}$

Figure 4.1: Proof structure which is not a proof net

connected this way, the resulting structure is called a proof structure. As an example, the sequent $\forall x.a(x) \multimap b \vdash \exists y.[a(y) \multimap b]$ has the proof structure shown in Figure 4.1.

In practice, we do not choose a term for the negative $\forall$ or for the positive $\exists$ link during formula unfolding but we rather use meta-variables for unfolding and unification during the axiom link connections. So the axiom link does not connect identical formulas but rather unifiable formulas and performs this unification. This is a rather standard theorem-proving strategy and has the result that we can read off the most general term for each negative $\forall$ and positive $\exists$ rule in our proof net. Some care must be taken when we repeatedly unify the positive and negative atomic subformulas of $\forall x.a(x) \multimap a(f(x,x))$, where the size of the term argument grows exponentially in the number of occurrences of the given formula ($a(x)$, $a(f(x,x))$, $a(f(f(x,x),f(x,x)))$, $a(f(f(f(x,x),f(x,x)),f(f(x,x),f(x,x))))$, $\ldots$). Even in these cases, we can ensure linear time unification by adopting a sharing strategy (**?**, **?**). In addition, for the typical uses of first-order linear logic which interest us — notably the applications discussed in Sections 4.4 and 4.5 — each quantifier binds exactly two occurrences of its variable, so we might even decide to exclude the case above, where the quantifier $\forall x$ binds three occurrences.

Returning to the example proof structure of Figure 4.1, the given sequent is underivable in linear logic and, in general, proof structures need not correspond to proofs. As before, proof structures which correspond to proofs are called *proof nets* and we can distinguish them from other proof structures using properties of the graph.

Acyclicity and connectedness

Table 4.3 shows **?** extension of the multiplicative switchings of **?** to the first-order case. The switching for the $\forall$ link allows us to connect the conclusion of the link to any vertex containing a free occurrence of x. In addition, we can

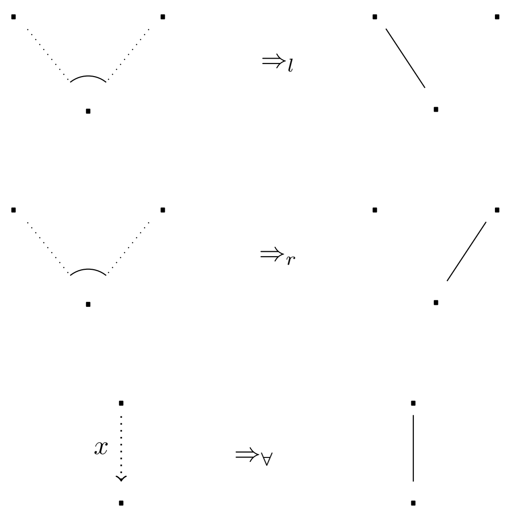


Table 4.3: Switchings for par and universal links. In addition to the $\forall$ switching shown, the $\forall$ link has a switching connecting the conclusion of the link to *any* vertex which a free occurrence of its eigenvariable x.

always connect the conclusion of the link to its premiss — this last case is only needed to correctly derive cases of vacuous quantification such as $p \vdash \forall x.p$.

? proves the following (**?**, Section 3, also give a detailed treatment of first-order proof nets and subnets including all relevant proofs).

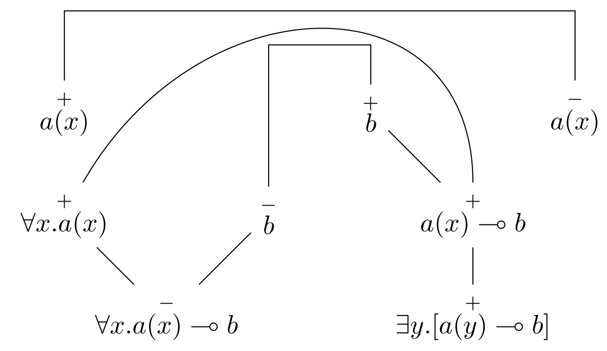
Theorem 4.1 *A first order linear logic proof structure is a proof net iff all its switchings are acyclic and connected.*

The proof structure of Figure 4.1 is not a proof net; it should therefore have a switching with is cyclic and/or disconnected. There are three possible switches for the $\forall$ link and two for the $\parr$ link. Figure 4.2 shows the switching with is cyclic (and disconnected) and therefore establishes the proof structure is not a proof net.

Graph contractions

? presents a contraction criterion for first-order linear logic in the style of **?**.

The contractions for first-order linear logic are shown in Table 4.4. As a first step, we forget about all formulas in the proof structure keeping track only of the free variables at each vertex in the graph. For Figure 4.1 this produces the graph shown on the left hand side of Figure 4.3. Then we progressively shrink the proof structure using the contractions shown in the figure. Each contraction removes an edge (a linked pair of edges in the case of the p contraction) and identifies the two nodes which were connected by



$\overset{+}{a(x)}$
$\overset{+}{b}$
$\overset{-}{a(x)}$
$\overset{+}{\forall x.a(x)}$
$\overset{-}{b}$
$\overset{+}{a(x) \multimap b}$
$\overset{-}{\forall x.a(x) \multimap b}$
$\overset{+}{\exists y.[a(y) \multimap b]}$

Figure 4.2: Switching showing the proof structure of Figure 4.1 is not a proof net.

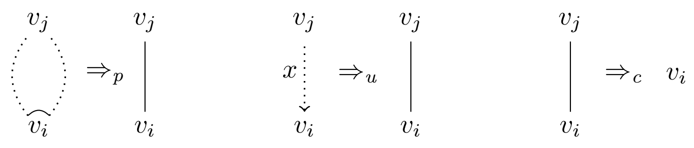


Table 4.4: Contractions for first-order linear logic. Conditions: for the c contraction, $v_i \neq v_j$ and, for the u contraction, all free occurrences of x are at v_j.

it. A c contraction can only be performed on two distinct vertices; that is, we are not allowed to eliminate self-loops. The free variables of the result vertex of the c are the union of the sets of free variables of the two input vertices. The u contraction verifies all occurrences of its eigenvariable x occur at v_j, then removes the eigenvariable from the result (this amounts to verifying the eigenvariable condition for $R\forall/L\exists$). A proof structure is a proof net if and only if it contracts to a single point using the contractions of Table 4.4.

As an example, Figure 4.3 shows the contractions performed on the proof structure of Figure 4.1, with the initial structure on the left and the structure after all c contractions on the right. The arrow and eigenvariable of the $\forall$ link and the connection between the two other dotted links ensure the notation is unambiguous. The displayed graph is not a single vertex but it cannot be further contracted: the universal contraction u cannot apply since the variable x occurs at the bottom vertex instead of only at the right vertex as required for the contraction and the contraction p cannot apply until its two branches end at the same vertex. Since the contractions of Table 4.4 are confluent, any graph which is not further contractible but is not a unique vertex, such as the one shown at the right of Figure 4.3, suffices to show that the given sequent

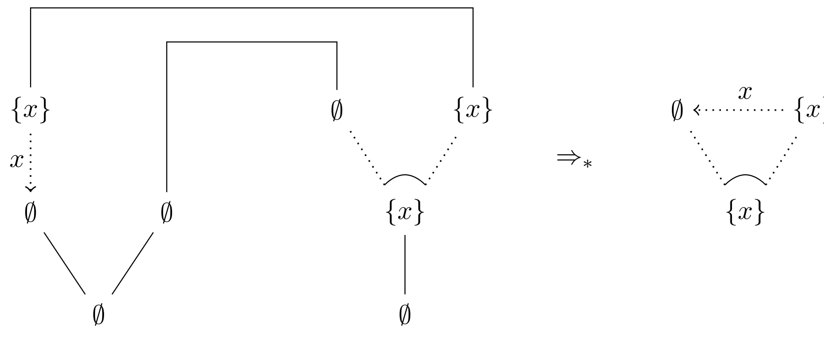


Figure 4.3: Contractions for the proof structure of Figure 4.1

is underivable for the given reading (that is axiom linking).

First-order linear logic for parsing

Summarising, parsing/proof search in first-order linear logic operates as follows.

1. For each word in the sentence, we find a first-order formula in the lexicon.
2. We unfold a sequent using the rules of Table 4.2.
3. We connect atomic formulas of opposite polarity, unifying variables.
4. We contract the resulting proof structure to a single vertex.

Combinatorially, the complex steps are step 1 (in the case of high lexical ambiguity) and step 3 where we connect the atomic formulas. For an actual implementation (for example **?**) it is therefore desirable to contract early — thereby keeping a compact representation of the current state of the proof — and to develop ways of detecting 'doomed' configurations, that is graphs which can never be contracted to a single vertex, no matter how we continue the construction of our proof structure. Examples of such configurations are connections between a node and its ancestor with a path of dotted links (this corresponds to a cycle in the proof structure and though we can validly reduce the size of this cycle, such a configuration will, at best, end up producing a self-loop) or isolated vertices (an isolated vertex is a vertex which is not connected to the rest of the graph but which also doesn't have any unlinked atomic formulas; unless it is the last vertex of the graph, such a vertex corresponds to a disconnected proof structure). More subtle strategies can be used, such as a type of look-ahead (**?**), and partial order constraints (**?**). Combining early failure with a smart backtracking strategy for selecting which atomic formulas to unify (**?**) produces an effective algorithm.

Figure 4.4 shows the first-order linear logic formula unfolding for a slightly more complicated example. It corresponds to the sentence "everyone slept

	$\overset{+}{np(X,1)}$	$\overset{+}{np(Y,Z)}$
$\overset{-}{np(0,1)}$		
$\overset{-}{np(x,Z)}$		

	$\overset{+}{s(V,W)}$	$\overset{+}{s(x,2)}$	$\overset{+}{s(0,3)}$
$\overset{-}{s(V,W)}$			
$\overset{-}{s(X,2)}$			
$\overset{-}{s(Y,3)}$			

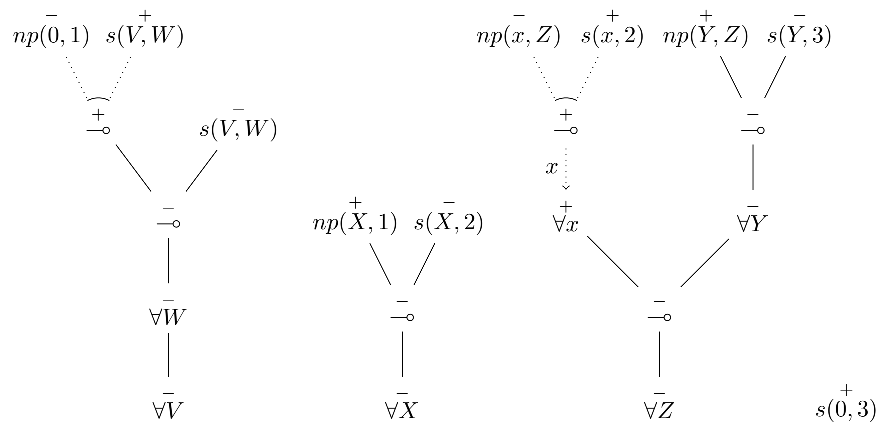


Figure 4.4: Formula unfolding for "everyone slept soundly" in first-order linear logic

soundly" in both the Displacement calculus and hybrid type-logical grammars seen from the point of view of first-order linear logic in a way to be made more precise in Sections 4.4 and 4.5 respectively. The assignments to "slept" (Lambek formula $np\backslash s$ at positions $1, 2$, translated using Equations **??**-4.4) and "soundly" (Lambek formula $(np\backslash s)\backslash(np\backslash s)$ at positions $2, 3$). Finally, the formula for "everyone" at positions $0, 1$ is the one proposed by **?** for quantifiers in first-order linear logic, but it also corresponds to the Displacement calculus formula $(s \uparrow np) \downarrow s$ and to the hybrid grammar lexical entry $\lambda P.(P\ everyone) : (np \multimap s) \multimap s$.

Given a formula unfolding, the search space for a statement consists of enumerating the possible matchings of positive and negative atomic formulas. We can summarise the search space by a number of square matrices, one for each type of atomic formula occurring in the statement. In the current example, we have 2 positive and 2 negative occurrences of np, and 3 positive and 3 negative occurrences of s. If the number of positive and negative occurrences of an atomic formula are not equal, there can never be a proof. The condition that the number of positive and negative formula occurrences for each atom must be equal is generally called the 'count check' (**?**), and it is a very useful test allowing us to quickly (and correctly!) conclude that a statement must be underivable.

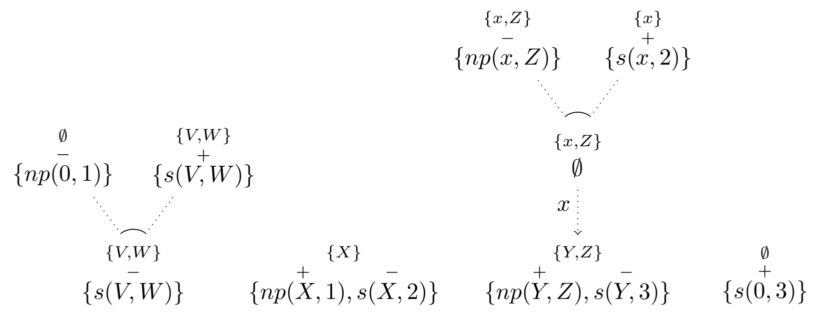


Figure 4.5: Partially contracted version of the proof structure from Figure 4.4

Figure 4.4 show the matrices for np and s at the top of the figure. To obtain a proof structure, we must find a perfect matching between each column (representing the positive occurrences) and each row (representing the negative occurrences). This is the complicated part of proof search. For n positive (or negative) occurrences, there are $n!$ possible matchings. For the example, there are 2 (= 2!) ways of matching the np formulas and 6 (= 3!) ways of matching the s formulas. Therefore, even this simple example already has 12 possible proof structures — the choices for the np formulas and for the s formulas are independent, giving $2 \times 6 = 12$ possibilities — and for more complicated structures, this can become intractable very quickly.

However, in a research context, it is vital for grammar developers to be able to find all derivations which their grammar generates, since different proofs correspond to (at least potentially) different readings predicted by their grammar. If we want to claim that first-order linear logic is a useful formalism for computational linguistics then it is essential for implementations of proof search in first-order linear logic to side-step as much of this combinatorial explosion as is possible.

One standard way to do this is enumerate the different possibilities for each of the atomic formulas (positive or negative) and connect the formula which has the least available possibilities. This amounts to implementing the so-called 'dancing links' strategy of **?** and it has been implemented in several type-logical theorem provers (**?**, **?**). In the context of first-order linear logic, there are a number of simple but effective ways to exclude axiom links which can never lead to a proof net.

First of all, we contract the formula unfolding as much as possible. This allows us to provide a compact representation of the state of the current proof attempt. We need to be careful and keep track, at each node, of the atomic formulas which have not yet been linked and of the free variables which may (at least potentially) be unified with the eigenvariable of a $\forall$ link. Figure 4.5 shows the contracted version of the formula unfolding from Figure 4.4. Only the par and universal links remain.

When we look at the negative $np(x, Z)$ formula and investigate the two

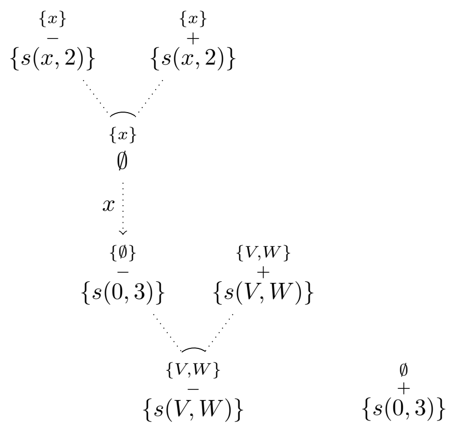


Figure 4.6: The partially contracted version of the proof structure from Figure 4.5 after identification of the np formulas

available positive formulas $np(X,1)$ and $np(Y,Z)$, we can immediately see that identifying $np(x,Z)$ with $np(Y,Z)$, would unify Y with x, and thereby produce a structure which can never be contracted to a single vertex: $Y = x$ means there is an occurrence of x at the 'wrong' side of the universal link (indicated by the arrow) and which would therefore fail the condition of the u contraction — this corresponds to a failure of the variable condition in a sequent calculus proof. Another way to see this connection leads to failure of contraction is that we have connected a node to one of its 'ancestors': we can never contract both the par link and the universal link already connecting $np(x,Z)$ to $np(Y,Z)$ after these nodes are merged, since this would produce a cycle.

Only one of the two possibilities for linking the np formulas can therefore lead to a proof net: the one connecting $np(x,Z)$ to $np(X,1)$, which unifies $X = x$ and $Z = 1$, and connecting $np(0,1)$ to $np(Y,Z)$ (the only remaining possibility after the two other np formulas have been identified), which unifies $Y = 0$ and $Z = 1$ (the previous axiom link already unified Z to 1). Figure 4.6 shows the result of this identification of the np formulas.

Each identification of two atomic formulas merges the two vertices (technically, we add an edge for the axiom link, then contract this edge using the c contraction). For the identification of $np(X,1)$ and $np(x,Z)$, the negative atomic formula $s(X,2)$ (which becomes $s(x,2)$ after unification) remains at the merged node. In general, we take the multiset union of the atomic formulas at the two nodes and remove the two matched atomic formulas. For the other identification, of $np(0,1)$ and $np(Y,Z)$, the negative formula $s(Y,3)$ (which becomes $s(0,3)$ after unification) remains at the merged vertex. The variables Y and Z, having been unified to 0 and 1 respectively, are not longer at risk of unification with the eigenvariable of a $\forall$ link and therefore no longer

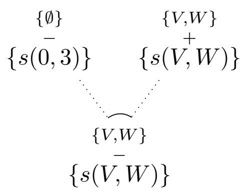


Figure 4.7: The partially contracted version of the proof structure from Figure 4.6 after identification of the $s(x,2)$ formulas

appear in the structure.

What are the remaining possibilities? There are different ways to again obtain a unique matching. First of all, when we look at the positive $s(x,2)$ formula (we arrive at similar conclusions by looking at the negative $s(x,2)$ formula instead), it has three potential axiom links. However, we can exclude two of these: $s(0,3)$ is impossible (since $x \neq 0$ and $2 \neq 3$) and although $s(V,W)$ would unify, it would again produce an x on the 'wrong' side of the $\forall x$ link (and would also be another link to an ancestor, indicating a cycle). Therefore, the only possibility which can lead to a proof net is connecting the positive $s(x,2)$ formula to its negative sister node $s(x,2)$ which produces a par redex and furthermore allows us to contract the $\forall x$ link immediately afterwards. This produces the structure shown in Figure 4.7.

There are again several ways to see there is only one possibility. Firstly, when we connect the positive $s(V,W)$ formula to the negative one, we create a loop and will therefore create a structure to which the par contraction can never apply. Similarly, connecting the positive and negative $s(0,3)$ formulas will also produce a structure to which the par contraction can never apply, this time because the two daughters can never 'join' as required for the par contraction. This means the only possible matching is the positive $s(0,3)$ formula with the negative $s(V,W)$ formula and the negative $s(0,3)$ formula with the positive $s(V,W)$ formula. These two identifications unify V with 0 and W with 3, and in addition produce a par redex. We have now connected all atomic formulas and, after the final par redex, contracted the structure to a single vertex, thereby showing our initial structure is a proof net. Moreover, at least in this simple case, we have avoided doing any axiom links which are not part of the final proof net. When we assign only a polynomial amount of time to each axiom link, it is of course *extremely* unlikely that we can find a procedure which guarantees we only perform axiom links which are part of a proof net, since this would provide a polynomial solution to a known NP-complete problem.

Although the early failure strategies used are rather simple, they work well even for much more complicated examples (**?**, **?**). They are also easy to implement and have been integrated in a number of theorem provers for type-logical grammars (**?**, **?**, **?**)

Resolution and proof nets In a sense, proof search for linear logic using contracted structures like we do here is resolution-like[1]: when we restrict to the $\otimes$, $\exists$ fragment — that is negative $\multimap$, $\forall$ and positive $\otimes$, $\exists$ — then negative formulas are Horn clauses and positive formulas are goals and all such formulas contract to a single vertex. Each connection of atomic formulas then corresponds to a resolution step, combining two vertices (clauses or goals) into a single one. Seen from this perspective, the par and universal links are extensions of (linear, Horn clause) resolution.

4.4 The Displacement calculus

The Displacement calculus extends the Lambek calculus by changing the structures of the logic from strings to string tuples. This idea translates to first-order linear logic as a move from formulas with pairs of position variables to formulas with $2k$ position variables (**?**, **?**). So a string segment a is represented by two position variables x_0, x_1 representing a string with x_0 as its leftmost position and x_1 as its rightmost position, and a string segment $a_1 + \mathbf{1} + a_2$ is represented by four position variables x_0, x_1, x_2, x_3, with x_0, x_1 representing a_1, x_2, x_3 representing a_2 and x_1, x_2 the insertion point '$\mathbf{1}$'. The technical details of the translation in its general form (**?**, Equations 1 to 15) are a bit complicated because of the many position variables involved[2]. However, we can already translate nearly all of the phenomena of **?** using just the simple instantiations of the general translation shown below. Given that we only consider formulas with at most one insertion point, there is no need to distinguish between the multiple versions of the discontinuous connectives, and we therefore drop the indices, writing simply $\uparrow$ instead of $\uparrow_k$, etc.

$$\|C/B\|^{x_0,x_1} = \forall x_2.[\|B\|^{x_1,x_2} \multimap \|C\|^{x_0,x_2}] \tag{4.5}$$
$$\|A\backslash C\|^{x_1,x_2} = \forall x_0.[\|A\|^{x_0,x_1} \multimap \|C\|^{x_0,x_2}] \tag{4.6}$$
$$\|C \uparrow B\|^{x_0,x_1,x_2,x_3} = \|B\|^{x_1,x_2} \multimap \|C\|^{x_0,x_3} \tag{4.7}$$
$$\|A \downarrow C\|^{x_1,x_2} = \forall x_0, x_3.[\|A\|^{x_0,x_1,x_2,x_3} \multimap \|C\|^{x_0,x_3}] \tag{4.8}$$
$$\|C/_b B\|^{x_0,x_1,x_2,x_3} = \forall x_4.[\|B\|^{x_3,x_4} \multimap \|C\|^{x_0,x_1,x_2,x_4}] \tag{4.9}$$
$$\|A\backslash_b C\|^{x_3,x_4} = \forall x_0, x_1, x_2.[\|A\|^{x_0,x_1,x_2,x_3} \multimap \|C\|^{x_0,x_1,x_2,x_4}] \tag{4.10}$$
$$\|{}^\wedge A\|^{x_0,x_2} = \exists x_1.\|A\|^{x_0,x_1,x_1,x_2} \tag{4.11}$$
$$\|\triangleright^{-1} A\|^{x_1,x_2} = \forall x_0.\|A\|^{x_0,x_0,x_1,x_2} \tag{4.12}$$

For ease of reference, Equations 4.5 and 4.6 just repeat the standard translation of the Lambek calculus implications into first-order linear logic. Figure 4.8 shows, at the top of the figure, the different string segments involved. Both connectives correspond to 'concatenating' a segment x_0, x_1 to a segment x_1, x_2.

[1]Technically, resolution moves all formulas to the left hand side of the turnstile and tries to find refutations, whereas proof net proof search moves all formulas to the right hand side of the turnstile and tries to find proofs.

[2]We need an additional mechanism to enforce linear order constraints to correctly embed the $\uparrow I$ rule with 6 or more string positions as well (**?**).

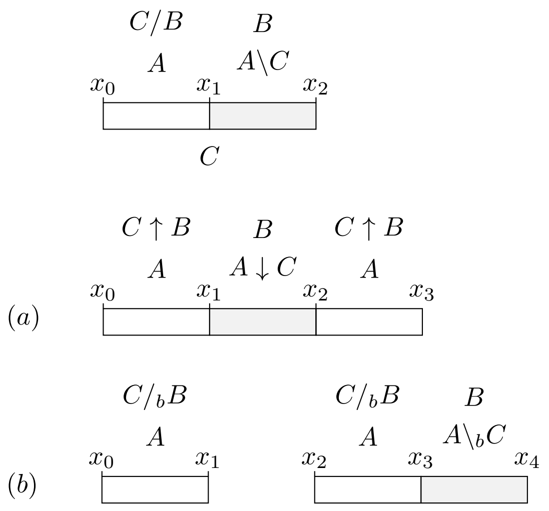


Figure 4.8: String positions for some Displacement calculus connectives

The translations for $\uparrow$ (4.7) and $\downarrow$ (4.8) are the simplest versions of the discontinuous connectives. As indicated by the (a) case in the middle of Figure 4.8, the formula $C \uparrow B$ corresponds to a *pair* of strings $a_1 + \mathbf{1} + a_2$ (and therefore to four position variables), the first segment a_1 at positions x_0, x_1 and the second segment a_2 at x_2, x_3. Its B argument 'fills the hole' x_1, x_2 (that is, the insertion point) between the two segments, resulting in a formula C at positions x_0, x_3 (which corresponds to $a_1 + b + a_2$ as desired). This implements the intuition of $C \uparrow B$ as a C formula which is incomplete for a B formula (while keeping track of its position).

Similarly, the argument A of formula $A \downarrow C$ corresponds to $a_1 + \mathbf{1} + a_2$ and therefore to two string segments x_0, x_1 and x_2, x_3 (with the starting position x_0 and ending position x_3), with the argument A circumfixing itself around the positions x_1, x_2 of the formula $A \downarrow C$ (segment b) to produce a result formula C at x_0, x_3 corresponding to $a_1 + b + a_2$. The intuition is that the argument A appears 'around' the formula $A \downarrow C$, connected at both the left and right positions of $A \downarrow C$.

The $/_b$ and $\backslash_b$ translation of Equations 4.9 and 4.10, which corresponds to the (b) case at the bottom of Figure 4.8 are generalisations of the Lambek calculus implication where A and $C/_bB$ represent a pair of string (and therefore require four position arguments). As we can see for the (b) cases at the bottom of Figure 4.8, when we forget about the initial segment x_0, x_1, which is passed along unchanged, we just have the standard Lambek calculus implications.

Finally, the bridge connective $^\wedge$ (used for extraction) and the right projection connective $\triangleright^{-1}$ (used for Dutch verb clusters) operate by adding the empty string as a segment, with $\triangleright^{-1}$ adding an empty segment as the first segment (and quantifying universally) and $^\wedge$ adding it as the middle segment (and quantifying existentially).

Example: quantifiers We can combine 4.7 and 4.8 to translate the generalised quantifier formula $(s \uparrow np) \downarrow s$ (assuming input positions $1, 2$) as shown below.

$$\begin{aligned}
&\|(s \uparrow np) \downarrow s\|^{1,2} \\
\forall x_0, x_3.&\|s \uparrow np\|^{x_0,1,2,x_3} \multimap \|s\|^{x_0,x_3} \\
\forall x_0, x_3.&[\|np\|^{1,2} \multimap \|s\|^{x_0,x_3}] \multimap \|s\|^{x_0,x_3} \\
\forall x_0, x_3.&[np(1,2) \multimap s(x_0,x_3)] \multimap s(x_0,x_3)
\end{aligned}$$

This captures the intended meaning of the quantifier formula: a quantifier occurring at positions $1, 2$ takes as its argument a sentence s (at any positions x_0, x_3) missing a noun phrase at positions $1, 2$ (that is, the position of the quantifier formula itself) to produce a sentence at the positions x_0, x_3 of the argument sentence. This assignment solves many of the quantifier scope problems of the Lambek calculus discussed in Section 1.5.2. Incidentally, this formula is identical to the one proposed by **?** for generalised quantifiers in first-order linear logic.

Example: ellipsis As a more complicated example, will give a translation of the lexical entry for "did" — as used in sentences like "John left before Mary did" — in the Displacement calculus. **?** propose we assign it the following formula (where $vp = np\backslash s$).

$$((vp \uparrow vp)/vp)\backslash(vp \uparrow vp)$$

Given that $vp \uparrow vp$ represents a pair of strings (it represents a verb phrase vp with a vp gap somewhere inside it), we need to use the b versions of the Lambek calculus slashes, and translate this formula as $((vp \uparrow vp)/_b vp)\backslash_b(vp \uparrow vp)$. We can then translate this formula into first-order linear logic as follows.

$$\begin{aligned}
&\|((vp \uparrow vp)/_b vp)\backslash_b(vp \uparrow vp)\|^{4,5} \\
\forall F, I, J.&\|(vp \uparrow vp)/_b vp\|^{F,I,J,4} \multimap \|vp \uparrow vp\|^{F,I,J,5} \\
\forall F, I, J.&[\forall x_1.\|vp\|^{4,x_1} \multimap \|vp \uparrow vp\|^{F,I,J,x_1}] \multimap \|vp\|^{I,J} \multimap \|vp\|^{F,5} \\
\forall F, I, J.&[\forall x_1.\|vp\|^{4,x_1} \multimap \|vp\|^{I,J} \multimap \|vp\|^{F,x_1}] \multimap \|vp\|^{I,J} \multimap \|vp\|^{F,5}
\end{aligned}$$

We have left the final $vp = np\backslash s$ subformulas untranslated. We can see that except for some fairly complicated manipulations with string positions, the formula simply indicates it selects a function of two vp's into a single vp to become a vp modifier.

Figure 4.9 shows the formula unfolding for "John left before Mary did" using the formula we just computed. The proof has been slightly simplified by removing the negative universal and positive existential quantifiers and by not unfolding the vp formulas where doing so would simplify the proof (not unfolding the vp formulas hides a non-trivial part of the combinatorics of proof search for this example). The square tables above the figure represent the (somewhat simplified) combinatorics of proof search for this example.

However, a number of axiom links are easily seen as forced: $np(0,1)$ cannot unify with $np(X',4)$ which forces the axioms $np(3,4) - np(X',4)$ and $np(0,1) - np(Y',X)$. Given $Y' = 0$ (as one of the unifications of the np axiom links), $s(Y',5)$ become $s(0,5)$ which no longer unifies with $s(3,W)$, thereby forcing $s(Y',5) - s(0,5)$ and $s(X',x) - s(3,W)$ for the s axioms. Given $W = x$ (by the unifications of the s axioms), there is now only one possibility for the negative $vp(V,W)$, which becomes $vp(V,x)$ given $W = x$: $vp(V,2)$ is excluded because of unification failure $x \neq 2$, but the positive $vp(Y,Z)$ would violate the quantifier scope condition for x, because the eigenvariable x would appear on the wrong side of the $\forall x$ link. This forces the axiom $vp(V,W) - vp(X,x)$. For the final two vp connections, we only need to realise that connecting positive $vp(Y,Z)$ to negative $vp(Y,Z)$ produces a cycle (since they are already connected by a switching of the module). We therefore complete the axiom links by connecting negative $vp(1,2)$ to positive $vp(Y,Z)$, and negative $vp(Y,Z)$ to positive $vp(V,2)$.

Figure 4.10 shows the complete proof net after all axioms links have been performed (we should verify the correctness condition is satisfied, either verifying all switching are acyclic and connected, or contracting the structure to a single vertex, but this is easily done).

Example: gapping As a final example, **?** propose the formula $((s \uparrow tv)\backslash(s \uparrow tv))/{}^\wedge(s \uparrow tv)$ for "and" as it occurs in gapping constructions like "John studies logic and Charles, phonetics" (where tv abbreviates $(np\backslash s)/np$).

$$\|((s \uparrow tv)\backslash_b(s \uparrow tv))/{}^\wedge(s \uparrow tv)\|^{3,4}$$
$$\forall B.[\|{}^\wedge(s/_a tv)\|^{4,B} \multimap \|(s/_a tv)\backslash_b(s/_a tv)\|^{3,B}]$$
$$\forall B.\exists A.\|s/_a tv\|^{4,A,A,B} \multimap \forall E,C,D.[\|s/_a tv\|^{E,C,D,3} \multimap \|s/_a tv\|^{E,C,D,B}]$$
$$\forall B.\exists A.[\|tv\|^{A,A} \multimap \|s\|^{4,B}] \multimap \forall E,C,D.[\|tv\|^{C,D} \multimap \|s\|^{E,3}] \multimap \|tv\|^{C,D} \multimap \|s\|^{E,B}$$

Proving correctness of the translation In order the prove the translation correct, we need to show that the position variables in the first-order linear logic proof reflect the operations on string tuples for each of the rules in the Displacement calculus. In addition, we need to show that we preserve a unique linear order for the position variables. From first-order linear logic back to the Displacement calculus, the key property that makes the proof work is that the translation produces combinations $\forall/\multimap$ or $\exists/\otimes$, and these combinations can be treated as abbreviated proof rules. In other words, the rules of the Displacement calculus are derived rules under the translation into first-order linear logic, and from the point of view of first-order linear logic, when formulas have been translated from the Displacement calculus, we can obtain all proofs using only these derived rules. For example, for the $\downarrow$ case

	$np(X',4)$	$np(Y',X)$
$np(0,1)$		
$np(3,4)$		

	$s(3,W)$	$s(0,5)$
$s(X',x)$		
$s(Y',5)$		

	$vp(V,2)$	$vp(X,x)$	$vp(Y,Z)$
$vp(1,2)$			
$vp(V,W)$			
$vp(Y,Z)$			

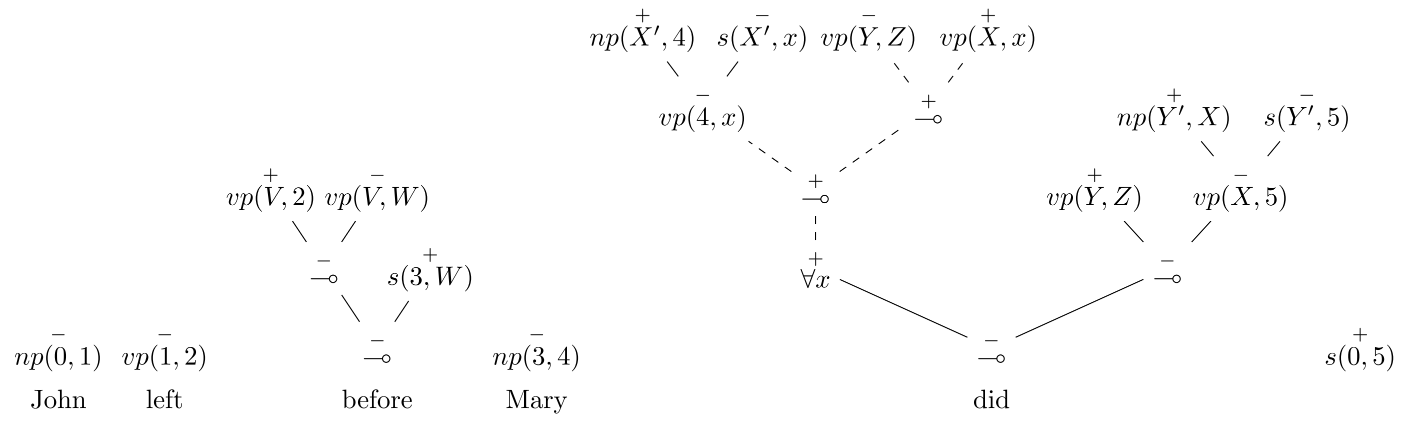


Figure 4.9: Formula unfolding and possible axiom links for “John left before Mary did” translated into first-order linear logic

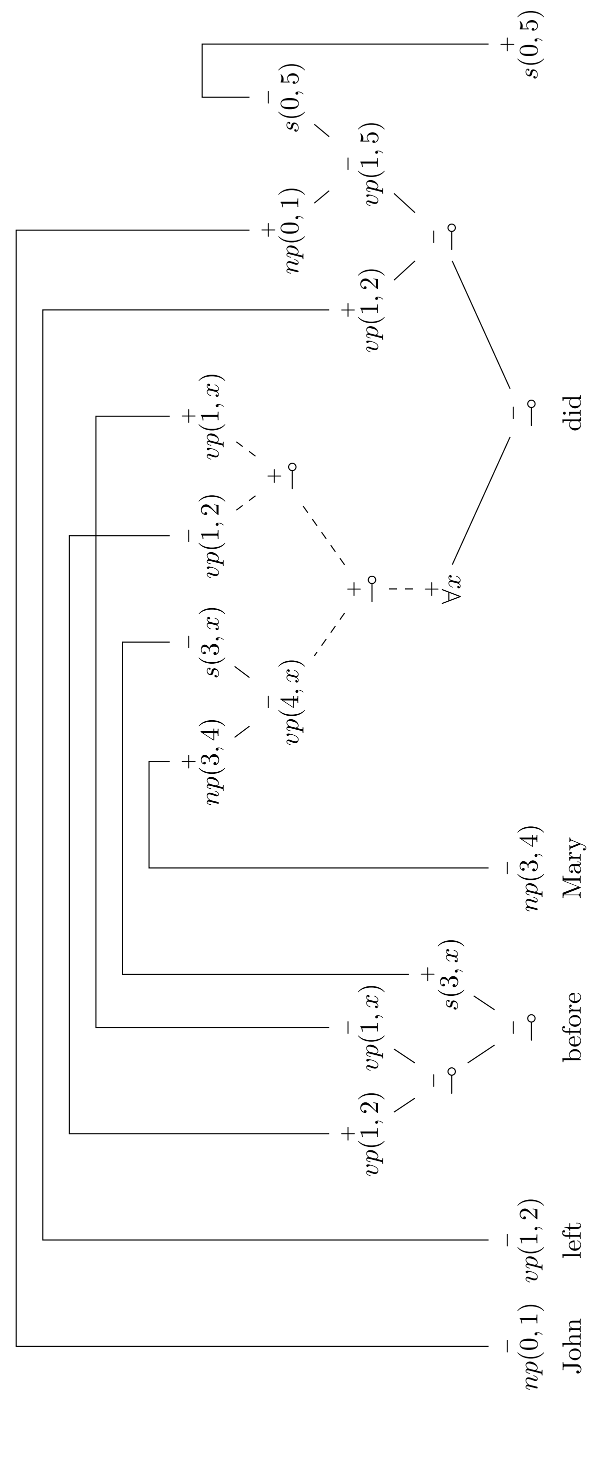


Figure 4.10: Proof net for “John left before Mary did”

shown below, we can, without loss of generality, restrict the proofs in first-order linear logic to those where the two $\forall I$ and the $\multimap I$ rules occur together as shown below[3].

Given $b = (x_1, x_2)$, $a_1 = (x_0, x_1)$, $a_2 = (x_2, x_3)$, and therefore $a_1 + \mathbf{1} + a_2 = (x_0, x_1, x_2, x_3)$ and $a_1 + b + a_2 = (x_0, x_3)$, the cases for $\uparrow$ and $\downarrow$ look as follows (**?** provides complete proofs).

$$\dfrac{\begin{array}{c} b : B \\ \vdots \\ a_1 + b + a_2 : C \end{array}}{a_1 + \mathbf{1} + a_2 : C \uparrow B} \uparrow I \qquad \dfrac{\dfrac{\begin{array}{c} \|B\|^{x_1,x_2} \\ \vdots \\ \|C\|^{x_0,x_3} \end{array}}{\|B\|^{x_1,x_2} \multimap \|C\|^{x_0,x_3}} \multimap I}{\|C \uparrow B\|^{x_0,x_1,x_2,x_3}} \equiv_{def} \qquad \textit{units}$$

$$\dfrac{\begin{array}{c} a_1 + \mathbf{1} + a_2 : A \\ \vdots \\ a_1 + b + a_2 : C \end{array}}{b : A \downarrow C} \downarrow I \qquad \dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} \|A\|^{x_0,x_1,x_2,x_3} \\ \vdots \\ \|C\|^{x_0,x_3} \end{array}}{\|A\|^{x_0,x_1,x_2,x_3} \multimap \|C\|^{x_0,x_3}} \multimap I}{\forall x_3.[\|A\|^{x_0,x_1,x_2,x_3} \multimap \|C\|^{x_0,x_3}]} \forall I}{\forall x_0 \forall x_3.[\|A\|^{x_0,x_1,x_2,x_3} \multimap \|C\|^{x_0,x_3}]} \forall I}{\|A \downarrow C\|^{x_1,x_2}} \equiv_{def} \qquad \textit{units}$$

4.5 Hybrid type-logical grammars

Hybrid type-logical grammars are a combination of the Lambek calculus with lambda grammars/abstract categorial grammars (ACGs). Standard results in the ACG literature show that we can translate abstract categorial grammars into first-order linear logic (**?**, **?**, **?**, **?**) using principal types (**?**). From the point of view of lambda grammars, we can see hybrid type-logical grammars as an extension which allows us to replace atomic formulas of the string type by Lambek calculus formulas.

Example: quantifiers An HTLG/lambda grammar quantifier like "every" is assigned the formula $n \multimap (np \multimap s) \multimap s$ and the prosodic term $\lambda N^s \lambda P^{s \to s}.P(every + N)$. The type s denotes the type of string and the infix operator '+' denotes string concatenation.

It is convenient to replace the basic type s (string) by $\sigma \to \sigma$ (a *function* from string positions to string positions, more specifically from the end position to the starting position). This has the advantage that we can remove explicit references to string concatenation: '+' is definable as function

[3]This is essentially the insight of **?**. For the $\forall/\multimap$ cases in natural deduction this property is guaranteed by using long normal form proofs, although this is not as obvious for the $\exists/\otimes$ cases. For proof nets, we simply use the fact that we can remove multiple par/forall links in one step, and for the tensor/exists case we use a variant of the 'splitting tensor' property.

composition (but only for terms of type $\sigma \rightarrow \sigma$) $\lambda Q^{\sigma\rightarrow\sigma} \lambda P^{\sigma\rightarrow\sigma} \lambda z^{\sigma}.(P\,(Q\,z))$. Similarly, the empty string can be defined as $\lambda z.z$ (that is, the string where the start position and end position are identical).

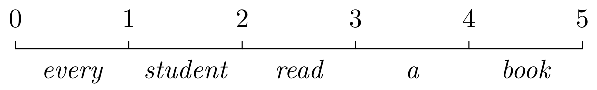
With this in mind, the prosodic term for "every" becomes

$$\lambda N^{\sigma\rightarrow\sigma} \lambda P^{(\sigma\rightarrow\sigma)\rightarrow\sigma\rightarrow\sigma} \lambda z^{\sigma}.((P\,\lambda v.(\mathit{every}\,(N\,v)))\,z)$$

The subterm $\lambda v.(\mathit{every}\,(N\,v))$ is simply the translation of $\mathit{every} + N$, and z is the position corresponding to the end of the string.

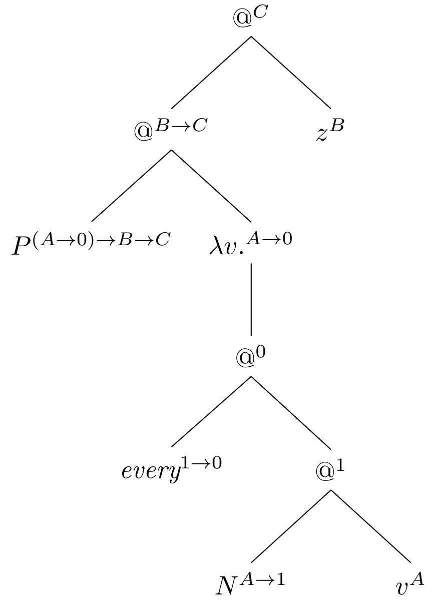
Now, it is easy to verify this term is well-typed, and that its Church type is $(\sigma \rightarrow \sigma) \rightarrow ((\sigma \rightarrow \sigma) \rightarrow \sigma \rightarrow \sigma) \rightarrow \sigma \rightarrow \sigma$. However, we want to compute its *principal type* (?), that is, a type from which we can uniquely reconstruct the term (up to the standard lambda calculus equivalences $\alpha\beta\eta$). Intuitively, this amounts to replacing the atomic σ subtypes which distinct type variables as much as possible. In practice this means we start by assuming that all atomic subtypes are different and then compute the most general unifier for each application step. This ensures that the application is well-typed and that the least amount of variables are identified.

Finally, we need some way of encoding the positions of the initial string. One of the standard ways of doing this in the parsing literature is to assign an n word sentences position variables from 0 to n, with word n appearing between positions $n-1$ on the left and n on the right. For a sentence like "every student read a book" this would look as follows.



Translating this idea back to types, given that "every" occurs between string positions 0 and 1, the type of its constant is $1 \rightarrow 0$ (conforming to the convention that a string is given a functional type from its rightmost position 1 to its leftmost position 0).

We now have everything in place to compute the principal type. When we forget, at least for the moment, the initial lambda abstractions, the tree of the term $((P\,\lambda v.(\mathit{every}\,(N\,v)))\,z)$ looks as follows, where '@' denotes application.



Given that we have assigned "every" the type $1 \to 0$, the type of $(N\,v)$ must be 1 and therefore N must is of type $A \to 1$ and v of type A (for a new type A). The type of $(every\,(N\,v))$ is then 0, and abstraction over v (with type A) gives $A \to 0$. Completing the principal type calculations produces the following type judgment, given *every* of type $1 \to 0$.

$$every^{1\to 0} \vdash \lambda N^{A\to 1} \lambda P^{(A\to 0)\to B\to C}.\lambda z^{B}.((P\,\lambda v^{A}.(every^{1\to 0}\,(N\,v)))\,z)$$

Looking only at the types leaves us with the following.

$$1 \to 0 \vdash (A \to 1) \to ((A \to 0) \to B \to C) \to B \to C$$

This statement has only one (η long, β normal) proof, namely the following.

$$\cfrac{\cfrac{\cfrac{\cfrac{[B]^2 \qquad \cfrac{\cfrac{\cfrac{1\to 0 \qquad \cfrac{[A]^1 \quad [A\to 1]^4}{1}}{0}\,\to E}{A\to 0}\,\to I^1 \qquad [(A\to 0)\to B\to C]^3}{B\to C}\,\to E}{C}\,\to E}{B\to C}\,\to I^2}{((A\to 0)\to B\to C)\to B\to C}\,\to I^3}{(A\to 1)\to((A\to 0)\to B\to C)\to B\to C}\,\to I^4$$

It is easy to show that if this type (seen as a formula) has a proof, it must be unique, since each atomic formula has one positive and one negative occurrence[4], which 'forces' a matching (**?**, gives a proof of this, but it is also easy to see using proof nets: each atomic formula has one positive and one negative occurrence and there is therefore only one possible proof structure).

Given the proof shown above, using the Curry-Howard isomorphism, we can read back the lambda term we started with (choosing "*every*" as the variable for the hypothesis of type $1 \to 0$, z as the variable of type B, N as the variable of type $A \to 1$ and P as the variable of type $(A \to 0) \to B \to C$ even gives us back the identical term).

To translate this to first-order linear logic, we simply combine the linear formula $n \multimap (np \multimap s) \multimap s$ using the principal type as arguments of the corresponding atomic formulas as follows.

$$1 \to 0 \vdash (A \to 1) \to ((A \to 0) \to B \to C) \to B \to C$$
$$\forall A \forall B \forall C.\ [n(1,A) \multimap [np(0,A) \multimap s(C,B)] \multimap s(C,B)]$$

For ease of reading, we have reversed the order of the $R \to L$ subtypes so they become arguments in left-to-right order; for example, the type $A \to 1$ of atomic formula n becomes the predicate $n(1,A)$.

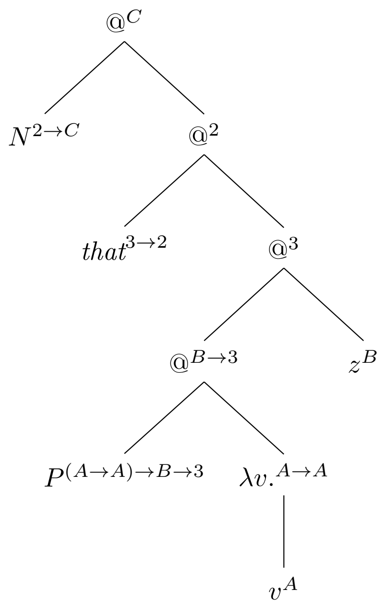
Example: relativisers A word like "that" is assigned the formula $(np \multimap s) \multimap n \multimap n$ and prosodic term $\lambda P^{s \to s} \lambda N^{s}.N + \mathit{that} + (P\,\epsilon)$, where ϵ denotes the empty string.

Translating this to a version with string positions produces the following term.

$$\lambda P^{(\sigma \to \sigma) \to \sigma \to \sigma} \lambda N^{\sigma \to \sigma} \lambda z^{\sigma}.(N\,(\mathit{that}\,((P\,\lambda v.v)\,z)))$$

Assuming that "that" occurs between positions 2 and 3 (which gives it the type $3 \to 2$), we compute the principal type of this term as follows (to save space, we have again removed the initial abstractions).

[4]In a statement $A_1, \ldots, A_n \vdash C$, the formulas A_i are negative and C is positive. For complex formulas or types, if $A \multimap B$ (or $A \to B$) is positive, then A is negative and B is positive, whereas if $A \multimap B$ (or $A \to B$) is negative, then A is positive and B is negative. It is easy to show that negative formulas end up at the left hand side of the turnstile '$\vdash$' of a sequent proof, whereas positive formulas end up at the right hand side.



This gives the following principal type and corresponding formula in first-order linear logic.

$$3\to 2 \vdash ((A\to A)\to B\to 3)\to (2\to C)\to (B\to C)$$
$$\forall A\forall B\forall C.\ [np(A,A)\multimap s(3,B)]\multimap [n(C,2)\multimap n(C,B)]$$

Example: gapping Up until now, we have only used the expressivity of lambda grammars without using any of the Lambek calculus formulas allowed in hybrid type-logical grammars. However, from the point of view of first-order linear logic, this is a relatively simple extensions, and translating hybrid type-logical grammars into first-order linear logic amounts to 'composing' the translation of lambda grammars into first-order linear logic with the one of Lambek grammars into first-order linear logic (**?**). The gapping analysis of **?** assigns a word like "and" in a sentence like "John studies logic and Charles phonetics" the following syntactic type (leaving *tv*, for transitive verb, unanalysed for the moment).

$$(tv\multimap s)\multimap (tv\multimap s)\multimap tv\multimap s$$

The prosodic term for "and" is the following.

$$\lambda Q^{s\to s}\lambda P^{s\to s}\lambda TV^{s}.(P\ TV)+ and + (Q\,\epsilon)$$

And this corresponds to the following 'string position' term.

$$\lambda Q^{(\sigma\to\sigma)\to\sigma\to\sigma}\lambda P^{(\sigma\to\sigma)\to\sigma\to\sigma}\lambda TV^{\sigma\to\sigma}\lambda z^{\sigma}.((P\ TV)\,(and\ ((Q\,\lambda y.y)\,z)))$$

We compute the following principal type for "*and*" at string segment $3, 4$.

$$4 \to 3 \vdash ((A \to A) \to B \to 4) \to ((D \to C) \to 3 \to E) \to (D \to C) \to B \to E$$
$$\forall A \forall B \forall C \forall D \forall E.\ [tv(A, A) \multimap s(4, B)] \multimap [tv(C, D) \multimap s(E, 3)] \multimap tv(C, D) \multimap s(E, B)$$

To complete the translation, we simply use $tv = (np\backslash s)/np$ and the translation of Lambek calculus formulas into first-order linear logic. We can then simply use the translation of Lambek calculus formulas into first-order linear logic of **?** to obtain a formula in first-order linear logic. For the formula above, this amounts to replacing the formulas $tv(V, W)$ as follows.

$$\begin{aligned}
\|tv\|^{V,W} &= \|(np\backslash s)/np\|^{V,W} \\
&= \forall Z.\|np\|^{W,Z} \multimap \|np\backslash s\|^{V,Z} \\
&= \forall Z.\|np\|^{W,Z} \multimap \forall X.\|np\|^{X,V} \multimap \|s\|^{X,Z} \\
&= \forall Z.[np(W, Z) \multimap \forall X.[np(X, V) \multimap s(X, Z)]]
\end{aligned}$$

What is surprising about the resulting formula in first-order linear logic is that it is logically equivalent to the lexical entry for gapping of **?** when we translate this entry into first-order linear logic as well (we have seen this translation in Section 4.4).

4.6 Comparing formalisms

One important advantage of translating lexical entries in different formalisms into first-order linear logic is that it makes it very easy to *compare* the lexical entries in different formalisms.

As a simple example, the lambda grammar/ACG lexical entry $np \multimap s$ with prosodic term $\lambda y.(y + \mathit{sleeps})$ becomes, according to the translation into first-order linear logic, the formula $\forall x.np(x, 1) \multimap s(x, 2)$, just like the Lambek calculus (and Displacement calculus) formula $np \setminus s$. Even though these translations follow rather different paths, they end up at the same destination, and it is this agreement on many of the 'basic' lexical entries which allows the comparison of formalisms using first-order linear logic. The same is true for other lexical entries, for example, those for generalised quantifiers (**?**, **?**, **?**, **?**) where upon translation into first-order linear logic, we end up with the same (or at least equivalent) formulas.

4.6.1 Relative pronouns

As a more interesting example of this way of comparing formulas, here are five different first-order linear logic formulas expressing extraction. These formulas would be assigned to a relativiser such as “which” occurring at position 3-4.

$$\forall x_0.[[\forall x_1.[np(x_1,x_1)] \multimap s(4,x_0)] \multimap \forall x_2.[n(x_2,3) \multimap n(x_2,x_0)]] \quad (4.13)$$

$$\forall x_0.[\exists x_1.[np(x_1,x_1) \multimap s(4,x_0)] \multimap \forall x_2.[n(x_2,3) \multimap n(x_2,x_0)]] \quad \text{D} \quad (4.14)$$

$$\forall x_0 \forall x_1 \forall x_2.[((np(x_1,x_1) \multimap s(4,x_0)) \multimap (n(x_2,3) \multimap n(x_2,x_0))] \quad \text{ACG} \quad (4.15)$$

$$\forall x_0.[\forall x_1.[np(x_1,4) \multimap s(x_1,x_0)] \multimap \forall x_2.[n(x_2,3) \multimap n(x_2,x_0)]] \quad \text{L} : (n\backslash n)/(np\backslash s) \quad (4.16)$$

$$\forall x_0.[\forall x_1.[np(x_0,x_1) \multimap s(4,x_1)] \multimap \forall x_2.[n(x_2,3) \multimap n(x_2,x_0)]] \quad \text{L} : (n\backslash n)/(s/np) \quad (4.17)$$

The first three formulas, though with slightly different scopes for the quantifiers, intuitively mean that a relative pronoun spanning positions 3-4 is looking to its left for a noun n (a noun spanning positions x_2-3 for some x_2 of our choice) and to its right for a sentence s, which itself is missing a noun phrase anywhere (where this sentence spans positions 4-x_0 for some x_0 of our choice, the relation between the position of the np and this sentence is not specified, though the proof theory will ensure this np will occur ‘inside’ the sentence). The result will be a noun from position x_2, the start of the n argument, to x_0, the end of the s argument.

The first formula is a possibility which I have not seen before. Formula 4.14 is the formula from **?** as well as the translation into first-order linear logic of the extraction formula for the Displacement calculus (D). Formula 4.15 is the translation of the lambda grammar lexical entry proposed by **?**. Finally, formulas 4.16 and 4.17 are the translations of the two Lambek calculus formulas for peripheral extraction. These formulas are related as follows (where a directed path between the two formulas denotes derivability of the target from the source).

$$\begin{tikzcd} & & & 4.16 \\ 4.13 \arrow[r] & 4.14 \arrow[r, shift left] & 4.15 \arrow[l, shift left] \arrow[ur] \arrow[dr] & \\ & & & 4.17 \end{tikzcd}$$

So the formulas of the Displacement calculus and the one proposed directly for first-order linear logic are identical (formula 4.14). This formula is equivalent to formula 4.15 proposed for λ-grammars/ACGs; although we cannot always transform a linear logic formula into an equivalent prenex normal form (**?**), formula 4.14 does allow such a form which is formula 4.15. When we look at lambda grammars in isolation, we cannot even directly ask the question about the relation to Lambek calculus formulas, though here it is clear that formulas 4.13 to 4.15 all have the Lambek calculus formulas 4.16 and 4.17 as special cases. The new formula 4.13 is the most general formula, but it is unclear whether or not there is any useful (or harmful!) difference in behaviour between this formula and the formulas corresponding to those use in the Displacement calculus and lambda grammars[5]

This brings up an important question: since, in (classical) first-order logic, formula 4.13 is equivalent to formulas 4.14 and 4.15 maybe first-order *linear* logic is too fine-grained a tool and the suitable notions of equivalence are better formulated directly in first-order logic. Is the difference between classical first-order equivalence and linear first-order equivalence important, and if, so which is the more suitable notion in the current context?

4.6.2 Adverbs, higher-order formulas and lambda grammars

The first-order linear logic perspective also clarifies the limitations of abstract categorial grammars/lambda grammars. For adverbs, for example, we are looking for a lexical entry which functions at least as well as the Lambek calculus formula $(np \setminus s) / (np \setminus s)$. However, as shown by **?**, we can simply enumerate all possible ACG lexical entries l, compute their translation into first-order linear logic, compute the translation of $(np \setminus s) / (np \setminus s)$ into first-order linear logic and compare. Keeping only the plausible lexical entries (that is, those which generate the right semantics and right word order) leaves us with three possibilities, which are shown as items 4.19 to 4.21 below, together with the translation of $(np \setminus s) / (np \setminus s)$ as item 4.18 (note the narrow scope of $\forall x_1$ in this translation). The adverb is assumed to span positions 1-2.

$$\forall x_0 \forall x_2.[\forall x_1.[np(x_1, 2) \multimap s(x_1, x_2)] \multimap (np(x_0, 1) \multimap s(x_0, x_2))] \tag{4.18}$$
$$\forall x_0 \forall x_1 \forall x_2.[(np(x_1, x_1) \multimap s(2, x_2)) \multimap (np(x_0, 1) \multimap s(x_0, x_2))] \tag{4.19}$$
$$\forall x_0 \forall x_1 \forall x_2.[(np(1, 2) \multimap s(x_1, x_2)) \multimap (np(x_0, x_1) \multimap s(x_0, x_2))] \tag{4.20}$$
$$\forall x_0 \forall x_1 \forall x_2.[(np(x_1, 2) \multimap s(x_0, x_2)) \multimap (np(x_1, 1) \multimap s(x_0, x_2))] \tag{4.21}$$

The translations of ACG lexical entries are always formulas with only universal quantifiers and in prenex normal form[6]. It is easy to verify that all of items 4.19 to 4.21 are strictly more general than the translation of $(np \setminus s) / (np \setminus s)$, shown as item 4.18.

However, where in the case of relative pronouns, a more general formula turned out to be a benefit, in the case of adverbs, it turns out to be a source of overgeneration. For example, item 4.19, the adverb lexical entry most commonly used in the ACG literature, predicts that an adverb selects to its right, a sentence missing a noun phrase *anywhere*. In other words, the lexical entry for adverbs is modelled after the lexical entry for relative pronouns and

[5] An advantage of formula 4.13 is that it is the only one we can extend for so-called parasitic gapping (where a single relative pronoun binds multiple extracted noun phrases) since copying the $\forall x_1.[np(x_1, x_1)]$ subformula allows us to instantiate x_1 to different positions (one for each extracted np), something impossible for formulas 4.14 and 4.15.

[6] The term Skolem normal form is often used for a prenex normal form with universal quantifiers, but I don't use it here since it suggests that existential quantifiers have been replaced by universally quantified Skolem terms and 1) there are no terms in the translations of ACG formulas 2) Skolemization is unsound in first-order linear logic (**?**).

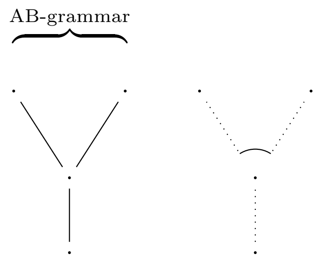


Figure 4.11: Lambek grammar

therefore follows a 'medial extraction' analysis, whereas items 4.20 and 4.21 predict a type of quantifying-in behaviour: item 4.20 is modelled after the type assigned to a generalised quantifier but with an extra *np* argument; it takes as its argument a sentences missing a noun phrase at the position of the adverb (just like a generalised quantifier takes a sentence missing an *np* at the position of the quantifier as its argument), making the odd prediction that adverbs occur at the same place as noun phrases. Formulas 4.19 and 4.20 therefore predict Sentences (1) and (2), along with many other strange possibilities, are correct.

(1) John deliberately Mary hit.

(2) Mary the friend of deliberately left.

Other higher-order Lambek calculus formulas have similar problems when we try to translate them into ACG. For example, the word "and" when used for the coordination of transitive verbs has 3024 possible translations, with 420 generating the correct surface structure and 148 having, in addition, the correct semantics as a possible reading. However, these many possibilities all follow the same pattern we have seen above for adverbs: they use a combination of extraction-like and quantifying-in constructions and therefore overgenerate (**?**, **?**)[7].

4.7 A visual comparison of the different calculi

Figure 4.11 shows the Lambek calculus connectives as links for first-order linear logic proof nets. Curry's (**?**) criticism of the Lambek calculus connectives, seen from the current perspective, is that they combine subcategorization information (functor-argument structure) and string operations. Although from a modern proof-theoretical point of view (**?**) it is perfectly valid to combine

[7]This is not to say that ACG cannot treat these phenomena *at all*, but that solutions require us to do at least one of the following: 1) abandon type-logical deep structure, 2) use a lexical duplication strategy. Some partial results have been obtained reducing the worst cases of overgeneration by using one or both of these options (**?**, **?**, **?**) all of which result in an explosion of the grammar size.

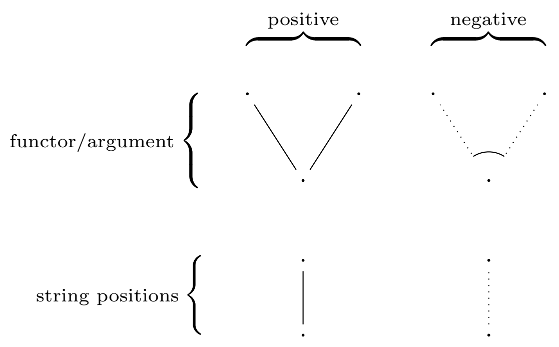


Figure 4.12: First-order linear logic

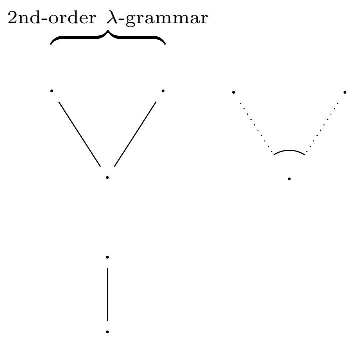


Figure 4.13: Lambda grammars

multiple positive and multiple negative rules into a single rule, separating the two gives us more freedom (that is, it allows us to express more relations between the string positions and go beyond simple concatenation — the prefix and postfix operations of the Lambek calculus).

As shown in Figure 4.12, the first-order linear logic solution decomposes the Lambek connectives into separate subcategorization and string position components. This decomposition answers Curry's critique in a very simple way.

Curry's own solution is different and causes a loss of symmetry: as Figure 4.13 makes clear, the positive universal link is missing! This loss of symmetry is easy to miss in a unification-based presentation of the logic where, in addition, the quantifiers occur only as an implicit prefix of the formula. For a logician/proof theorist, this is worrying since many classical results and desirable properties of the system (restriction to atomic axioms, cut elimination, interpolation[8]) depend on this symmetry. However, this also means

[8]Interpolation, proved first for the Lambek calculus by **?** is a key component of the

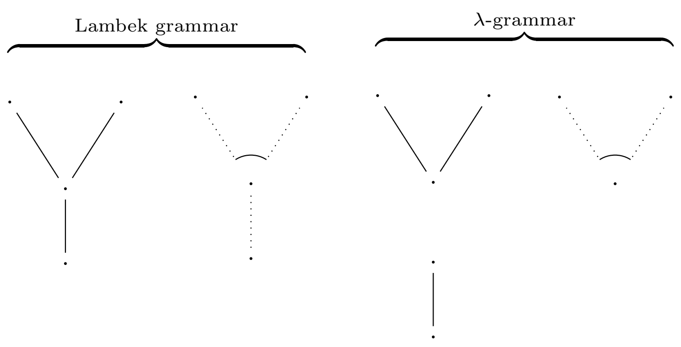


Figure 4.14: Hybrid grammar

that lambda grammars implicitly claim that positive A/B and $B\backslash A$ formulas play no useful role in language modelling or at least that these formulas are replaceable. As we have seen in the previous section, this leaves lambda grammars without a treatment of adverbs, coordination, gapping and other phenomena.

Another way to look at this is that lambda grammars require all formulas to be expressed in prenex normal form (using only the $\forall$ and $\multimap$ connectives, and without function symbols). However, because we are operating under several restrictions (linear logic without function symbols), not all formulas have such a prenex normal form. The following are all underivable (assuming no occurrences of x in B). Refer back to Figure 4.2 to see why the first statement is underivable.

$$\begin{aligned}(\forall x.A) \multimap B &\nvdash \exists x(A \multimap B)\\ \exists x(A \multimap B) &\nvdash (\forall x.A) \multimap B\\ B \multimap \exists x.A &\nvdash \exists x.(B \multimap A)\\ \exists x.(B \multimap A) &\nvdash B \multimap \exists x.A\end{aligned}$$

The hybrid solution to the problem of the missing rules is shown in Figure 4.14: we simply reintroduce the positive Lambek connectives directly. There are now two ways of coding the negative Lambek connectives. The resulting system is also greater than the sum of its parts, since gapping, which has a satisfactory neither in Lambek grammars nor in lambda grammars, can be elegantly treated in hybrid type-logical grammar (**?**, **?**).

context-freeness proof for the Lambek calculus of **?**. However, it seems this proof strategy does not have a simple extension to any of the systems under consideration here, since, as noted by **?**, a system which is not completely non-commutative need not have interpolants which are smaller. Moreover, a simple counting argument (**?**, Section 4) shows that, since a sentence with n quantifiers has up to $n!$ readings, we cannot enumerate these using MCFGs and related formalisms (as would be needed to establish the class of formal languages).

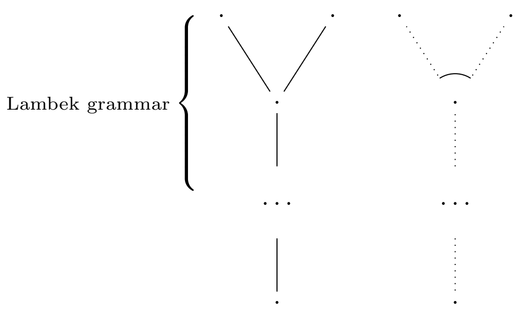


Figure 4.15: D grammars, binary

Symmetry is still lost[9], but empirically the system seems comparable to the Displacement calculus (**?**): the Displacement calculus has the full symmetry absent from hybrid type-logical grammars. In spite of this — as we have seen at the end of Section 4.5 and in Section **??** — in many cases, the analyses proposed for the two formalisms basically agree, as is made especially clear by their translation into first-order linear logic.

The differences between the two systems seems to be that hybrid type-logical grammars can, like lambda grammars, generate non-well-nested string languages and that Displacement grammars (seen from the point of view of hybrid type-logical grammars) allow the Lambek connectives to outscope the discontinuous connectives. Further analysis is necessary to decide which of these two systems has better empirical coverage.

D grammars (**?**) have a different perspective, which is shown in Figure 4.15. Functor argument structure and string positions are still joined, but a greater number of combinations are possible (from 0 to n quantifiers, for a small value of n determined by the grammar). Lambek grammars are now the restriction to a single quantifier for each binary connective.

D grammars enriched with bridge, left projection and right projection, shown in Figure 4.16, permit combinations of string position/subcategorization which are not of the same polarity. These uses are rather restricted compared to the visually similar quantifier link of first-order linear logic: essentially, they enable us to require that a pair of positions spans the empty string.

4.8 Conclusions

Summing up, first-order linear logic decomposes the connectives of different grammatical frameworks — the Lambek calculus, lambda grammars, hybrid

[9]Neither full logical symmetry nor having the Lambek calculus as a subsystem is of course necessary to have a formal system with sufficient empirical coverage, as shown, for example by CCG (**?**). However it calls for further investigation as to what exactly is absent from the system and if this absence is important from a descriptive point of view.

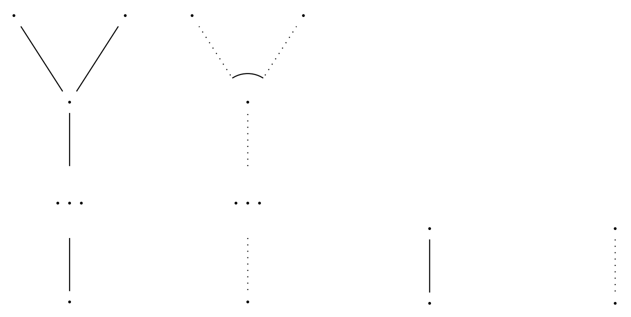


Figure 4.16: D grammars

type-logical grammars and the Displacement calculus — in a natural way into its four types of links. This visual comparison both highlights the differences between these calculi and opens the way for a more detailed comparison of the descriptive limitations of one calculus compared to another.

The translations to first-order linear logic slightly increase the formula size in terms of the total number of connectives in the lexical entries. However, the basic operations are simple and well-understood and the first-order variables actually function as powerful constraints during proof search. Thanks to the embedding results (**?**, **?**), we can import the large range of linguistic phenomena treated by Displacement grammars and hybrid type-logical grammars directly into first-order linear logic.

From the point of view of first-order linear logic, the connectives of the other calculi are synthetic connectives: combined connectives of the same polarity. We can mix and match these synthetic connectives as we see fit. We can also exploit the symmetry of first-order linear logic and use lambda grammar lexical entries as arguments[10], restoring the symmetry of lambda grammars (and of hybrid type-logical grammars). In addition, we can add the product $\otimes$ and quantifier $\exists$ to our calculus essentially for free. Moreover, as discussed by **?** we can use the quantifiers of first-order linear logic to give an account of agreement and island constraints as well. So we can improve upon Displacement grammar analyses by adding agreement and island constraints and improve upon hybrid type-logical grammar analyses by adding symmetry, across-the-board extraction, agreement and island constraints, all with the same logical primitives.

Besides the theoretical benefits of providing a common logical core for a number of prominent formalisms in type-logical grammars, we can also use first-order linear logic as a parser/theorem prover for all these formalisms. The proof net theorem prover of **?** allows the grammar writer to specify his grammar as a lambda grammar, Lambek grammar, displacement grammar or hybrid grammar, the theorem prover then translates the grammar into first-

[10]From the perspective of first-order linear logic, it seems that the so-called *phenominators* of **?** and **?** provide a way of doing this, but this needs to be further investigated.

order linear logic, and when a proof is found the proof net in first-order linear logic can be translated back into the source logic (as a natural deduction or sequent calculus proof). If the grammar writer desires, the underlying first-order linear logic formulas and proofs can be completely hidden. It is therefore possible to view first-order linear logic as a sort of underlying 'machine language' for these other logics. In this view, the lambda-terms of hybrid grammars and the logical connectives of Lambek and Displacement grammars are convenient high-level descriptions, which hide the technical aspects of position variables and quantifiers.

First-order (multiplicative, intuitionistic) linear logic and its corresponding proof nets are a simple and standard fragment of linear logic. However, translating type-logical grammars into first-order linear logic requires us to show that the structure and operations of such type-logical grammars can be uniquely recovered from a finite amount of constants and variables. For the Displacement calculus, this is done by showing that we inductively maintain a linear order on the string segments represented by the free variables/constants. For lambda grammars, this is done using principal types, where the variables represent lambda terms module $\beta\eta$ equivalence. However, similarly efficient encodings may not be available for other type-logical grammars, such as multimodal type-logical grammars, the logical of scope $\mathbf{NL}_\lambda$, and the Lambek-Grishin calculus $\mathbf{LG}$. In the next chapter, we will introduce a graph rewriting perspective on proof nets, inspired by the interaction nets of **?** and the multimodal proof nets of **?** and show they can be extended to give an account of *all* modern type-logical grammars.

5 Graph rewriting

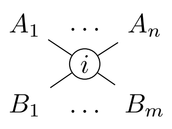
5.1 Multimodal proof nets

As we have seen in Section 3.2, proof nets for multiplicative linear logic are built from the one-sided sequent formulation of linear logic, which exploits the de Morgan symmetries to move all formulas to the right hand side of the turnstile and to move all negations to the atoms. In this setup, intuitionism corresponds to a restriction on the forms of allowed formulas and sequents, where these restrictions are formulated on negative and positive subformulas. As a consequence of this one-sided setup, proof structures have only conclusions and no hypotheses, and all links have the main formula of the link as their conclusion.

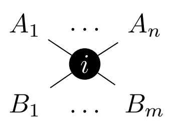
The multimodal proof nets of **?** are based on the standard two-sided sequent formulation of multimodal type-logical grammars. These proof nets have both hypotheses and conclusions. In this setup, each connective has a link with the main formula as its conclusion (corresponding to the positive polarity version of a link in the multiplicative intuitionistic system) as well as an up-down symmetric link with the main formula as its premiss (corresponding to the negative polarity version of a link for multiplicative intuitionistic proof nets).

5.1.1 Two-sided Links

Multiplicative proof nets are hypergraphs where the vertices are formula occurrences and the hyperedges are links connecting these formulas. The general form of the links is shown below.



Tensor link (general form)



Par link (general form)

There are two types of links: tensor links, with an open center (shown above on the left), and par links, with a filled center (shown above on the right)[1]. The formulas written above the central node of a link are its premisses, whereas the formulas written below it are its conclusions. Left-to-right order of the premisses as well as the conclusions is important. A par link has an arrow pointing either to one of its premisses or to one of its conclusions.

Logical links have the following general principles:

- each connective has both a tensor link and a par link, which are the up-down symmetric images of each other,
- one of the formulas is the main formula of the link, the others are its active formulas (the direct subformulas of the main formula),
- the par link has an arrow pointing to its main formula.

A link in a multimodal proof net is defined as follows.

Definition 5.1 *A* link *is a tuple* $\langle$*Type*, *Label*, *Premisses*, *Conclusions*, *Main*$\rangle$, *where* Type *is one of* tensor *and* par *(displayed as an open and filled central circle respectively),* Label *is a member* i *of a set of labels or modes* I, Premisses *is a list of nodes representing the premisses of the link (these are displayed from left to right above the central circle),* Conclusions *is a list of nodes representing the conclusions of the link (these are displayed from left to right below the central circle), and Main designates one of the premisses, one of the conclusions or no vertex as the main vertex of the link (these are displayed by an outgoing arrow from the central circle).*

Definition 5.1 is very general and can be instantiated in various ways. We start with a very simple example. In general, the tensor links define the structure of the sequents; intuitively, tensor links are structure-building operations whereas par links are structure-erasing operations. Suppose the tensor link has two premisses and a single conclusion. This means the structures under consideration are labeled binary trees, with label-alphabet I. Given this choice of structure, there are three choices for the main formula: the left premiss, the right premiss and the conclusion. This gives us the links shown in Table 5.1.

[1] I have followed **?** in separating the links between par and tensor. For the binary multimodal connectives, par and tensor links correspond to underlying par and tensor links in linear logic. However, this is no longer true for many of the extended versions of these links that we will see later. In hindsight, it might have been better to follow the terminology of **?** and call par links 'asynchronous' or 'reversible' links, and call tensor links 'synchronous' or 'not reversible' links.

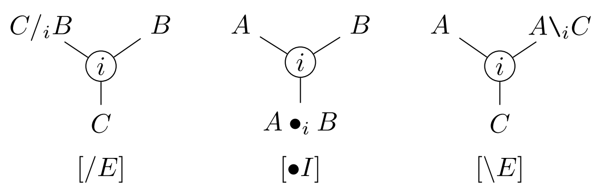


Table 5.1: Tensor links for multimodal proof nets

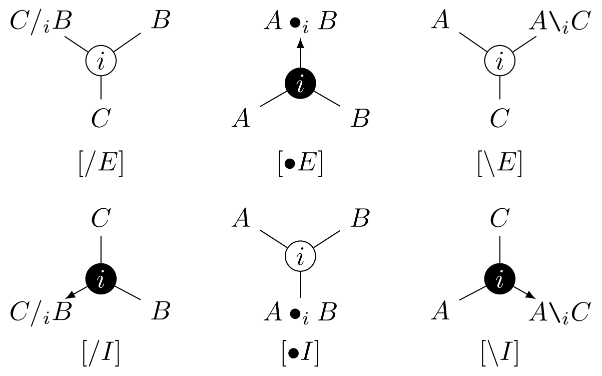


Table 5.2: Links for multimodal proof nets

According to the general principle, each tensor link induces an up-down symmetric par link, with an arrow pointing to the main formula. Applying this principle to the tensor links of Table 5.1 produces Table 5.2. Each connective has two links, one where the main formula is a premiss (the top row of the table) and one where the main formula is a conclusion (the bottom row of the table).

The top row of Table 5.2 lists the links corresponding to the elimination rules of natural deduction (and the left rules in sequent calculs), the bottom row those corresponding to the introduction rules (and the right rules in sequent calculus).

5.1.2 Proof structures

A proof structure is a special type of (hyper)graph, where the vertices are formula occurrences and the (hyper)edges are as defined in Table 5.2.

Definition 5.2 *A* proof structure *is a set of formula occurrences and a set of links such that:*

1. *each link is an instance of one of the links of Table 5.2 (for some A, B, C, i),*

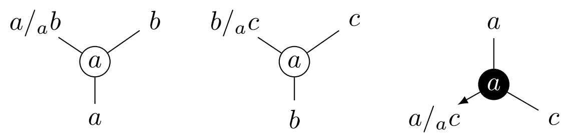


Figure 5.1: Lexical unfolding of $a/_a b, b/_a c \vdash a/_a c$

2. *each formula is at most once the premiss of a link,*

3. *each formula is at most once the conclusion of a link.*

A formula which is not the premiss of any link is a conclusion of the proof structure. A formula which is not the conclusion of any link is a hypothesis of the proof structure.

Compared to the standard one-sided definition of proof nets, our proof structures are *modules* (see Definition 3.3 in Section 3.2.1). In other words, the distinction between proof structures and modules disappears in the two-sided case. Most other definitions are unchanged for the two-sided case. For example, we still call a maximal, connected set of tensor links a *component* (and, as usual, we can obtain the components of a proof structure by taking the connected components after removing all par links from the structure).

We say a proof structure with hypotheses Γ and conclusions Δ is a proof structure of $\Gamma \vdash \Delta$ (we are overloading of the '$\vdash$' symbol here, though this use should always be clear from the context; note that Δ can contain multiple formulas).

After the first step of lexical lookup we have a sequent $\Gamma \vdash C$, and we can enumerate its proof structures as follows: unfold the formulas in Γ, C, ensuring that the formulas in Γ are hypotheses and the formula C is a conclusion of the resulting structure, until we reach the atomic subformulas, then identify atomic subformulas. Taken together, these are just the first three steps of the procedure for standard one sided proof nets (as discussed, for example, in Section 4.3), only with slightly different structures. We turn to the last step, checking correctness, below. By the conditions on proof structures, this identification step can only identify hypotheses with conclusions and must leave all formulas of Γ, including atomic formulas, as hypotheses and C as a conclusion.

Figure 5.1 shows the lexical unfolding of the sequent $a/_a b, b/_a c \vdash a/_a c$. It is already a proof structure, although it is a proof structure of sequent $a, a/_a b, b, b/_a c, c \vdash a, a/_a c, b, c$ (as noted above, according to the standard definitions of one-sided proof nets, a 'partial proof structure' like the one shown in the figure would be called a *module*).

To turn this proof structure into a proof structure of $a/_a b, b/_a c \vdash a/_a c$, we identify the atomic formulas. In this case, there is only a single way to do this, since a, b and c all occur once as a hypothesis and once as a conclusion,

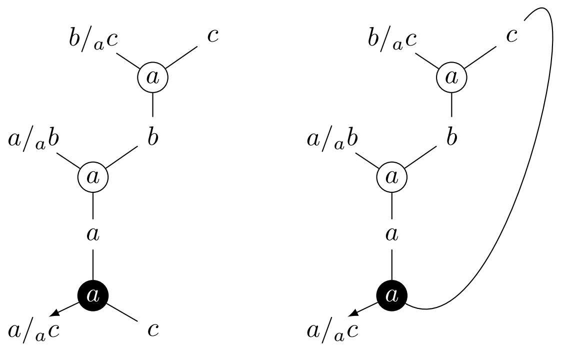


Figure 5.2: The proof structure of Figure 5.1 after identification of the a and b atoms (left) and after identification of all atoms

though in general there may be many possible matchings. Figure 5.2 shows, on the left, the proof structure after identifying the a and b formulas. Since left and right (linear order), up and down (premiss, conclusion) have meaning in the graph, connecting the c formulas is less obvious: c is a conclusion of the $/I$ link and must therefore be below it, but a premiss of the $/E$ link and must therefore be above it. This is hard to achieve in the figure shown on the left. A possible solution would be to draw the figure on a cylinder, where 'going up' from the topmost c we arrive at the bottom one. However, for ease of type-setting and reading the figure, I have chosen the representation shown in Figure 5.2 on the right. The curved line goes up from the c premiss of the $/E$ link and arrives from below at the $/I$ link, as desired. One way so see this strange curved connection is as a graphical representation of the coindexation of a premiss with a rule in the natural deduction rule for the implication.

Figure 5.2 therefore shows, on the right, a proof structure for the sequent $a/_ab, b/_ac \vdash a/_ac$. However, is it also a *proof net*, that is, does it correspond to a proof? In a multimodal logic, the answer depends on the available structural rules. For example, if no structural rules are applicable to mode a then $a/_ab, b/_ac \vdash a/_ac$ is underivable, but if mode a is associative, then it is derivable.

5.1.3 Proof nets and graph contractions

We decide whether a proof structure is a proof net based only on properties of the graph. As a first step, we erase all formula information from the internal nodes of the graph; for administrative reasons, we still need to be able to identify which of the hypotheses and conclusion of the structure correspond to

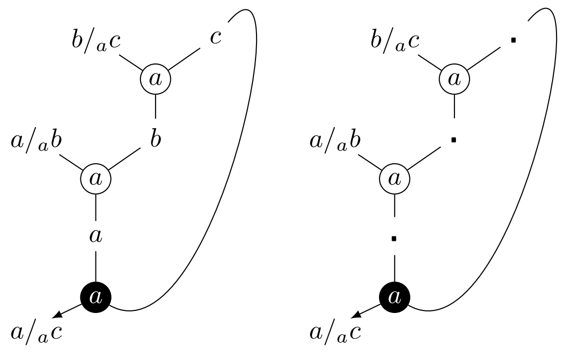


Figure 5.3: The proof structure of Figure 5.2 (left) and its abstract proof structure (right)

which formula occurrence[2]. All relevant information for correctness is present in this graph, which we call an *abstract proof structure.*

We talked about how the curved line in proof structures (and abstract proof structure) corresponds to the coindexation of discharged hypotheses with rule names for the implication introduction rules. However, the introduction rules for multimodal type-logical grammars actually do more than just discharge a hypothesis, they also check whether the discharged hypothesis is the immediate left (for $\backslash I$) or right (for $/I$) daughter of the root node, that is, that the withdrawn hypothesis A occurs as $A \circ_i \Gamma$ (for $\backslash I$ and mode i) or $\Gamma \circ_i A$ (for $/I$ and mode i). The par links in the (abstract) proof structure represent a sort of 'promise' that we will produce the required structure. We check whether it is satisfied by means of contractions on the abstract proof structure.

The multimodal contractions are shown in Table 5.3. All portrayed configurations contract to a single vertex: we erase the two internal vertices and the paired links and we identify the two external vertices, keeping all connections of the external vertices to the rest of the abstract proof structure as they were: the vertex which is the result of the contraction will be a conclusion of the

[2]We make a slight simplification here. A single vertex abstract proof structure can have both a hypothesis and a conclusion without these two formulas necessarily being identical, e.g. for sequents like $(a/b) \bullet b \vdash a$. Such a sequent would correspond to the following abstract proof structure.

$$\begin{matrix} (a/b)\bullet b \\ \cdot \\ a \end{matrix}$$

So, formally, both the hypotheses and the conclusions of an abstract proof structure are assigned a formula and when a node is both a hypothesis and a conclusion it can be assigned two different formulas. In order not to make the notation of abstract proof structure more complex, we will stay with the simpler notation. **?** present the full details.

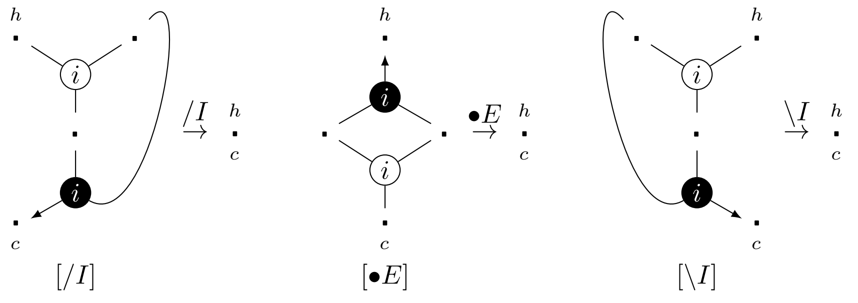


Table 5.3: Contractions — multimodal binary connectives

same link as the top external vertex (if it is not the conclusion of any link, it will be a hypothesis of the abstract proof structure) and it will be a premiss of the same link as the bottom external vertex (if it is not the premiss of any link, it will be a conclusion of the abstract proof structure).

The contraction for $/I$ checks if the withdrawn hypothesis is the right daughter of a tensor link with the same mode information i, and symmetrically for the $\backslash I$ contraction. The $\bullet E$ contraction contracts two hypotheses occurring as sister nodes.

All contractions are instantiations of the same pattern: a tensor link and a par link are connected, respecting left-right and up-down, to the two vertices of the par link without the arrow.

To get a better feel for the contractions, we will start with its simplest instances. When we do pattern matching on the contraction for $/I$, we see that it corresponds to the following patterns, depending on our choice for the tensor link (the par link is always $/I$).

$$\begin{aligned} C/_iB &\vdash C/_iB \\ A &\vdash (A \bullet_i B)/_iB \\ A &\vdash C/_i(A\backslash_i C) \end{aligned}$$

Figure 5.4 shows the three proof nets corresponding to the patterns above (these hold for all $i \in I$).

A proof structure is a proof net iff it contracts to a tree containing only tensor links using the contractions of Table 5.3 and any structural rewrites, discussed below — **?** present full proofs. In other words, we need to contract all par links in the proof structure according to their contraction, each contraction ensuring the correct application of the rule after which it is named. The abstract proof structure on the right of Figure 5.3 does not contract, since there is no substructure corresponding to the $/I$ contraction: for a valid contraction, a par link is connected to both ‘tentacles’ of a single tensor link, and in the figure the two tentacles without arrow are connected to different

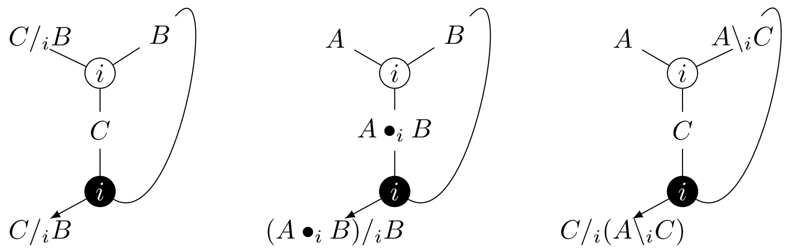


Figure 5.4: The three simplest proof nets requiring the $/I$ contraction.

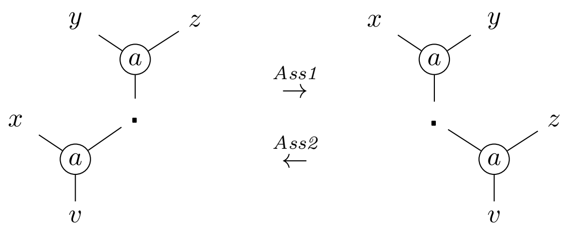


Figure 5.5: Structural rewrites for associativity of mode a.

tensor links. This is correct, since $a/_ab, b/_ac \vdash a/_ac$ is underivable in a logic without structural rules for a.

However, we have seen that this statement becomes derivable once we add associativity of a and it is easily verified to be a theorem of the Lambek calculus. How can we add a modally controlled version of associativity to the proof net calculus? We can add such a rule by adding a rewrite from a tensor tree to another tensor tree with the same set of leaves. The rewrite for associativity is shown in Figure 5.5. To apply a structural rewrite, we replace the tree on the left hand side of the arrow by the one on the right hand side, reattaching the leaves and the root to the rest of the abstract proof structure.

Just like the structural rules, a structural rewrite always has the same leaves on both sides of the arrow — neither copying nor deletion is allowed[3], though we can reorder the leaves in any way (the associativity rule doesn't reorder the leaves).

Figure 5.6 shows how the contractions and the structural rewrites work together to derive $a/_ab \circ_a b/_ac \vdash a/_ac$.

We start with a structural rewrite, which rebrackets the pair of tensor links. The two hypotheses are now the premisses of the same link, and this

[3]From the point of view of linear logic, we stay within the purely multiplicative fragment, which is simplest proof-theoretically.

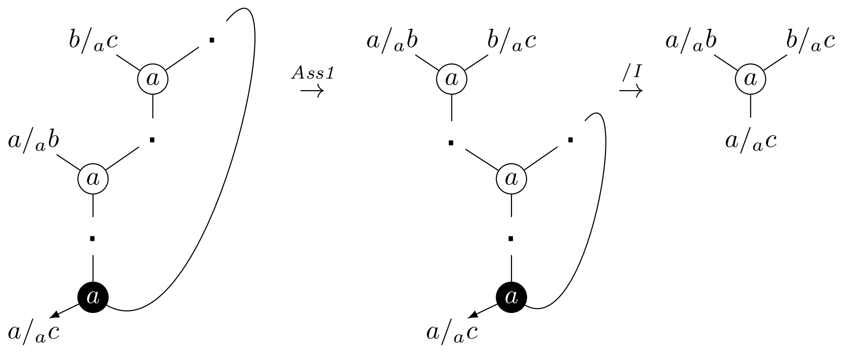


Figure 5.6: Structural rewrite and contraction for the abstract proof structure of Figure 5.3, showing this is a proof net for $a/_a b \circ_a b/_a c \vdash a/_a c$

also produces a contractible structure for the $/I$ link. Hence, we have shown the proof structure to be a proof net.

Although the structural rules give the grammar writer a great deal of flexibility, such flexibility complicates proof search. As discussed at the end of Section 4.3 on page 76, theorem proving using proof nets is a four step process, which in the current situation looks as follows.

1. *Lexical lookup* a choice of a formula for each word in the input sequence (and a choice of the goal formula),

2. *Unfolding* decomposing the complex formulas using the links (this step is deterministic and can be done in linear time)

3. *Identification of the atoms* connecting positive (conclusion) and negative (hypothesis) atomic formulas to produce a proof structure of the input sequent,

4. *Check correctness* by means of graph rewriting.

In the current case, both graph rewriting and the identification of atoms are complicated[4]. The reason the graph rewriting component is more complex in the multimodal case is that although the rewrites cannot increase the structure[5], they do not necessarily reduce the size of the structure (unlike the first-order contractions of Section 4.3). Moreover, we need to keep track

[4]Lexical ambiguity is a major problem for automatically extracted wide-coverage grammars as well. We will return to this problem in Chapter **??**.

[5]This condition is imposed by **?** to guarantee decidability even when unary branches are added.

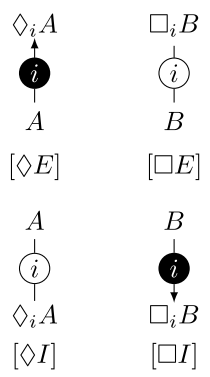


Table 5.4: Links for the unary connectives

of already visited structures (for example, to avoid repeatedly applying the associativity rules back and forth) and confluence of the rewrite operations is not guaranteed, so some sort of search is required, at least in the most general case (e.g. breadth-first search while keeping track of visited structures[6]).

However, the graph rewriting calculus can also serve as some sort of stepping stone towards a confluent calculus. This is easiest for rules like associativity, which simply involve n premiss tensor links (that is, the premisses of the tensor link directly represent the list structure of the antecedent) for the Lambek calculus (**?**), but we can use similar strategies for other structural rules (**?**, Section 7.1.3). We will see several instances of this later in this chapter.

5.2 Generalized multimodal proof nets

The links for multimodal proof nets are sufficiently general to allow quite a number of connectives in addition to the binary ones discussed before. One of the simplest additions is to allow unary branches. This means our sequents are structured as labelled trees where all nodes have either one or two daughters (such trees are sometimes called 1-2-trees), and it gives a pair of new connectives, generally written $\Diamond$ and $\Box$, operating on the new unary branches.

Table 5.4 shows the new links for the unary connectives. The contractions for the unary connectives look as shown in Table 5.5.

We can see the links and contraction for '$\Diamond$' as a version of the links and contraction for '$\bullet$' with one of the branches removed. Similarly, the links and contractions for '$\Box$' correspond to those of '/' (or '\').

[6]This strategy is used by **?** because it finds shortest rewrite sequences, that is, without any unnecessary structural rule applications.

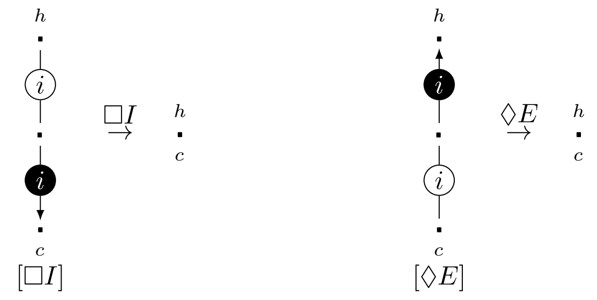


Table 5.5: Contractions for the unary connectives

The correctness condition for proof nets with the unary modalities added remains the same: a proof structure is a proof net whenever its abstract proof structure contracts to a tree using the contractions and the structural tree rewrites corresponding to the structural rules of the grammar.

To make this more concrete, we will now translate the analysis of extraction in multimodal type-logical grammars from Section 2.1.1 to a proof net analysis. This is quite straightforward, but I believe it is important at this point to have a non-trivial example of how to use multimodal proof nets with the new unary branches as a tool for theorem proving.

We want to analyse the noun shown as (1) below.

(1) book which John read yesterday

These are the now-classic cases of medial extraction motivating the development of multimodal grammars and unary connectives (as well as many other modern type-logical grammars).

$$\begin{aligned} \mathit{Lex}(\text{book}) &= np \\ \mathit{Lex}(\text{which}) &= (n\backslash n)/(s/\Diamond\Box np) \\ \mathit{Lex}(\text{John}) &= np \\ \mathit{Lex}(\text{read}) &= (np\backslash s)/np \\ \mathit{Lex}(\text{yesterday}) &= s\backslash s \end{aligned}$$

The lexicon above is quite standard, the only formula which merits some discussion is the one assigned to "which": the formula $(n\backslash n)/(s/\Diamond\Box np)$ indicates it is looking for a sentence missing a noun phrase anywhere[7]. The two structural rules required for this behaviour are repeated below.

$$\frac{\Gamma[\Delta_1 \circ (\Delta_2 \circ \langle\Delta_3\rangle)] \vdash C}{\Gamma[(\Delta_1 \circ \Delta_2) \circ \langle\Delta_3\rangle] \vdash C}\ MA \qquad \frac{\Gamma[(\Delta_1 \circ \langle\Delta_3\rangle) \circ \Delta_2] \vdash C}{\Gamma[(\Delta_1 \circ \Delta_2) \circ \langle\Delta_3\rangle] \vdash C}\ MC$$

[7]More precisely, it can be missing from any right branch of the structure, which **?** and **?** argue to be the correct behaviour for English.

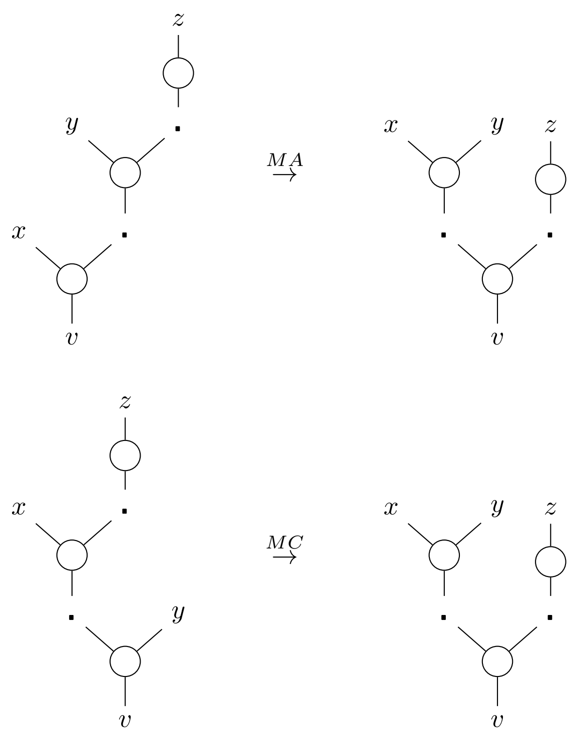


Figure 5.7: Tree rewrites corresponding to the structural rules for mixed associativity and mixed commutativity

From the graph rewriting perspective, the structural rule correspond to tree rewrites on the abstract proof structures. Figure 5.7 shows the rewrites corresponding to the mixed associativity and mixed commutativity structural rules. The tree rewrites are just a graphical way to read the operations of the structural rules, they allow us to replace the structure used as the premiss of the structural rules by the one used as its conclusions (read $x = \Delta_1$, $y = \Delta_2$ and $z = \Delta_3$ to make this even more clear). Both rewrite rules provide a way to move a unary branch closer to the root of the tree.

Returning to our proof net analysis, Figure 5.8 shows the unfolding of the lexical formulas. We have numbered the atomic formulas to allow for easy reference to them, these numbers are not formally part of the proof structure. The square tables above the figure give a summary of the combinatorics of the axiom matching stage of proof search. The rows denote the atomic conclusions, whereas the columns denote the atomic hypotheses of the structure. Since "book" and "John" are hypotheses of the proof (that is, their formulas should appear in the antecedent of the end-sequent of the proof) and the formula n_4 labelled *Goal* is a conclusion of the proof (that is, it should be the succedent of the end-sequent of the proof) these formulas are only available

	n_2	n_4
n_1		
n_3		

	np_3	np_4
np_1		
np_2		

	s_1	s_3
s_2		
s_4		

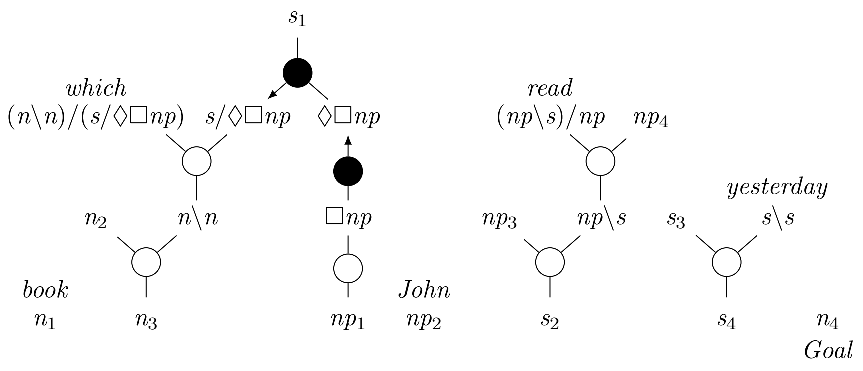


Figure 5.8: Lexical lookup and possible atom matchings for the multimodal proof net analysis of "book which John read yesterday"

as conclusions (in the case of "book" and "John") and as a hypothesis (in the case of the goal formula n_4).

We have two possibilities for linking the n formulas, two for the np formulas and two for the s formulas. However, as with first-order linear logic proof nets, we can use a number of strategies for eliminating possibilities.

One powerful strategy is exploiting the fact that, whatever structural rules we use, our final proof should translate through a forgetful mapping to a multiplicative linear logic proof. This means we can use the acyclicity and connectedness criterion to eliminate invalid connections. For example, connecting the n_1 of "book" to the n_4 of the goal formula produces a disconnected structure: it would create an isolated vertex which can never be connected to the rest of the structure. Similarly, connecting n_2 to n_3 produces a cycle. This means that for the noun formulas, only the connections $n_1 - n_2$ and $n_3 - n_4$ can lead to a proof.

The s formulas are similarly restricted. Connecting s_3 to s_4 produces a cycle. This leaves s_2 as only alternative for s_3, and s_1 as only alternative for s_4. For the np formulas, neither possibility violates the correctness conditions of multiplicative linear logic. However, the connection of the np_2 of "John" to the object noun phrase np_4 of "read" would not allow us to produce an analysis with the correct word order: it would try (and fail) to find a proof for "book which read John". We can use some reasoning about the available structural rewrites to argue why this atom connection can not produce a proof (for example, by showing that the structural rules only allow us to move a structure with a unary bracket from one right branch to another, and the connections $np_2 - np_4$, $np_1 - np_3$ would put the unary branch at a left branch, resulting in failure of contraction). However, this requires some complex reasoning about properties of grammar-dependent sets of structural rules. In practical applications, a number of solutions have been used. The theorem prover of **?** allows the grammar writer to explicitly declare which modes are 'word order preserving' and for which the theorem prover can therefore reject partial proof structures not respecting the desired word order. The theorem prover of **?** allows the grammar writer to specify approximations in first-order linear logic for the structural rules. Using such specifications then allows the theorem prover the rule out any structure which is invalid according to these first-order linear logic approximations. **?** gives a detailed presentation of the different ways to reduce the number of axioms links in proof net proof search. Of course, the grammar writer needs to be extremely careful when making declarations of this type: an incorrect declaration of (say) a first-order approximation could cause the theorem prover to fail to find proofs. Efficiency improvements of this type should never filter out proofs, only proof attempts which can never be extended to proofs.

Figure 5.9 shows (on the left hand side) the result of performing all axiom links in the previous step. The reader is invited to verify the other axiom links can never reduce to a tree given the rewrite rules and contractions available[8]. On the right of the figure we have shown the corresponding abstract proof structure. As usual, the abstract proof structure is obtained by removing all formula information at the internal nodes. We have kept the words instead of their corresponding formulas as hypotheses of the abstract proof structure.

Looking at the abstract proof structure on the right hand side of Figure 5.9, we see that in order to show the structure is a proof net — that is, to convert it to a tree without par links — we need to do a minimum of two par contractions: one $\Diamond E$ contraction and one $/E$ contraction.

We can do the $\Diamond E$ contraction immediately, since the lexical entry for "which" already contains the required redex. However, performing this contraction removes our unary branch, thereby blocking all structural rule applications: both the MA and MC structural rules require a unary branch. In this analysis, the $\Diamond E$ contraction plays a role similar to the dereliction rule in linear logic: it takes a formula licensed to use additional structural rules and removes this license when we no longer need it. We therefore delay the $\Diamond E$ contraction until we have used all necessary structural rules.

Looking at the structural rules, only the mixed associativity rule MA can apply, and it produces the structure shown on the left of Figure 5.10. We have moved the unary branch up, from depth three in its component to depth two. We can then apply the mixed commutativity rule MC to move it further up to depth one, as shown on the right of Figure 5.10. We have moved the unary

[8]Changing the structural rules (or the lexical entry for "which") can make "book which read John" derivable (**?**). Although this sentence is odd for the given words, allowing it provides a way to treat cases of subject extraction such as "book which $[]_{np}$ was on the bestseller list for 14 months". Whatever the structural rules used, there can only be two readings for this sequent since, according to our previous discussion, no other readings are valid in multiplicative linear logic.

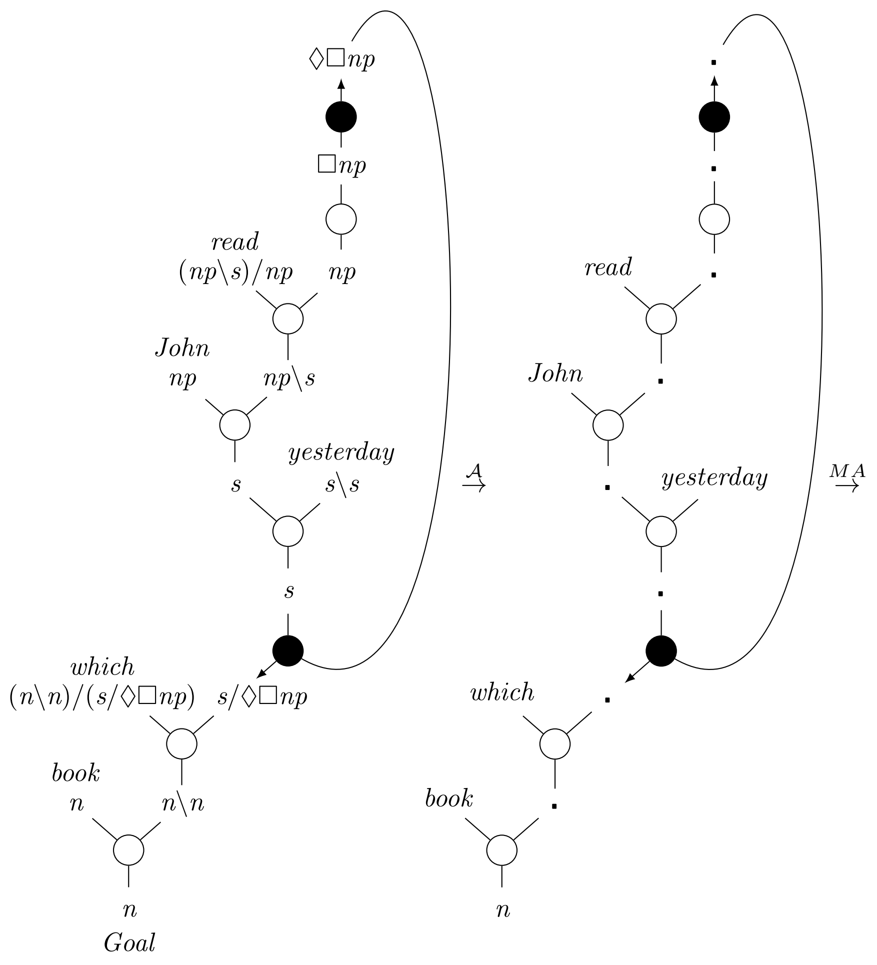


Figure 5.9: Proof structure and abstract proof structure for the lexical lookup of Figure 5.8

branch up as far as the structural rules allow us to go. The only rule which can apply now is the $\Diamond E$ contraction.

Figure 5.11 shows, on the left, the result of applying the contraction. The key reason for waiting so long applying the $\Diamond E$ contraction was to provide a structure which allows us to perform the $/I$ contraction immediately afterwards. The result of this contraction is shown on the right of Figure 5.11. The resulting tensor tree represents the following sequent.

$$\text{book} \circ (\text{which} \circ ((\text{John} \circ \text{read}) \circ \text{yesterday})) \vdash n$$

What we have shown with this example is how a multimodal grammar can be transparently translated into a proof net analysis for proof search. The for-

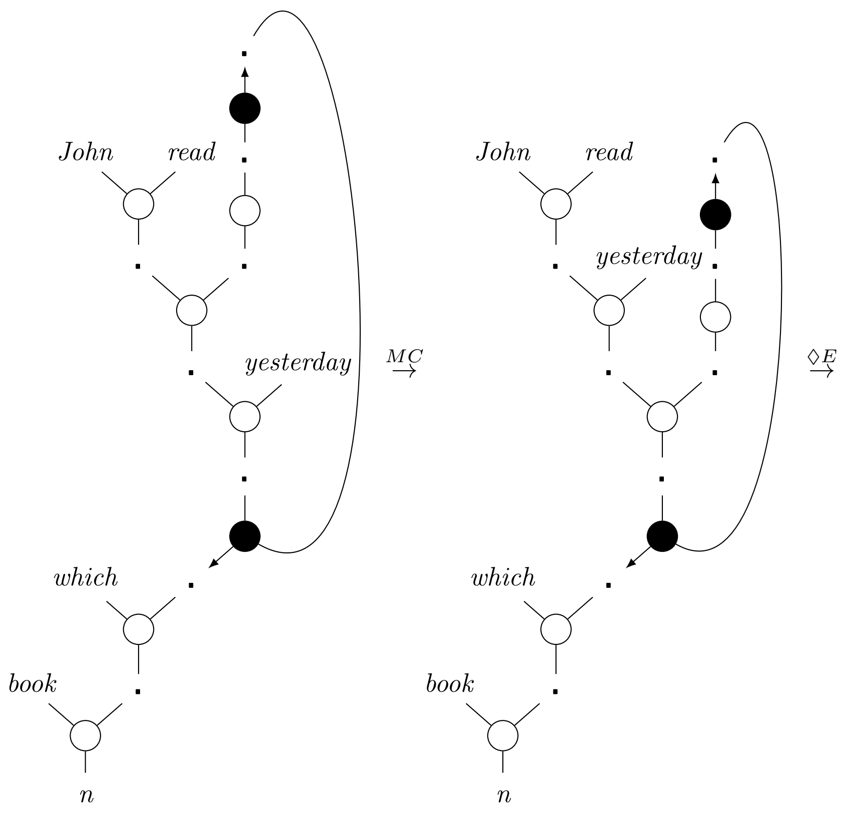


Figure 5.10: We rewrite the abstract proof structure from Figure 5.9 first with the MA conversion, then with the MC conversion

mula unfolding rules and corresponding contractions are fully general, whereas structural rules translate to tree rewrites. This provides a simple but effective proof search method, which, moreover, computes the structure of the words/formulas required for the derivation.

Compared the first-order linear logic, the graph rewrite operations required for proof search are much more flexible, but this flexibility has a price: because of their inherent flexibility, our rewrite operations can be complicated, and deciding whether or not a proof structure is a proof net becomes more of a search problem (as in symbolic artificial intelligence, with the associated problems of keeping track of nodes to be visited and nodes already explored) than a normalisation problem (as the contraction criterion for multiplicative linear logic, which we can apply eagerly and efficiently).

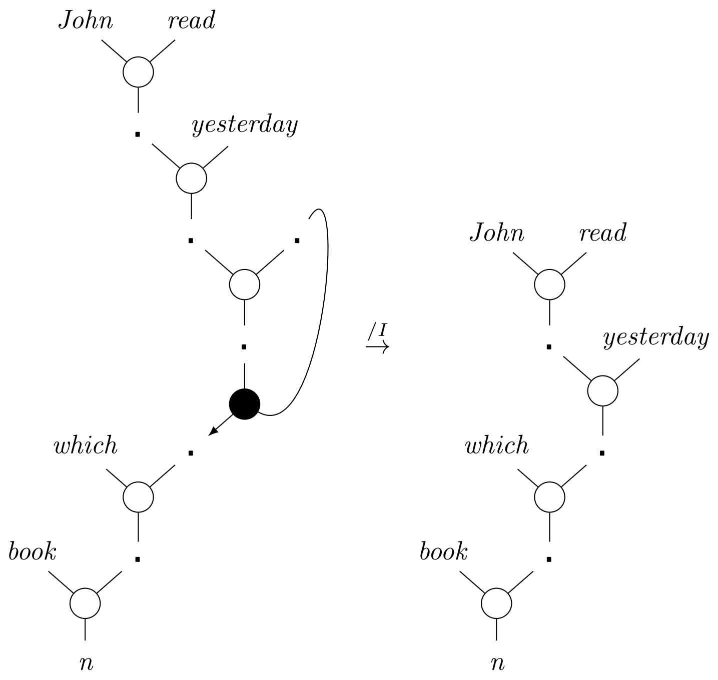


Figure 5.11: We complete the proof by first applying the $\Diamond E$ contraction, then the $/I$ contraction to the abstract proof structure of Figure 5.10

5.2.1 Discussion

Before going further, we are going to take a step back and look at the general properties of the proof net calculus which make it work, since these properties will allow us to extend the calculus to other logics. While the proof structures represent the logical backbone of a derivation, the abstract proof structures represent the structured sequents in our logical calculus. More specifically:

- the (acyclic) components of an abstract proof structure (that is, the tensor trees we obtain when removing all par links) represent structured sequents; we can uniquely recover the necessary information about hypothesis and conclusion formulas from the corresponding proof structure,
- the structural rewrites correspond to the application a structural rule to the sequent; this is just the standard way to translate between bracketed representations and trees,
- the par contractions verify we have the correct structure for the application of the logical rule. A par contraction can only apply when the active tentacles of the rule are attached to the same tensor link (and

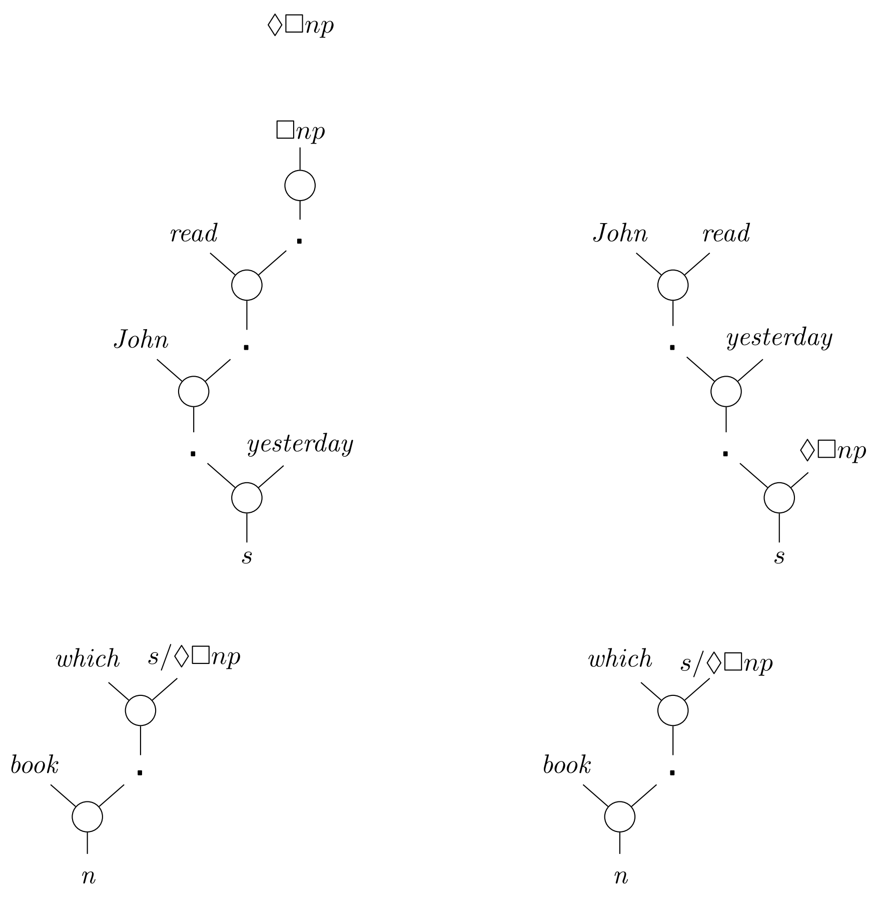


Figure 5.12: Components for the abstract proof structure on the right of Figure 5.9 (left) and the one of the left of Figure 5.11 (right).

therefore to the same component as well). The contraction removes a par/tensor combination and reconnects the rest of the structure (refer back to Figure 5.11 for an example); a par contraction removes a tensor link in addition to the par link and connects the two components attached to the par link (the one connected to its active formulas and the one connected to its main formula).

Figure 5.12 illustrates (on the left of the figure) how removing the par links from an abstract proof structure produces a set of tensor trees[9]. The

[9]If any component contains a cycle, then the structure is not a proof net.

three tensor trees represent the following three proofs.

$$\frac{}{\Diamond\Box np \vdash \Diamond\Box np} \qquad (5.1) \qquad units$$

$$\frac{\dfrac{John \vdash np \qquad \dfrac{read \vdash (np\backslash s)/np \qquad \dfrac{\dfrac{}{\Box np \vdash \Box np}}{\langle\Box np\rangle \vdash np}\,\Box E}{read \circ \langle\Box np\rangle \vdash np\backslash s}\,/E}{(John \circ (read \circ \langle\Box np\rangle)) \vdash s}\,\backslash E}{(John \circ (read \circ \langle\Box np\rangle)) \circ yesterday \vdash s}\,\backslash E \qquad yesterday \vdash s\backslash s \qquad (5.2) \qquad units$$

$$\frac{book \vdash n}{book \circ (which \circ s/\Diamond\Box np) \vdash n}\,\backslash E \frac{which \vdash (n\backslash n)/(s/\Diamond\Box np)}{which \circ s/\Diamond\Box np \vdash n\backslash n}\,/E \quad \frac{}{s/\Diamond\Box np \vdash s/\Diamond\Box np} \quad units \qquad (5.3)$$

To ensure sequentialisation and cut elimination, we need to ensure the components of the initial proof structure correspond to proofs, that structural rewrites extend the proof of the corresponding component (e.g. the structural rewrites in the middle component correspond to structural rules extending proof 5.2) and the par contractions correspond to application of the corresponding rule followed by combining the two proofs. In our example, once we have applied the MA and MC structural rules to proof 5.2, we can combine this proof with the axiom 5.1 using the $\Diamond E$ rule corresponding to the $\Diamond E$ contraction as follows.

$$\frac{\dfrac{}{\Diamond\Box np \vdash \Diamond\Box np}}{((John \circ read) \circ yesterday) \circ \Diamond\Box np \vdash s}\,\Diamond E \begin{array}{c}\vdots\\ ((John \circ read) \circ yesterday) \circ \langle\Box np\rangle \vdash s\end{array} \quad units \qquad (5.4)$$

Figure 5.12 shows the components at this stage of rewriting the abstract proof structure (corresponding to the abstract proof structure on the left of Figure 5.11), with the top component corresponding to proof 5.4 and the bottom component to proof 5.3. Since the $\Diamond\Box np$ hypothesis of the top component is the right daughter of the root node, we can apply the $/E$ contraction and its corresponding logical rule, combining the two proofs by substituting the proof of $(John \circ read) \circ yesterday \vdash s/\Diamond\Box np$ for the axiom $s/\Diamond\Box np$ axiom in 5.1 as follows.

$$\frac{book \vdash n}{book \circ (which \circ ((John \circ read) \circ yesterday)) \vdash n}\,\backslash E \frac{which \vdash (n\backslash n)/(s/\Diamond\Box np)}{which \circ ((John \circ read) \circ yesterday) \vdash n\backslash n}\,/E \quad \frac{\begin{array}{c}\vdots\\ ((John \circ read) \circ yes\end{array}}{(John \circ read) \circ ye} \qquad (5.5)$$

With all this in mind, the correctness, sequentialisation and cut elimination proofs become very simple (**?**) but it also gives an indication of the

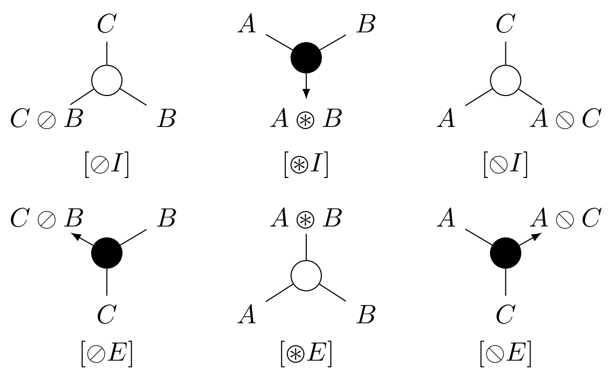


Table 5.6: Links for the Grishin connectives

properties we need to preserve when adapting the proof net calculus to different type-logical grammars.

There is another general point worth mentioning here: when we show that we can decide in polynomial time whether or not an abstract proof structure contracts to a tree, we thereby show the logic is in NP. This provides a simple tool for complexity analysis of type-logical grammars.

5.2.2 Up-down symmetry and Lambek-Grishin

One of the main interests of the general setup of multimodal proof nets is that it is easy to change the links and thereby the allowed structures. One possibility is to allow binary tensor nodes which are up-down symmetric with the binary multimodal nodes. The resulting calculus, which generally uses only a single mode, structures sequents as a type of free tree, where hyperedges are still oriented but the root in no longer necessarily unique. This leads to the Lambek-Grishin calculus, an extension of the Lambek calculus introduced by **?**.

Proof nets for the Lambek-Grishin calculus are obtained from two-sided proof nets by simply taking the up-down symmetric version of each link (or, alternatively, by replacing all tensor links with par links and vice versa). This produces the links shown in Table **??**. We can see, for example, that $\oslash E$ is the up-down symmetric version of $/I$ from Table 5.1, and similarly for the other rules. Whereas for standard multimodal links, up-down symmetry turns a tensor link into a par link, in the Lambek-Grishin calculus, each par link and each tensor link has an up-down symmetric version of the same type, using a connective of the opposite family (Lambek $/$, $\bullet$, $\backslash$ becoming Grishin $\oslash$, $\circledast$, $\obslash$, and vice versa), or viewing things differently, each link has another one of the same shape switching both connective family (Lambek/Grishin) and link type (par/tensor).

Since there are now tensor links with two conclusions, the structure of the

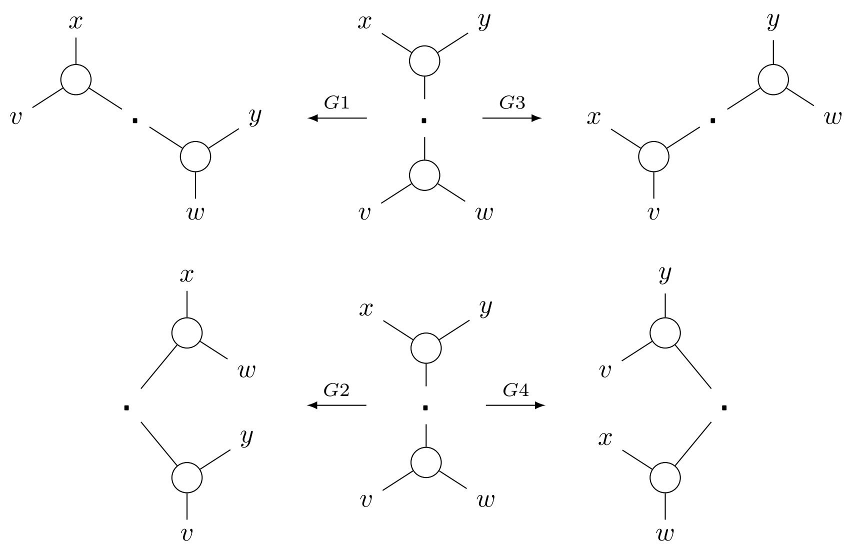


Table 5.7: Grishin interaction principles

antecedent is no longer a rooted tree. However, antecedents are still acyclic, connected graphs of the two types of tensor link (Lambek, branching upwards, and Grishin, branching downwards).

Contractions for the Grishin connectives are simply the up-down symmetric versions of the contractions for the multimodal Lambek calculus, although with only a single mode. The Grishin interaction principles, repeated below from Equations 2.7 and 2.8, correspond to the structural rules shown in Table **??**[10].

$$(A \oslash B) \bullet C \to A \oslash (B \bullet C) \qquad A \bullet (B \oslash C) \to (A \bullet B) \oslash C \qquad (5.6)$$

$$B \bullet (A \oslash C) \to A \oslash (B \bullet C) \qquad (A \oslash C) \bullet B \to (A \bullet B) \oslash C \qquad (5.7)$$

As an example, for their analysis of quantifier scope in the Lambek-Grishin calculus, **?** propose the formula $(s \oslash s) \oslash np$. We can show that we can derive the standard formula for subject quantifiers $s/(np\backslash s)$ in the (non-associative) Lambek calculus from this formula. Figure **??** show the proof structure and corresponding abstract proof structure for this example.

We cannot contract either of the par links in the initial abstract proof structure shown on the right of Figure **??**. However, we are in the correct configuration to apply the Grishin interaction principles from Table **??**. While the configuration allows us to rewrite according to *any* of $G1$, $G2$, $G3$ or $G4$, choosing $G1$ produces the structure shown on the right of Figure **??**.

[10]The correspondence between **??** and $G1$, $G3$ and between **??** and $G2, G4$ from Table **??**, although not quite as immediate as for standard multimodal structural rules, is relatively easy to show using a combination of the relevant rewrite and two contractions.

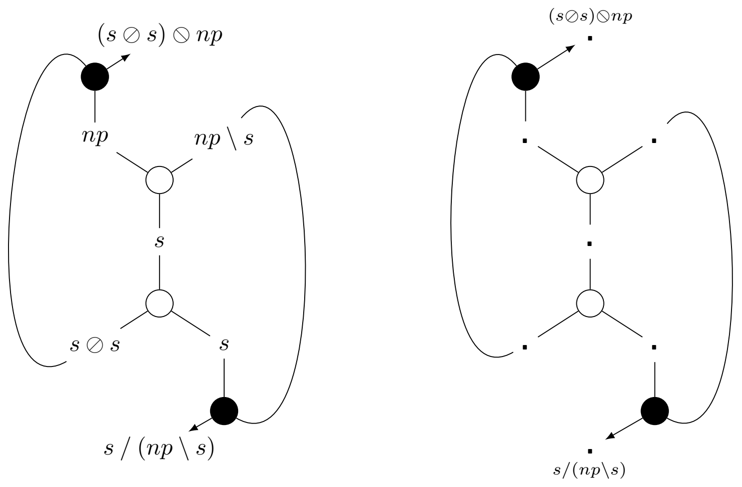

$(s \oslash s) \otimes np$
np
$np \setminus s$
s
$s \oslash s$
s
$s \,/\, (np \setminus s)$
$(s \oslash s) \otimes np$
$s/(np \setminus s)$

Figure 5.13: Proof structure of $(s \oslash s) \otimes np \vdash s \,/\, (np \setminus s)$ (left) and its corresponding abstract proof structure

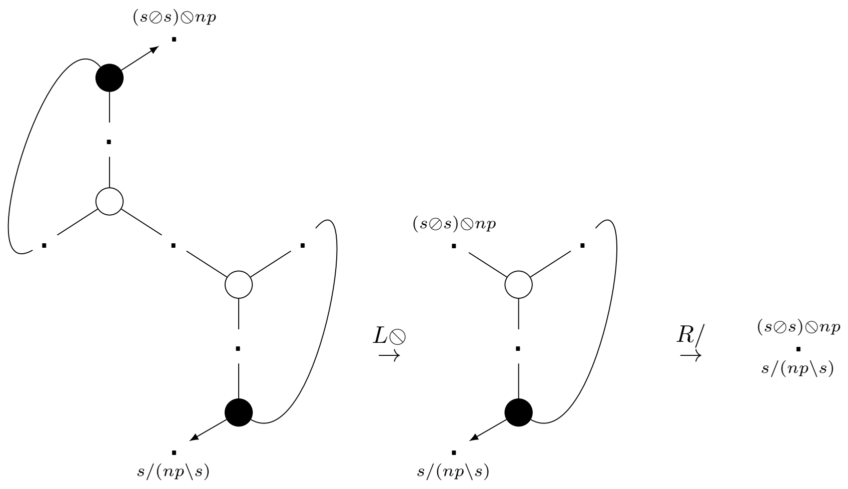

$(s \oslash s) \otimes np$
$s/(np \setminus s)$
$L\otimes$
$(s \oslash s) \otimes np$
$s/(np \setminus s)$
$R/$
$(s \oslash s) \otimes np$
$s/(np \setminus s)$

Figure 5.14: Applying the $G1$ conversion, and the $L\otimes$ and $R/$ contractions to the abstract proof structure of Figure **??**

This structure is in the right configuration for both the $L\oslash$ and the $R/$ contraction (the par links are connected by both tentacles without the arrow to a corresponding tensor link). Figure **??** shows we can contract the abstract proof structure to the tensor tree which corresponds to the following sequent.

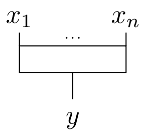
$$(s \oslash s) \oslash np \vdash s \,/\, (np \setminus s)$$

The proofs showing equivalence of the proof net calculus with the sequent calculus are easily adapted to the case of **LG** (**?**).

The Lambek-Grishin calculus is a type of classical logic, but only in the weak sense that it allows structures on both the left- and the right hand side of the turnstile. As a consequence, computing the derivational, lambda term meaning for proofs in the Lambek-Grishin calculus does not have the same simplicity as it has for the intuitionistic logics we have seen so far. However, we can use continuation based techniques to compute lambda terms from proofs in **LG** (**?**, **?**). Even though this complicates the way we obtain lambda terms from proofs, continuation based semantics gives us more flexibility as well. In addition, we obtain the same lambda term assignments from proof nets, although at the price of some technical complications (**?**, Section 3.2).

5.2.3 Associativity

As already noted by **?**, adding associativity to a multimodal system is done by allowing links with an arbitrary number of premisses, and replacing explicit rebracketing operations by a successive reduction of a binary tree with n leaves to a single link with n premisses of the form shown below.



We will call links of this form *combs*. Table 5.6 shows the contractions for an associative mode i (although, in general, most grammars will have only one associative mode, in which case we will not indicate the mode explicitly). These contractions look more involved than they are. For example, the associativity structure conversion Ass, takes two combs where the conclusion x_i of the first comb is a premiss of the second comb. It operates by replacing x_i by the premisses $y_1, \ldots, y_m$. The key points to observe are that this operation reduces the total number of links, and that it preserves the yield of the structure; successive application of the Ass rule will transform each tree with branches of mode i into a single comb with the leaves of this initial tree as its premisses. The $\bullet E$ contraction allows us to contract a $\bullet E$ par link whenever its two conclusions are adjacent premisses x_i, x_{i+1} of a single comb. Similarly, the $/I$ and $\setminus I$ contraction remove, respectively, the last and first premiss of a comb.

As usual, the contractions for $/I$ and $\setminus I$ have the side condition that the comb has at least one other premiss to avoid empty antecedent derivations.

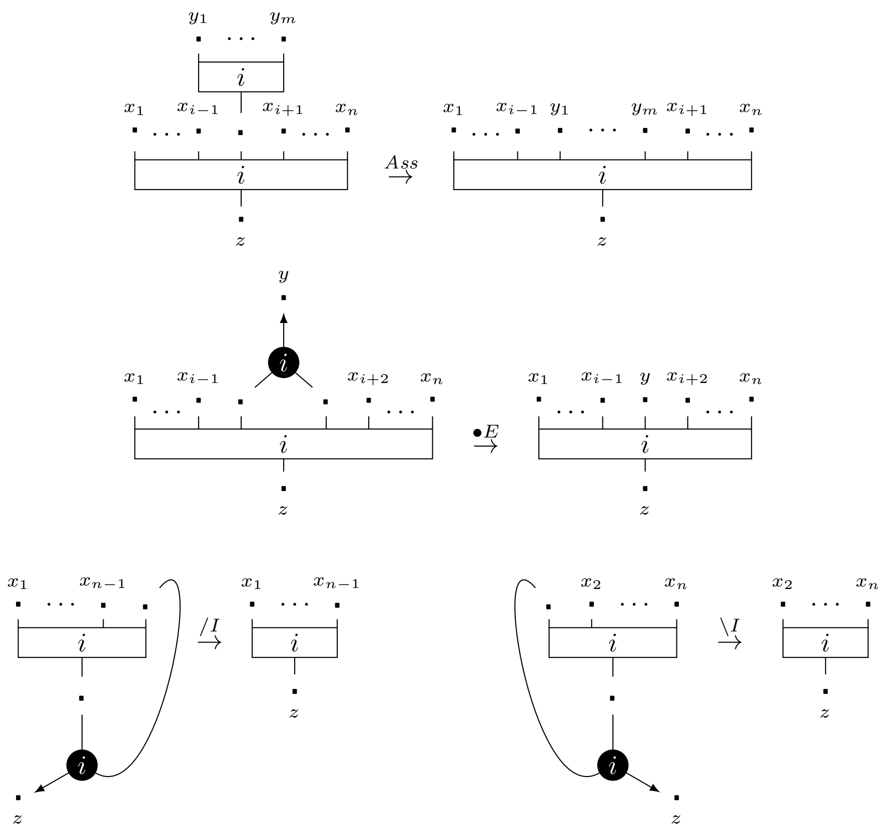


Table 5.8: Contractions under associativity for mode i

5.2.4 Wrap and the Displacement calculus

From a purely logical point of view, the Displacement calculus is a standard multimodal logic, and this is reflected at the level of the proof structures: we have the Lambek calculus connectives $/$, $\bullet$, $\backslash$ and corresponding structural connective '$+$' and the discontinuous connectives $\uparrow_k$, $\odot_k$, $\downarrow_k$ and corresponding structural connective '$\times_k$'. As discussed in Section 2.2, k is either the pair $>$, $<$ (denoting the wrap operation at, respectively, the leftmost and rightmost insertion point) or the set integers between 1 and the sort of the leftmost argument of $\times_k$ (denoting wrap at the kth insertion point). Given the discontinuous idiom "rang up" assigned to lexical term $rang + \mathbf{1} + up$ of sort 1, and the formula $(np\backslash s) \uparrow_> np$, and "everyone" assigned the standard Displacement calculus formula for quantifiers $(s \uparrow_> np) \downarrow_> s$, this lexicon produces the formula unfolding shown in Figure 5.13.

These are the standard multimodal formula unfoldings and we can connect the atomic formulas as usual to produce the proof structure shown on the left

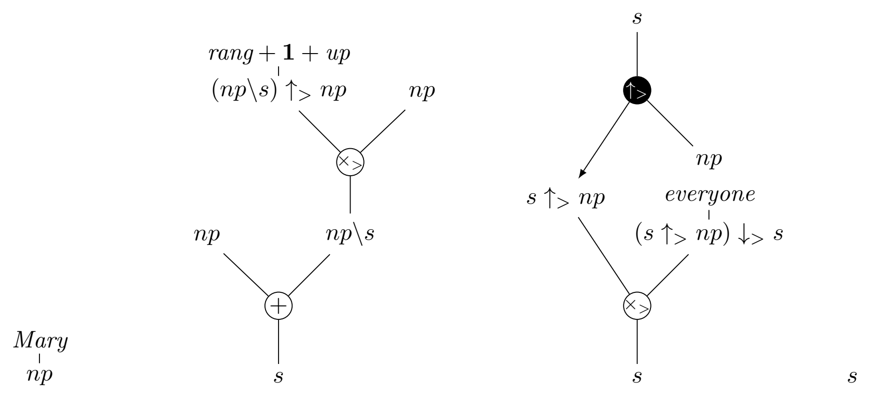


Figure 5.15: Unfolding for the sentence “Mary rang everyone up”.

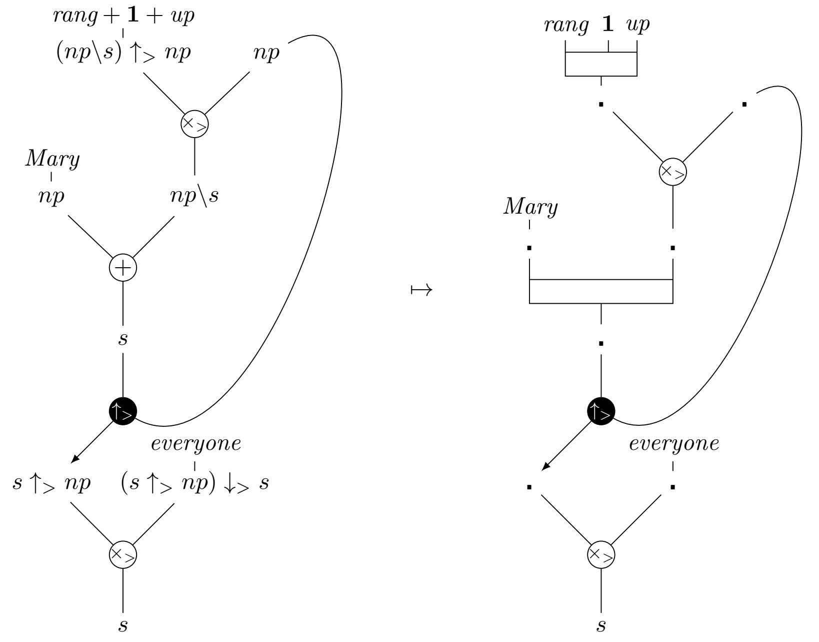


Figure 5.16: Proof structure (left) and abstract proof structure (right) for the unfolding of “Mary rang everyone up” shown in Figure 5.13.

of Figure 5.14. Where the Displacement calculus distinguishes itself from multimodal proof nets is at the level of the abstract proof structures. In the Displacement calculus, formulas are interpreted as expressions of the form $a_1 + \mathbf{1} + \ldots + \mathbf{1} + a_n$ (such as expression is of sort n, for $n \geq 0$, where n is the number of separator symbols ‘**1**’), corresponding to $(n+1)$-tuples of

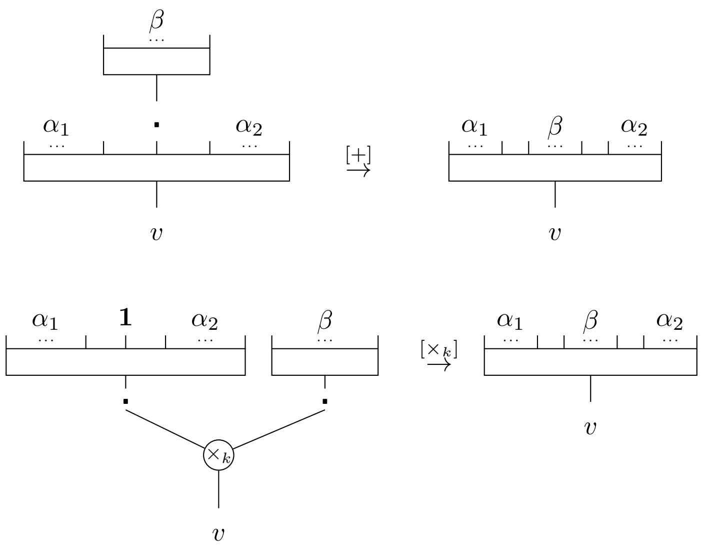


Table 5.9: Structural contractions for associativity and wrap

strings[11], and a wrap operator $\alpha \times_k \beta$ which replaces the kth occurrence of the separator symbol in α by β. What this means for the proof structure to abstract proof structure translation is that a vertex (formula) in the proof structure can correspond to a comb $a_1 + \mathbf{1} + \ldots + \mathbf{1} + a_n$ if that formula is of sort n. For our example, this is simple: only $\mathit{rang} + \mathbf{1} + \mathit{up}$ is replaced by a comb. We also replace the '+' links by two-premiss combs. This produces the abstract proof structure shown on the right of Figure 5.14[12].

For a correct proof net calculus, we need to provide the graph rewrites which correspond to the logical and structural rules of the Displacement calculus. Table 5.7 shows the structural rules for associativity and wrap on abstract proof structures. The associativity rules is the one we have see in the previous section (Figure 5.6) and the wrap structural rule $\times_k$ rewrites $W_k(\alpha_1 + \mathbf{1} + \alpha_2, \beta)$ to $\alpha_1 + \beta + \alpha_2$ when the separator symbol $\mathbf{1}$ is the kth one in the structure (in other words, when α_1 contains $k-1$ separator symbols; for '>', leftmost wrap, we require that α_1 contains no separator symbols, and for '<', rightmost wrap, we require that α_2 contains no separator symbols).

Table 5.8 shows the logical contractions of the Displacement calculus. They are a natural reflection of the natural deduction rules from Table 2.7. The extraction operator $\uparrow_k$ checks that the link is attached to a comb of the form $\alpha_1 + \beta + \alpha_2$ and produces a comb of the form $\alpha_1 + \mathbf{1} + \alpha_2$ (β can be of the

[11]To avoid cumbersome verbosity, for expressions of sort 0, we will use the term 'strings' instead of the more precise '1-tuples of strings'.

[12]If we want to be fully precise, the vertex "everyone" and the rightmost conclusion of the $\uparrow_>$ link (which is also the right premiss of the $\times_k$ link) should both be one-premiss combs. For convenience, we will treat these vertices as of sort 0 as one-premiss combs when applying the rewrite rules.

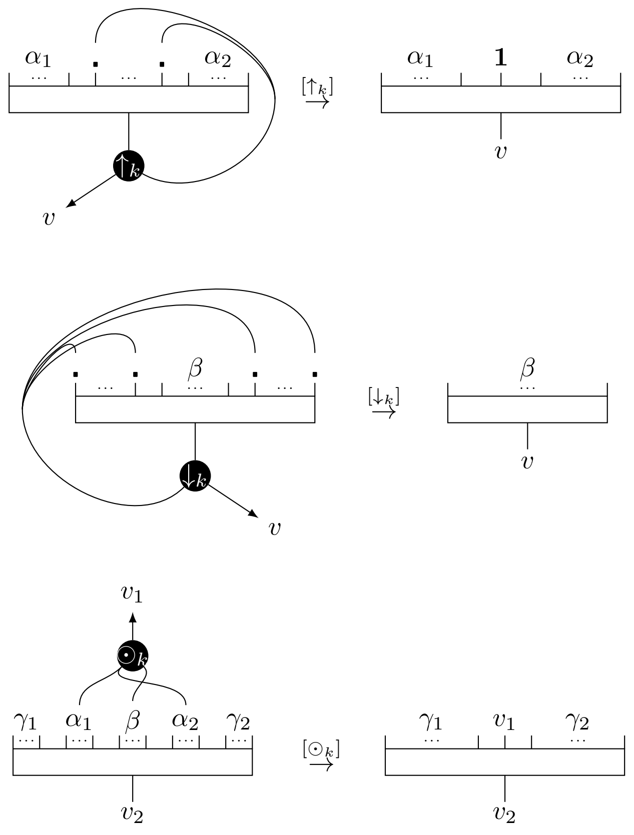


Table 5.10: Logical contractions for the discontinuous connectives

form $b_1 + \mathbf{1} + \ldots + \mathbf{1} + b_n$ according to the sort of the corresponding formula, and therefore multiple consecutive premisses of the comb can be removed by an $\uparrow_k$ step). The contraction has the standard condition that α_1 contains $k-1$ separator symbols. The net effect of the $\uparrow_k$ contraction is that it withdraws a hypothesis of the appropriate sort and leaves a separator symbol at the place of this hypothesis, just like the $\uparrow_k I$ rule.

The contraction for $\downarrow_k$ removes a (possibly complex) circumfix of the form $\alpha_1 + \mathbf{1} + \alpha_2$ from around β (in the same way as the $\downarrow_k I$ rule), whereas the $\odot_k$ rule replaces premisses of the form $\alpha_1 + \beta + \alpha_2$ by the structure at v_1 (that is, the structure computed for the corresponding product formula $A \odot_k B$, just like the $\odot_k E$ rule).

Because of the potentially many string segments for each formula in the proof structure, the logical contractions look a bit intimidating. In practice, however, the number of string segments is generally small: the maximum used by **?** is three segments (and therefore two separator symbols).

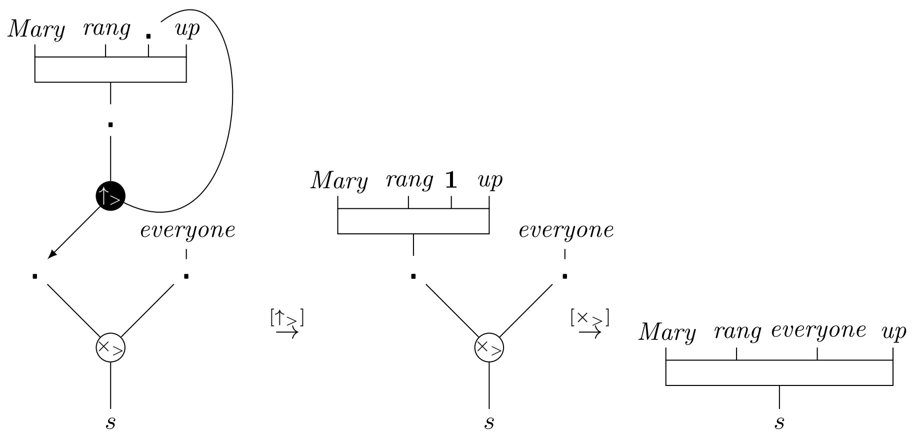


Figure 5.17: Contractions for the abstract proof structure for the sentence "Mary rang everyone up" shown on the right of Figure 5.14.

As an illustration, Figure 5.15 shows how the structural and logical contraction interaction for the abstract proof structure of Figure 5.14. We start with the $\times_>$ contraction, which replaces the leftmost (and only) occurrence of $\mathbf{1}$ in the comb of *rang*$+\mathbf{1}+$*up* by the vertex which is the rightmost conclusion of the $\uparrow_>$ link (corresponding to the hypothetical noun phrase). Applying the associativity rule produces the structural shown on the left of Figure 5.15. We then apply the $\uparrow_>$ contraction to withdraw this hypothetical *np* and leave a separator symbol at its place. Finally, we apply the wrap operation to replace the separator symbol by "everyone", thereby showing we have a proof net of "Mary rang everyone up".

With respect to the multimodal proof nets, the main technical complication is that formulas can correspond to multiple string segments. However, we can apply all of the standard machinery to prove the proof net calculus produces exactly the same proofs as natural deduction (**?**).

While it is a drawback of the graph rewriting calculus that confluence never holds automatically, it does hold for the fragment of the Displacement calculus containing only the binary multiplicative connectives (the residuated triples $/$, $\bullet$, $\backslash$, and $\uparrow_k$, $\odot_k$, $\downarrow_k$).

As an advantage over the translation of the Displacement calculus into first-order linear logic, the current translation can be unproblematically extended to other connectives, such as the 'split' connective $^\vee$. However, adding the split connective means we lose confluence: the contraction for this connective, like the corresponding logical rule, allows us to insert a separation marker anywhere (provided it is the kth separation symbol in the result). Starting with a hypothesis $a : C$ the split rule allows us to continue in the following two ways[13], with the terms $a + \mathbf{1}$ and $\mathbf{1} + a$ behaving differently in most contexts.

$$\frac{a : C}{\mathbf{1} + a : {}^{\vee}C}\ {}^{\vee}I \qquad\qquad \frac{a : C}{a + \mathbf{1} : {}^{\vee}C}\ {}^{\vee}I \qquad\qquad \textit{units}$$

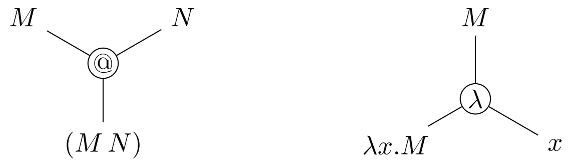
So while the graph rewriting perspective allows us to handle a larger fragment of the Displacement calculus, the price of this larger fragment is a loss of confluence in the rewrite system[14].

5.2.5 Beta-reduction, hybrid grammars and $\mathbf{NL}_\lambda$

Hybrid type-logical grammars combine the Lambek calculus with lambda grammars. For the Lambek calculus part of the logic, we can simply use the standard links and contractions for associativity. The logical links for the linear implication '$\multimap$' are not a problem either. Where hybrid type-logical grammars (and lambda grammars/abstract categorial grammars) change from other type-logical grammars is that they allow linear lambda terms in their structures.

Linear lambda terms and abstract proof structures

Given that abstract proof structures reflect the structure of the logic we are using, this means we need to find a way to add application and abstraction to abstract proof structures, but also a way to perform beta/eta reduction. However, this is not very complicated: there are many ways to implement lambda terms unambiguously — trees with back pointers, hypergraphs (**?**, **?**, **?**). Given that our abstract proof structures are hypergraphs, we choose to use a graph-like representation for (linear) lambda terms, with application represented by a binary term constructor '@' and lambda abstraction by a tensor link labeld λ with a single premiss and two conclusions. The intended meaning of this link is that the premiss of the link can correspond to any term M, that the rightmost conclusion corresponds to the abstracted variable x and that the leftmost conclusion corresponds to $\lambda x.M$. The labeled versions of the links for application and abstraction are shown below. Note, however, that the labels are not part of the structure and are only presented to indicate how linear lambda terms are translated into abstract proof structures.



[13]The problem is not restricted to forward chaining proof search. Backward chaining proof search has the same problem with the ${}^{\vee}E$ rule.

[14]Of course, when a rewrite system is not confluent, the immediate question we should ask is: can we reformulate it in such a way that it becomes confluent? However, I am not sure such a reformulation is possible for '${}^{\vee}$'.

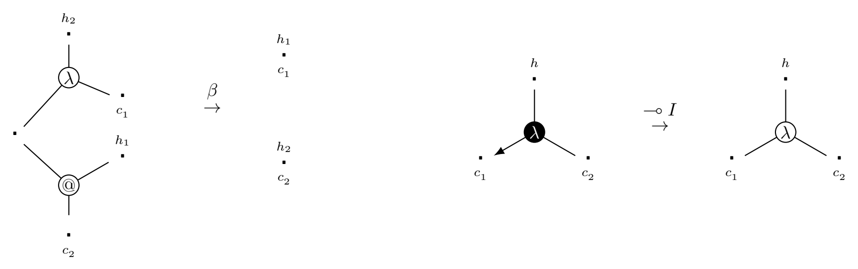


Table 5.11: The graph conversions for hybrid type-logical grammars

The tensor links in abstract proof structure represent the allowed structures in our logic. In hybrid type-logical grammars (as in lambda grammars) these structural can contain (linear) lambda terms, and therefore the lambda abstraction link 'λ' and the application link '@' are both tensor links. However, this also means the structures are no longer necessarily trees. For our correctness condition, we want to contract to a structure representing a linear lambda term. One way to enforce this is as follows.

1. the abstract proof structure should be a tree when we remove all branches between λ links and their rightmost conclusion (corresponding to the abstracted variable),

2. this tree should have a path from the rightmost conclusion of a λ link to its premiss.

We will call abstract proof structures satisfying these conditions *lambda graphs*. The conditions implement the essential net conditions of **?**, but can also be seen as a way of interpreting our hypergraph-theoretic representation of lambda terms as a tree with backpointers from the bound variables to their binding lambda abstraction. Together with the standard conditions on proof structures (restricting each node to be at most once a premiss and at most once a conclusion), these conditions ensure each variable is bound exactly once. In other words, lambda graphs represent linear lambda terms as abstract proof structures.

Proof nets for hybrid type-logical grammars

To obtain a proof net calculus for hybrid type-logical grammars, all that is missing are the graph contractions and structural rewrite operations. Table **??** shows the graph conversions which are added to that 'standard' conversions of associativity and the Lambek calculus contractions to obtain proof nets for hybrid type-logical grammars. Both contractions look a bit unusual and require some comments.

The $\multimap I$ contraction is not of the 'standard' form for residuated connectives, where a par link is connected to a tensor link and both are removed.

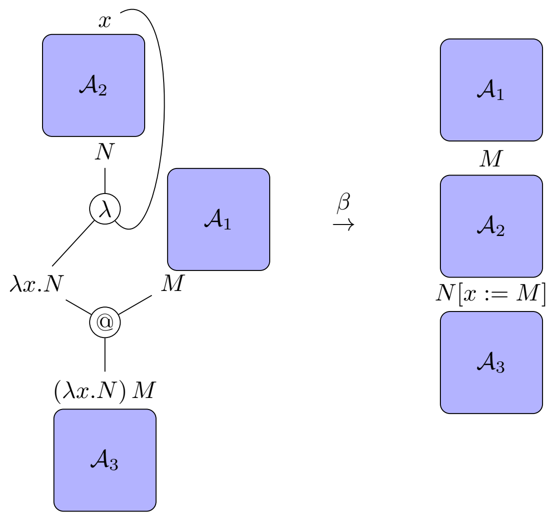


Figure 5.18: Beta conversion as a structural rule with term labels added.

This is because the residuated par links verify the structure is of the required form for the contraction. The $\multimap I$ rule, repeated below, does not have such a required on the structure for rule application.

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B} \multimap I$$

In the proof net calculus, we therefore simply replace the par link for the $\multimap I$ rule with a lambda tensor link. This has the net effect of adding an abstraction to the structure, which is the desired effect for the $\multimap I$ rule. However, we need to be careful. To avoid accidental capture of variables, correct application of the $\multimap$ contraction requires that c_2 is a descendant of h, through a path which does not pass through any par links. This amounts to verifying the structure produced by the $\multimap I$ contraction represents a linear lambda term.

The β contraction, as its name suggests, is the reflection of (linear) beta reduction in abstract proof structures. The term-labeled version of the β contraction shown in Figure 5.16 makes the correspondence to beta reduction clear. Note that β reduction defined this way is correct only because of the condition on the $\multimap I$ rule, which ensures that term N has exactly one occurrence of variable x.

With this in place, it is fairly easy to show that the proof net calculus for hybrid type-logical grammars corresponds exactly to its natural deduction calculus, and that it satisfies cut elimination. In addition, the rewrite rules for abstract proof structures are confluent (**?**, **?**)

To illustrate the usefulness of the calculus, **?** propose the formula $(tv \multimap s) \multimap (tv \multimap s) \multimap tv \multimap s$ with prosodic term $\lambda Q.\lambda P.\lambda v.(P\,v) + and + (Q\,\epsilon)$ for the coordinator "and" in gapping constructions (where tv is short for $(np\backslash s)/np$). The idea of this lexical entry is that it selects two sentences, each missing a transitive verb tv, then selects a transitive verb and inserts it in the leftmost sentence, whereas in the rightmost sentence the missing transitive verb is assigned the empty string at the term level. The advantage of this type of analysis is that it is now easy to get the desired semantics of the sentence.

Looking at this lexical entry in terms of proof nets, the abstract proof structure corresponding to this formula and its assigned term is shown in Figure 5.17 (the occurrences of tv have not been unfolded). The three λ links correspond to the abstractions over Q (corresponding to the rightmost sentence missing a transitive verb), P (corresponding to the leftmost one) and v (corresponding to the transitive verb). This is again just a graphical way to represent the lambda term assigned to the lexical entry. The unfolding of the lexical formula and its prosodic term has produced an abstract proof structure which can be further reduced by beta reduction — this is a fairly standard partial evaluation strategy on proof nets, which in the context of type-logical grammars is often applied on the meaning level (**?**, **?**). In the context of proof nets for hybrid type-logical grammars, we can generally simplify the lexical abstract proof structures like this (**?**, Section 4.7) After performing the three beta reductions, we obtain the abstract proof structure shown on the right of Figure 5.17.

We can combine this abstract proof structure with the standard assignments for "John", "logic", "Charles", "phonetics" (all assigned np in the lexicon) and "studies" of type $tv = (np\backslash s)/np$ to produce the abstract proof structure shown on the left of Figure **??**. The transitive verb "studies" has been connected to the tv hypothesis of "and"[15], whereas the two tv conclusions of "and" have been unfolded and combined with the four lexical noun phrases. For the $\multimap I$ contractions, the paths starting at the bent edges end up back at the root, thereby verifying that the par links labeled λ are valid linear lambda terms when we turn them into tensor links: they correspond to $\lambda x.John + (x + logic)$ and $\lambda y.Charles + (y + phonetics)$ respectively, where in both cases the abstracted variable represents the position of the missing transitive verb. The right hand side of Figure **??** is a graphical representation of the following lambda term.

$$((\lambda x.John + (x + logic))\ studies) + and + ((\lambda y.Charles + (y + phonetics))\ \epsilon)$$

We can then reduce the two beta redexes, replacing x by *studies* and y by the empty string ϵ, and remove the empty string to produce the desired string.

$$(John + (studies + logic)) + (and + (Charles + phonetics))$$

[15]To avoid making the structure overly complicated, neither "studies" nor the tv hypothesis with which it is matched have been unfolded.

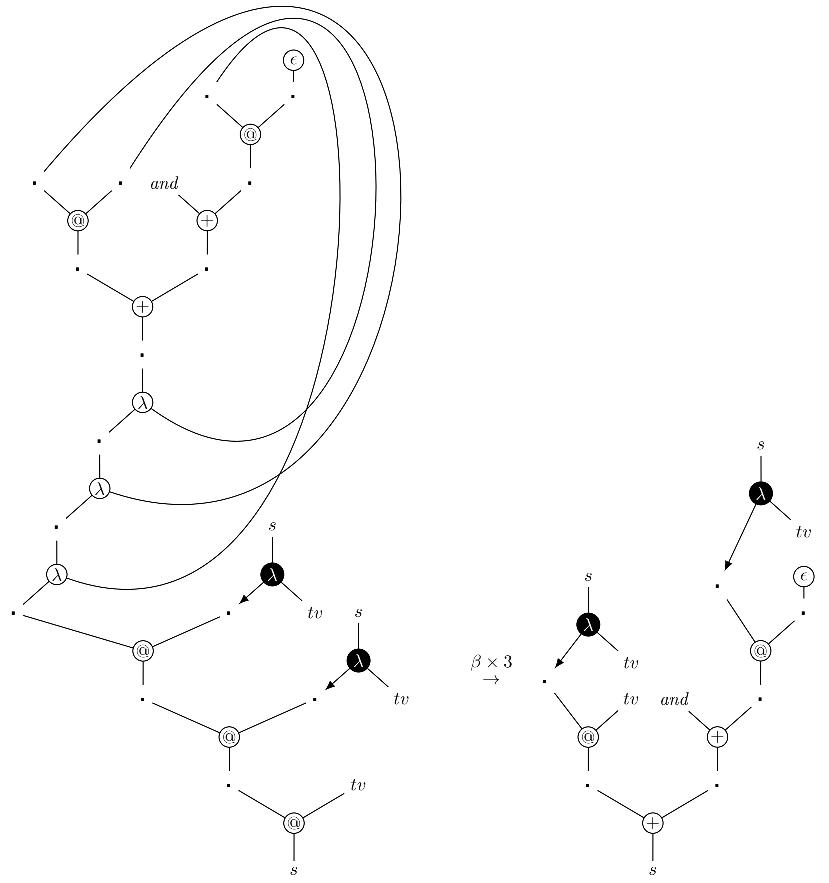


Figure 5.19: Abstract proof structure of the lexical entry for gapping from **?**, before and after β reductions.

The logic of scope

As discussed in Section 2.5, the logical rules of the logic of scope (**?**, **?**) are those of a standard multimodal logic with two modes, and corresponding binary tree constructors $\circ$ and $\circ_w$. This second constructor is the scope-taking or discontinuous constructor, and its behaviour is controlled by the following structural rules.

$$\frac{\Xi[\Gamma[\Delta]] \vdash C}{\Xi[\Delta \circ_w \lambda x.\Gamma[x]] \vdash C}\ \lambda \qquad \frac{\Xi[\Delta \circ_w \lambda x.\Gamma[x]] \vdash C}{\Xi[\Gamma[\Delta]] \vdash C}\ \lambda^{-1}$$

The structural rules have the condition that each x used in the rules is unique in the proof. The structural rules are clearly inspired by beta reduction and expansion in the linear lambda calculus.

$$\Delta \circ_w \lambda x.\Gamma[x] \leftrightarrow \Gamma[\Delta] \quad \approx \quad ((\lambda x.M[x])\, N) \equiv_\beta M[N]$$

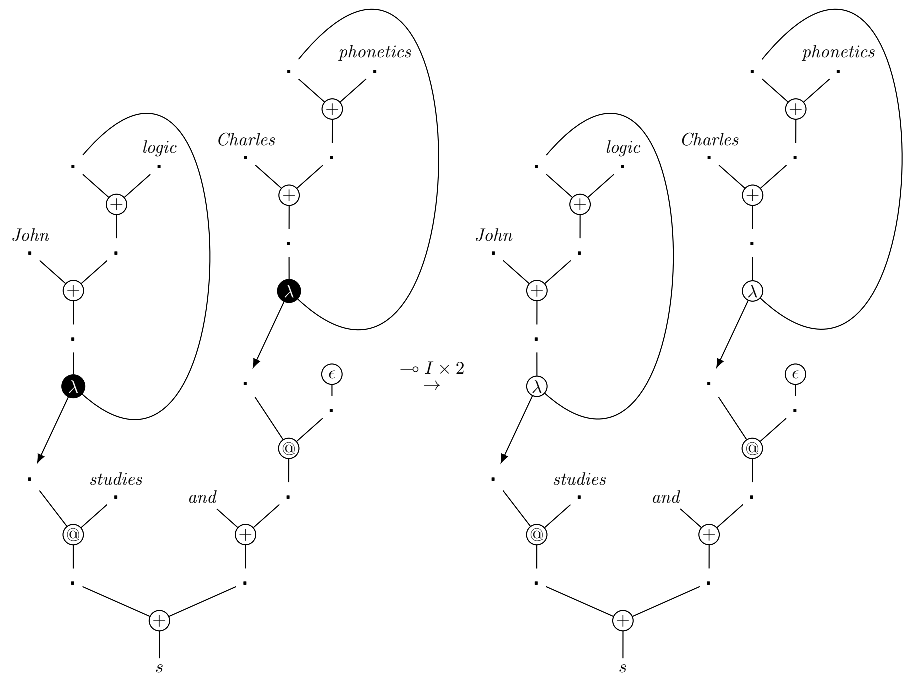


Figure 5.20: Abstract proof structure for "John studies logic and Charles, phonetics"

Given the preceding, it is then simple to provide a proof net calculus for $\mathbf{NL}_\lambda$: we have the standard multimodal contractions for the two binary modes and the structural rules given in Table **??**. Similar to the proof nets for hybrid type-logical grammars, we need the side condition that the node labeled c_1 is a descendant of the node labeled h_2 by a path not passing through par links. This again ensures our graphs represent linear lambda terms. Another way to see this condition is that it ensures that we maintain the property that structural rewrites always operate in the same component. Note also that the β conversion of Table **??** is left-right symmetric with the one from Table **??**.

We can use an inductive proof similar to those of **?** and **?** to show that the proof net calculus defined this way generates the same theorems as the sequent calculus for $\mathbf{NL}_\lambda$ (**?**, Section 3.4). However, some of the other properties of the calculus are not so clear. For example, the $\backslash_w I$ (or $\backslash_w R$) rule can remove a structure which corresponds to a 'beta redex' as shown below, and this raises questions about confluence of the operations.

$$\frac{B \circ_w \lambda y.\Gamma[y] \vdash A}{\lambda y.\Gamma[y] \vdash B \backslash_w A} \backslash_w I$$

Another potential problem is that the expansion rule β^{-1} increases the size of the structure. However, once we remove the 'detours' created by combinations of β^{-1} with β (which we can always eliminate), we can show that in $\mathbf{NL}_\lambda$ all

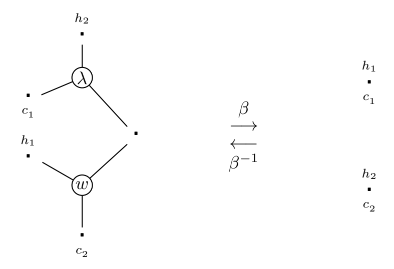


Table 5.12: The graph conversions for $\mathbf{NL}_\lambda$

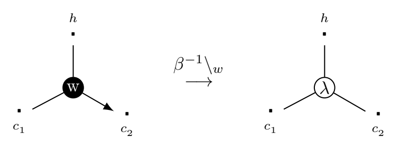


Table 5.13: Derived rule for $\mathbf{NL}_\lambda$.

useful applications of the β^{-1} rule must be immediately followed by a $\backslash_w I$ rule (**?**, Section 2.2) (**?**, Section 4). In terms of natural deduction proofs, this means the β^{-1} rule only occurs in the following context, where it extracts a B formula from any position in the antecedent while 'marking' the place it came from by an abstracted variable y.

$$\dfrac{\dfrac{\Gamma[B] \vdash A}{B \circ_w \lambda y.\Gamma[y] \vdash A}\ \beta^{-1}}{\lambda y.\Gamma[y] \vdash B \backslash_w A}\ \backslash_w I$$

This means we can replace the β^{-1} rewrite by the derived rules shown in Table **??**. This derived rule inherits the condition from the β^{-1} rule that c_2 should be a descendant of h in a path not passing through par links.

This derived rule is just the left-right symmetric rule of the contraction for $\multimap I$ we have seen for proof nets in hybrid type-logical grammars in Table **??**, with the same side condition to guarantee the result represents a linear lambda term, so this is again a case of diverging logical paths which arrive at the same destination.

The similarity between $\mathbf{NL}_\lambda$ and hybrid type-logical grammars goes further that this, as illustrated by Table **??**. In many case, we have structures which are isomorphic (up to some trivial left-right symmetries), *and* the same graph rewrites which can apply to them. There is, of course, the difference that HTLG (in its standard formulation) is associative, whereas $\mathbf{NL}_\lambda$ (also in its standard formulation) is not, but this is easily harmonised if desired (by adding/removing the associativity structural rules from the calculus), or even ignored when it is irrelevant for the particular linguistic analysis.

HTLG		$\mathbf{NL}_\lambda$
$+$ link	$\leftrightarrow$	$\circ$ link
@ with premisses $p_1 - p_2$	$\leftrightarrow$	$\circ_w$ with premisses $p_2 - p_1$
λ tensor (lexicon)		???
$\multimap I$ par with conclusions $c_1 - c_2$	$\leftrightarrow$	$\backslash_w$ par with conclusions $c_2 - c_1$
???		$/_w$, $\bullet_w$ par links
contractions for $/$, $\backslash$	$\leftrightarrow$	contractions for $/$, $\backslash$
???		contraction for $\bullet$
$\multimap I$ par rewrite	$\leftrightarrow$	$\beta^{-1}\backslash_w$ rewrite
β rewrite	$\leftrightarrow$	β rewrite
η rewrite	$\leftrightarrow$	contraction for $\backslash_w$
???		contractions for $/_w$, $\bullet_w$

Table 5.14: Translations between HTLG and $\mathbf{NL}_\lambda$.

While Table **??** has a number of 'holes', indicated by '???', where a construction in one calculus does not have a clear equivalent in the other one, a number of key linguistic analyses can be translated between the two formalisms without problem.

For example, take the lexical assignment for gapping shown, in its beta reduced form, on the right of Figure 5.17. The corresponding $\mathbf{NL}_\lambda$ formula is $((tv \bullet_w (tv \backslash_w s))\backslash s)/(t \bullet_w (tv \backslash_w s))$, where t is the formula equivalent of the empty string in $\mathbf{NL}_\lambda$, and tv is short for the transitive verb type $(np\backslash s)/np$ as usual[16].

Inversely, to get the correct reading of sentence **??** below, **?** propose the formula $(np \backslash_w s) /_w ((n/n) \backslash_w (np \backslash_w s))$ for "same".

(2) Everyone read the same book.

The key property of sentence **??** that **?** want to capture is that it has an $\exists\forall$ reading but no $\forall\exists$ reading (where every student reads a potentially different book). To understand the intuition behind the type assignment, we must remember that quantifiers in $\mathbf{NL}_\lambda$ are assigned the formula $s /_w (np \backslash_w s)$, which moves a noun phrase out of a sentences and then inserts the quantifier formula in the place of the extracted noun phrase (as shown in Section 2.5). The formula for "same" selects as its argument a sentence s missing both an adjective n/n and a noun phrase np (while keeping track of their positions). The resulting $np \backslash_w s$ formula then serves as the argument of the quantifier. This has the net effect of the quantifier taking the place of the extracted noun phrase and "same" the place of the extracted adjective (**?**, Appendix C gives a proof net derivation for this example in full detail).

[16]This formula looks a bit odd and unlike other formulas for coordination, however there is some precedent for using these types of formulas (**?**, Section 4.3.4, uses a similar formula), and from the perspective of linear logic the formula $(tv \multimap s) \multimap (tv \otimes (tv \multimap s)) \multimap s$ has a Curried version $(tv \multimap s) \multimap (tv \multimap s) \multimap tv \multimap s$ which is a standard coordination formula. Note, however, that although this Currying operation is valid in multiplicative linear logic, it is not valid in $\mathbf{NL}_\lambda$ for the given gapping formula.

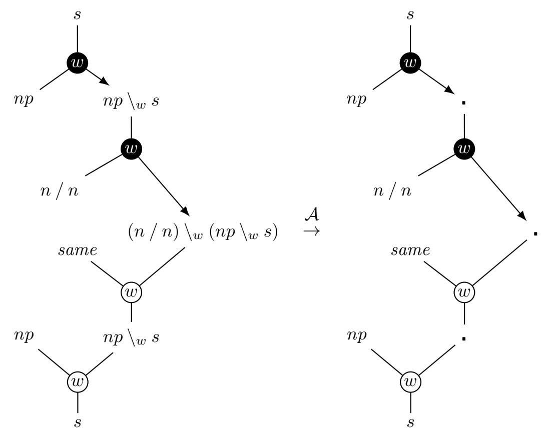


Figure 5.21: Proof structure and abstract proof structure of the word "same" according to the analysis of Barker & Shan

Figure **??** shows the proof structure and abstract proof structure of the formula unfolding for "same". To translate this lexical entry to hybrid type-logical grammars, we need to find an abstract proof structure which, using beta reduction, reduces to a graph which is isomorphic to the abstract proof structure on the right of Figure **??**. This requires a bit of puzzling. First, we translate the $\mathbf{NL}_\lambda$ formula into its hybrid counterpart, replacing both $B \backslash_w A$ and $A /_w B$ by $A \multimap B$, which produces $((n/n) \multimap (np \multimap s)) \multimap (np \multimap s)$. Given this formula, the string term of this lexical entry must be of type $(s \to s \to s) \to s \to s$. In this case, the correct term is $\lambda P \lambda x.((P\, same)\, x)$. We verify this assignment translates the analysis of **?** into hybrid type-logical grammar by unfolding the resulting lexical entry, then applying the two β reductions. Figure **??** shows the result, and we can see that the partially evaluated abstract proof structure on the right of Figure **??** is simply the left-right mirror image of the one on the right of Figure **??**.

? propose a different analysis of "same" and "different" in hybrid type-logical grammars. However, the current translation of the analysis of **?** into hybrid type-logical grammar facilitates a comparison between these different approaches (**?**, Section 5.4.1.2, discuss some empirical differences between the two analyses).

To conclude this section, although it would appear that representing lambda terms by hypergraphs is cumbersome both from the typesetting perspective and for the amount of space it takes with respect to the standard, flat representation, we should not automatically identify notation which is easy to read by humans with notation which is easy for computer manipulation or for proving meta-theorems (**?**). In the current case, our principal interest has been the comparison between different type-logical grammars by casting all of them as instances of a generic logical calculus.

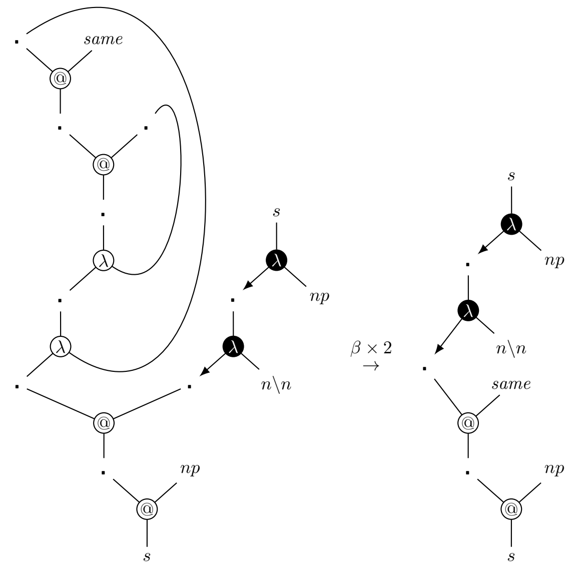


Figure 5.22: The analysis of “same” from Figure **??** translated into HTLG.

5.3 Discussion

We have seen how the ‘proofs nets as graph rewriting’ perspective provides an alternative analysis of modern type-logical grammars. A number of type-logical grammars for which currently no translation into first-order linear logic exists can be given a proof net calculus using this methodology. This includes the Lambek-Grishin calculus and $\mathbf{NL}_\lambda$, but also the extended versions of the Displacement calculus (notably including the split connective, but also the unary connectives of multimodal grammars) and extended versions of hybrid type-logical grammars (for example the multimodal version of hybrid type-logical grammars from **?**). Taken together, this means the proof net calculus is general enough to handle *all* modern type-logical grammars.

The similarity between the Displacement calculus on the one hand, and hybrid type-logical grammars and $\mathbf{NL}_\lambda$ on the other is that the ‘insertion points’ and wrap operation of the former play a similar role as the abstraction and beta-reduction operation of the latter, as indicated below.

$$W(\alpha_1 + \mathbf{1} + \alpha_2, \beta) \equiv \alpha_1 + \beta + \alpha_2 \quad \approx \quad ((\lambda x.\alpha_1 + x + \alpha_2)\,\beta) \equiv \alpha_1 + \beta + \alpha_2$$

While this similarity is rather approximative it does allow us give some translations between rules (and proofs) of the Displacement calculus and those of hybrid type-logical grammars. But this approximation is fragile at a number of points: the Displacement calculus refers to multiple insertion points by their linear order in the resulting output, something which appears antithetical to the lambda calculus, where only the relative order of the lambda abstractions is relevant. One the other hand, higher order lambda calculus terms have no clear correspondence in the Displacement calculus.

From the graph rewriting perspective the correspondance between the analyses of the Displacement calculus and hybrid type-logical grammars is therefore not quite as immediate as it was in first-order linear logic (Section **??**). However, the graph rewriting perspective has shown us a number of points of comparison between hybrid type-logical grammas and $\mathbf{NL}_\lambda$. So thanks to first-order linear logic, we know that hybrid type-logical grammars agree with the Displacement calculus for their respective analyses of quantifiers, gapping and other phenomena. And thanks to the graph rewriting perspective, we know that hybrid type-logical grammars agree with $\mathbf{NL}_\lambda$ for *their* analyses of quantifiers, gapping and other phenomena as well (although this time modulo associativity).

This suggests there is a 'common core' of phenomena on the syntax-semantics interface which are treated by most modern type-logical grammars, but it also points the way to challenge formalisms by providing analyses in one formalism which have no translations in the others, which can result in lively (and healthy) debates about the relative advantages of different formalisms (**?**, **?**) (we have already seen some of these comparisons in Sections **??** and **??**).

An important missing point for the graph rewriting perspective is that the current implementations (**?**, **?**) only handle the multimodal version of proof nets, and would need a significant overhaul to handle all of the more general cases presented in this chapter.

6 Wide-Coverage Semantics

This chapter presents my work on using type-logical grammars for wide-coverage semantics. The section on neural proof nets, Section **??**, is based on joint work with Konstantinos Kogkalidis and Michael Moortgat (**?**).

6.1 Introduction: natural language understanding in the age of deep learning

Until fairly recently (**?**, **?**, **?**, **?**), systems for natural language understanding had a more-or-less predictable structure: there was a part-of-speech tagger, a parser, a semantic representation language and a reasoning engine. And although many of these components were at least partly statistical or based on machine learning techniques, there were generally some intermediate structures (such as a parse tree or a logical representation of the meaning of a text) we could inspect.

Recent advances in deep learning, however, have managed to obtain incredibly impressive scores on a wide variety of tasks, such as the GLUE and SuperGLUE benchmarks (**?**, **?**). The GLUE and SuperGLUE benchmarks include many of the standard tasks in natural language understanding: anaphora/co-reference resolution, entailment, question answering. Modern language models such as GPT3 (**?**) and the various BERT systems (**?**, **?**, **?**) obtain remarkable results on each of these tasks, using an end-to-end machine learning architecture. That is, there is no recognisable parser, no recognisable meaning representation, no explicit inference engine. Yet, for many tasks the performance can be classified as superhuman[1]. Are the parser and the formal meaning component really superfluous? Or are they implicitly present in the billions of parameters of BERT and GPT3? These general-purpose language models obviously capture *some* knowledge of the language they model, but, although it is hard to determine what is and isn't represented in their many parameters, the evidence seems clear that a lot of basic knowledge is missing.

One of the great strengths of deep learning models (and machine learning models in general) is exploiting patterns in the data. And one of the main worries of developers of deep learning applications is that their models do not generalise outside the training data (this is the standard problem of *overfitting* in machine learning). To reduce overfitting, the original data is normally split into a test set, a development set and a training set, with the test set kept under lock and key until the model has been finalised using the training and development sets. The final model is then evaluated against the test set. Given that the test set contains data which have not been seen up until this point, the model performance on the test set gives a fair evaluation of the performance on unseen data. However, we should be careful about two things. First, when we do too many evaluations against the test set, the test set can no longer be considered to be truly unseen. Second, the original datasets (the training, development and test sets) are likely to be more similar to each other than to other data we would find 'in the wild'. This is why we should not expect a machine learning model trained on a journalistic dataset such as the Penn Treebank or the French Treebank to perform as well on other texts, such as social media posts. The holy grail of machine learning is for the algorithm to learn *exactly* the right patterns in the input data. Inversely, the greatest worry of a machine learning developer is that the models will instead learn a statistical shortcut which doesn't generalise on truly unseen data.

A number of authors have taken models which performed well on some of the standard benchmarks (entailment, question answering, etc.) and have tried to make them fail on slight modifications of the data. As a first type of test, **?** test a large number of question-answering systems as follows. The take a standard question-answer pair from the data set, such as the following.

Paragraph	Peyton Manning became the first quarter-back ever to lead two different teams to multiple Super Bowls. He is also the oldest quarterback ever to play in a Super Bowl at age 39. The past record was held by John Elway, who led the Broncos to victory in Super Bowl XXXIII at age 38 and is currently Denver's Executive Vice President of Football Operations and General Manager.
Question	What is the name of the quarterback who was 38 at Super Bowl XXXIII
Prediction	John Elway

[1] We should of course be very skeptical about classifying AI performance as superhuman. As we will see, just like humans, these machine learning systems have blind spots, they just have different blind spots than humans do.

The manipulation performed by **?** is to add an irrelevant sentence (at least as far as the question is concerned) such as "Quarterback Jeff Dean had jersey number 37 in Champ Bowl XXXIV" at the end of the input paragraph and evaluate the models on these new data. This changes the prediction to "Jeff Dean". Some clever additional test are performed: models are retrained on the new dataset, but then simply learn to ignore the last sentence and fail on the original dataset.

In a second type of test, **?** show that it is fairly easy to generate adversarial examples for textual entailment systems as well. They show that heuristics such as lexical overlap and subsequence can be used to entice textual entailment systems to give incorrect responses like the following.

(1) a. The doctor was paid by the actor
  b. The doctor paid the actor (not entailed from **??**, but incorrectly predicted as entailed)

(2) a. The doctor near the actor danced
  b. The actor danced (not entailed from **??**, but incorrectly predicted as entailed)

? have directly tested for overfitting by presenting entailment tasks with only the hypothesis and not the text from which we need to conclude whether the hypothesis is entailed, contradicted or neither. This means that for the entailment examples above, the model is given only the *b* sentences but not the *a* sentences[2]. Surprisingly, these models perform quite well, exploiting annotation artefacts like the fact that the presence of "no" or "nobody" is a fairly reliable indicator of 'contradiction'. Another indicator of contradiction is "sleeping" because it contradicts most other physical activities. **?** do similar experiments for the analysis of natural language arguments and their conclusions are harsh: for their dataset, "the entirety of BERT's performance can be accounted for in terms of exploiting spurious statistical cues".

As a fourth type of test, **?** and **?** show that natural language inference models perform poorly on inferences requiring lexical knowledge, as found in ontologies such as WordNet or JeuxDeMots (**?**, **?**). Performance is especially poor for unseen antonyms, and categories such as planets, countries, nationalities and ordinals.

Finally, **?** show that many of the results of natural language understanding systems are more or less invariant under random permutations.

None of this is supposed to belittle the impressive progress made on natural language understanding tasks, but it should make us wary of claims of superhuman performance. Adherents of the end-to-end machine learning approach believe that these problems point to the need for better datasets (and

[2]Logicians will surely note that we *can* make some reliable inferences in these cases: if *b* is a logical contradiction, then 'contradiction' is justified, whereas if *b* is a tautology, then 'entailment' is justified. However, such cases do not generally appear in entailment datasets, and even if they did, it think it would be good to at least warn the user when either of these is the case.

indeed SuperGLUE was developed at least in part with this purpose in mind) or maybe better machine learning algorithms.

Only the reader for whom this is the first chapter — in that case: welcome, no offence taken — will doubt that I am more on the side of good old-fashioned AI. While I believe that machine learning is a useful tool in many cases, I also believe that using machine learning is in many ways an admittance of defeat: it shows that we have failed to understand the problem in a way which allows us to explain it to a computer, or, more alternatively, that we have failed to find a computationally feasible algorithm. What I believe the experiments mentioned above show is that even current state-of-the-art deep learning systems suffer from a lack of explicit syntactic analysis, a lack of explicit meaning representation, and a lack of explicit world knowledge. However, I acknowledge that a significant research effort is required to bring such a natural language understanding system up to the level of the best performing deep learning systems. There is probably no panacea: whereas deep learning systems are robust in some areas but fragile in others, classic systems augmented with deep learning are likely to be robust and fragile in different areas.

Another important topic in machine learning is explainability: we not only want our machine learning system to provide the correct response, but also an *explanation* as to why this is the correct response, allowing humans to verify the soundness of the reasoning. This is especially true in high-stakes domains such as medicine, where a wrong diagnosis can put the lives of patients at risk.

When we replace the single, big black box of an end-to-end machine learning system by a series of small black boxes with intermediate results, it becomes much easier to analyse errors produced by the system[3]. The intermediate results also provide a built-in form of explainability. In the case of logical semantics, such explanations would be aimed more at logicians and at formal semanticists than at the end users of a machine learning system, but these intermediate structures could serve as the basis of explanations which are understandable by domain experts.

6.2 Type-logical grammars and wide-coverage semantics

Figure **??** represents the standard treatment chain for type-logical grammars. Text is input to the system, a lexicon transforms the text into statement to present to a theorem prover, which, when successful, returns a proof. This proof is then translated to a proof in multiplicative linear logic, which, by the Curry Howard isomorphism, is equivalent to a linear lambda term. Finally the

[3]A standard criticism against such a setup is that it causes errors to propagate. That is, if, say, the part-of-speech tagger makes a mistake, then the parser will take this mistake as its input (and maybe adds some errors of its own), and so on. However, errors need not propagate like this. We can, for example, use a neural network which outputs different structures at different levels.

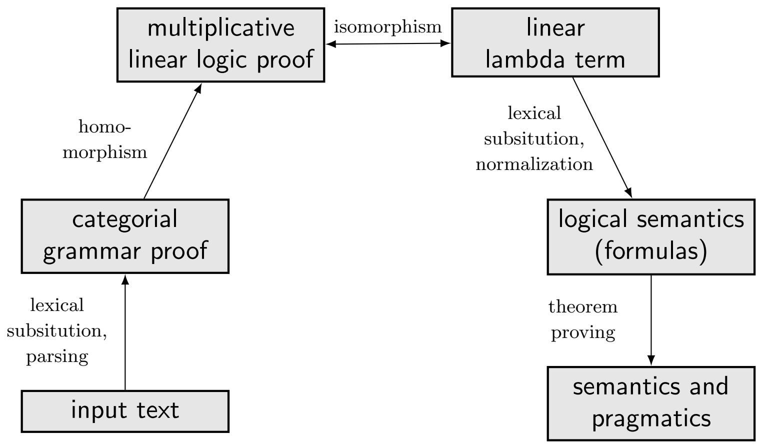

multiplicative linear logic proof
isomorphism
linear lambda term
homo-morphism
lexical subsitution, normalization
categorial grammar proof
logical semantics (formulas)
lexical subsitution, parsing
theorem proving
input text
semantics and pragmatics

Figure 6.1: Type-logical grammar parsing in a nutshell

lexicon is reconsulted to recover the lambda terms assigned to the individual words (these lexical lambda terms are generally not linear). Beta-reduction on this term then produces a term representing a formula (in first- or higher-order logic, or some other meaning representation language). Finally, depending on the logic chosen for meaning representation, standard theorem provers compute entailments, contradictions, etc. between the meanings assigned to different texts. The reader is invited to refer all the way back to Section 1.3 for a detailed example of how we produce a logical formula starting from an input sentence.

Now, when we want to adapt this picture to wide-coverage parsing, what are the main bottlenecks, and where would machine learning be most helpful? The naive way of converting an appropriate treebank — such as the TLGbank or Æthel (**?**, **?**) — into a type-logical grammar lexicon simply extracts all the lexical entries found in the treebank and gives the resulting grammar to a theorem prover. The problem with this is twofold. First, when using a treebank as basis, many common words, such as the coordinations "and" and "or" but also the forms of "to be" and different prepositions will have too many lexical assignments to allow effective parsing. Second, no matter how large the treebank, unseen text will still contain a non-negligible percentage of unseen words. Lexical lookup is therefore a prime candidate for a machine learning solution.

After lexical lookup, we could in principle let one of the type-logical theorem provers run with the result. This would enumerate the available readings for the given lexical assignments. However, for natural language understanding applications, we are generally interested not in an enumeration of all readings but rather in the 'correct' reading of the sentence. Just like it is hard to give explicit rules for the correct formula in the given context, it is hard to specify how to obtain the right reading — and, consequently, the right semantics — for a sentence. So this would be another area where machine learning could help.

Finally, theorem proving in first- or higher-order logic is undecidable. There are a number of possible solutions to this. Firstly, we can do as **?** and try an off-the-shelf first-order logic prover while at the same time running a model finder to find a counter model. This works surprisingly well, and suggests that the natural language inference tasks do not use logically sophisticated forms of reasoning. A second solution is to use a restricted but decidable logical fragment for reasoning. Natural logic is family of restricted logical calculi whose inference patterns are directly relevant for natural language entailment (**?**, **?**), and these logics can be very effective for natural language understanding tasks as well (**?**). Finally, we could also decide to use machine learning here. In many ways, proof search is a game just like Go or chess: the rules are clear, we know when we have won (found a proof) and in some cases also when we have lost. An important difference is that the search space is not bounded. However, it appears reinforcement learning algorithms such as AlphaZero (**?**) or its variants could — with a sufficiently large dataset of examples — become an effective theorem prover for logical statements obtained from natural language inference tasks.

Summarising, we have identified three places where machine learning would be useful in a type-logical natural language processing framework.

- Lexical lookup. Given the lexicon size some sort of approximation seems more or less unavoidable. This will be the topic of Section **??**

- Parsing, for finding the 'best' proof for a statement. This is more easily avoided, we can also use a standard parser and return the first proof found given the probability distributions of the supertagger. **?** call this supertag-factored parsing. Supertag-factored parsing can be used for type-logical grammars as well (**?**, **?**) and we will discuss it in Section **??**. However, a way of combining supertagging and proof net proof search into a neural network, neural proof nets (**?**), will de discussed in Section **??**.

- On the semantic side, machine learning has its place for word sense disambiguation, for example when we want to distinguish "right" (the opposite of "left") from "right" (the antonym of "wrong"). Although we will have little to say about word sense disambiguation (for solutions developed for incorporation into a system like ours, we refer the reader to **?**, **?**), we will discuss meaning assignment in Section **??**.

- Finally, the use of neural networks for theorem proving with logical meaning representations will not be explored here, but left to future research. This means the end result of our processing chain will be the logical representation of the meaning a text.

6.3 Supertagging

Given a type-logical treebank, such as those developed by **?** or **?**, we can obtain a lexicon by simply reading of the lexical leaves of all derivations. However, there are a number of obstacles to using the resulting grammar for parsing.

1. When using the parser on unseen text, no matter how big our initial treebank, there will typically be a percentage of unseen words. This will mean the parser will not find an analysis at all.

2. Many frequent words will be assigned a very large number of formulas. This is a considerable bottleneck: having to consider dozens of formulas for many words will, at best, make any parser extremely sluggish and unsuitable for all but the shortest sentences.

3. Exhaustive search of all readings for all formula assignments, although appropriate and useful for small and linguistically accurate grammars, is not feasible for wide-coverage parsing. We are typically only interested in getting the 'right' reading of a sentence.

These problems are all standard when using wide-coverage parsers and automatically extracted grammars. An influential solution is the use of a so-called supertagger (**?**, **?**). A supertagger is like a part-of-speech tagger (assigning words their syntactic category, typically 'N' for noun, 'V' for verb, etc.) but using richer lexical descriptions. In the case of type-logical grammar, a supertag is simply a formula in our logic. A supertagger is essentially a program which performs a statistical approximation of lexical lookup, and, as such, any type of machine learning algorithm can be used for a supertagger. We will evaluate a few of these algortithms below.

6.3.1 Maximum entropy

Maximum entropy models (**?**) were an important advance in statistical modelling. Their advantage over earlier statistical models was that it was easy to use multiple 'features' without any assumptions of statistical independence. In the context of maximum entropy, a feature can be any predicate which is true or false of the input. Typical features for maximum entropy taggers include things such as 'the current word is capitalised', 'the next word is the end-of-sentence marker'. Maximum entropy models were the state-of-the-art on many tagging tasks — including part-of-speech tagging and supertagging — at least before modern deep learning took over. We will discuss deep learning taggers in the next section (Section **??**).

The maximum entropy taggers of **?** were used with great success on the CCGbank, so as a first benchmark we can test how well their taggers work on the TLGbank. We have used their tools as-is, with no tuning or modification to the parameters. The standard use of the Clark & Curran tools is as a chain, which takes (tokenised) text as input, and then does a part-of-speech

Supertagger results with gold part-of-speech tags

	$\beta = 1$		$\beta = 0.1$		$\beta = 0.01$		$\beta = 0.001$	
	Correct	F/w	Correct	F/w	Correct	F/w	Correct	F/w
Merged	90.41	1.0	96.31	1.4	98.37	2.3	98.80	4.5
Melt	90.36	1.0	96.42	1.4	98.43	2.4	98.88	4.5
Tt	90.12	1.0	96.25	1.4	98.39	2.4	98.84	4.6

Combined POS-tagger/supertagger results

	$\beta = 1$		$\beta = 0.1$		$\beta = 0.01$		$\beta = 0.001$	
	Correct	F/w	Correct	F/w	Correct	F/w	Correct	F/w
Merged	88.71	1.0	94.77	1.4	97.23	2.4	98.06	4.5
Melt	88.78	1.0	94.83	1.4	97.22	2.4	98.10	4.5
Tt	88.79	1.0	95.05	1.4	97.55	2.4	98.26	4.6
Direct	86.87	1.0	94.81	1.5	97.30	2.9	97.94	6.2

Table 6.1: Maximum entropy supertagger results

(POS) tagging step, and uses the output of the part-of-speech tagger as input to the supertagger[4].

The TLGbank provides, for each word, POS tags from two different tag sets: the treetagger tagset (**?**) and the MElt tagset (**?**). Part-of-speech tagging is a task which is generally considered to be 'solved': the best POS taggers have an accuracy in the 97-98% range which is about the same accuracy humans have for this task. For the TLGbank, the maximum entropy taggers assign 98.25% of MElt tags correct and 94.41% of the treetagger tags (with *both* tags correct for 97.99%).

However, part-of-speech tag errors are propagated to the supertagger and this can lead to errors. For example, given that the TLGbank is based on a journalistic corpus (the largest part of the corpus consists of newspaper articles from the French journal "Le Monde"), it has some gaps in its vocabulary. One such gap is the second person pronoun "tu", which appears only four times in the Le Monde part of the TLGbank — "tu" is the familiar personal pronoun, the polite version, "vous" is generally used — and similarly the word "marché", which can be a noun (meaning *market*) or a past participle (meaning *walked*), appears only as a noun. In these cases, errors of the part-of-speech tagger are not recoverable by the supertagger. Other errors of the part-of-speech tagger, such as confusing the subjunctive tense with the present, or the French imparfait with the conditional, have little influence over the supertagger results (although the last type of POS-tag error will propagate to the *semantic* analysis discussed in Section **??**).

Table **??** shows the supertagger results for the different sets of part-of-speech tags (merged denotes the concatenation of the MElt and treetagger tags). The value β is a parameter indicating how many formulas are assigned to each word. Let w be a word, and p the model-assigned confidence in its best supertag, then we output all formulas with probability greater than βp.

[4]Their tools include a Combinatory Categorial Grammar parser as well, which has not been used here.

Corpus	POS	Super	0.1	0.01	F/w
French Treebank	97.8	90.6	96.4	98.4	2.3
Le Monde 2010	97.3	89.9	95.8	97.9	2.2
L'Est Républicain	97.3	88.1	94.8	97.6	2.4
Itipy/Forbes	95.7	86.7	93.8	97.1	2.6

Table 6.2: Supertagger and part-of-speech tagger performance on the different sections of the corpus

So with $p = 0.7$ and $\beta = 0.1$ we assign all formula with probability greater than 0.07. This has the advantage that more formulas are assigned to words about which the model has more 'doubts', but only a single formula when it is sufficiently confident in its assignment.

We can see from the tables that the results are fairly robust with respect to the part-of-speech tagset. When we look only at the best formula for each word, between 90.1 and 90.4% are assigned the correct formula. When we use $\beta = 0.01$, we move up to 98.4% of the words which have the correct formula among their assignments (with an average of around 2.4 formulas per word).

However, these are the results when we take the correct part-of-speech tags as given. To give a fairer indication of performance, the bottom of Table **??** lists the results of the combination of the part-of-speech tagger with the supertagger. With only the best formula for each word, this gives around 88.8% of words the correct formula (compared to 90.4% for the best models with the gold POS-tag), and with $\beta = 0.01$ this increases to a bit over 97% (compared to 98.4% with the correct POS-tag given).

As a second evaluation, Table **??** shows the evaluation of the models on data outside of the French Treebank. In addition to the data from the French treebank, the TLGbank contains additional data: "Le Monde" articles from 2010 (the same newspaper as the French treebank, but at a different time), articles from "L'Est Républicain" (a different French newspaper) and articles from the corpus of the Itipy project[5] containing books mostly from the 19th century describing different voyages through the Pyrenees mountain range. We take the French Treebank section of the TLGbank and split it into a test and a training section. We then use the three other sections as separate sections and evaluate the performance of the tagger, which has now only seen the training portion of the French Treebank, on these four different test sets. We see that performance of the supertagger (and the part-of-speech tagger) degrades when we move to different periods, different journals, and different writing styles. Note that Table **??** lists the results of the *combined* TLGbank corpus, of which the French treebank section is only a part.

The part-of-speech tag models and supertag models — for use with the taggers of **?** — can be found at the following address.

https://github.com/RichardMoot/models

[5]https://richardmoot.github.io/Itipy/

	dev			test		
	F/w	Acc.w	Acc.s	F/w	Acc.w	Acc.s
$\beta = 1$	1	93.22	33.08	1	93.17	33.17
$\beta = 0.1$	1.1	95.67	47.52	1.1	95.68	47.59
$\beta = 0.01$	1.5	97.53	63.84	1.5	97.43	62.16
$\beta = 0.001$	2.9	98.58	76.25	3.0	98.56	75.23
$\beta = 0.0005$	3.8	98.81	79.30	3.8	98.82	78.44

Table 6.3: ELMO LSTM supertagger performance

6.3.2 Deep learning

The use of neural networks for natural language processing is very old (**?**). However, a number of recent developments have greatly advanced the state-of-the-art in many areas in natural language processing, and deep learning models now *are* systematically the state-of-the-art for most tasks (but refer back to Section **??** for some caveats). Two of these important advances are the following:

1. the development of different kinds of word embeddings, including ELMo, BERT, fastText and GPT-3 (**?**, **?**, **?**),

2. the development of new kinds of neural architectures, notably long short-term memory (LSTM, **?**) and the transformer (**?**).

The word embeddings allow us to transform words (as sequences of characters) into fixed-length vectors which can then serve as the input layer of a neural network. Modern embeddings work equally well for unseen words, and the embedding generally takes information about the context of the word into account as well.

The sequence model architectures of LSTM and the transformer allow us to model dependencies between words at any distance — the LSTM using a memory mechanism and the transformer an attention mechanism allowing it to learn which inputs are relevant for producing the output. This should be contrasted with older statistical models (including maximum entropy models) which typically used a fixed window as their context, generally consisting only of the previous two words and the next two words.

We have used the French ELMo embeddings of **?** together with a simple two-layer (bidirectional) LSTM network. The first LSTM layer outputs the two different part-of-speech tags, whereas the second LSTM layer outputs the supertag. This setup has been chosen to remove, as much as possible, the dependence of the supertagger on the correct part-of-speech tag and avoid propagation of errors (which was a problem for the maximum entropy taggers).

6.3.3 Comparison and evaluation

Table **??** shows the performance of the LSTM supertagger. It separates the performance on the development set (dev) from the performance of the test set (test). The important evaluation is on the test set, but the similarity of performance on the dev and test set suggests there is relatively little overfitting on the dev set (this is always a potential problem with machine learning problems). The table lists the average number of formulas assigned to each word for different values of β as 'F/w', and it lists both the word level (Acc.w) and sentence level accuracy (Acc.s). Sentence level accuracy measures whether for each word, the correct formula is among those assigned to it[6].

Compared the the maximum entropy supertagger, the gain in accuracy is important. Even when the gold part-of-speech tag was given, the maximum entropy supertagger assigned the correct formula in 90.4% of cases (without the correct POS tag, only 88.7% of the words are assigned the correct formula). The LSTM supertagger assigns 93.2% of words the correct supertag, and does so without POS tag information. Even though the advantage diminishes when we increase the β value (as with the maximum entropy results, a lower β value results in more supertags per word), the advantage of the LSTM tagger remains.

It is a bit complicated to compare Table **??** directly with Table **??**. The β value is not really the value which interests us: for parsing, it is the trade-of between precision (that is, the supertagger providing the correct supertag among its possibilities) and the number of formulas per word. Too many formulas per word will reduce parsing speed, but a precision which is too low will not allow us to find a proof at all. Figure **??** therefore plots the performance of the maximum entropy supertagger against the performance of the LSTM supertagger. The real comparison is of course between the green curve (representing the maximum entropy tagger using the output of its corresponding part-of-speech tagger) and the blue one, although a slight advantage over performance of the supertagger when given to gold POS tags remains.

The figure also makes it clear that the final 1% of word level precision is where the real difficulties are. At the sentence level, everything is much harder. While a bit less than a third of all sentences are unproblematic (that is, they can be fully parsed using only the best supertag for each word) there is also the final 20% which is very hard. In this context, 'very hard' means that the parser will make at least one mistake, although not all parser mistakes necessarily translate into a problem for downstream tasks. However, this still means that many linguistically interesting phenomena are currently out of reach for the parser. As an example of this problem, the gapping examples which we have seen several times since sentence (23) in Section 1.5.2, have been annotated as such in the TLGbank (**?**, Appendix B.4). However, since

[6]This should be confused with the most likely provable sequent given the assignments (and their probabilities) being the correct one. It only shows that, in principle, the parser could find the correct parse but not that it actually does so. We will use this much harder evaluation in the section on neural proof nets (Section **??**).

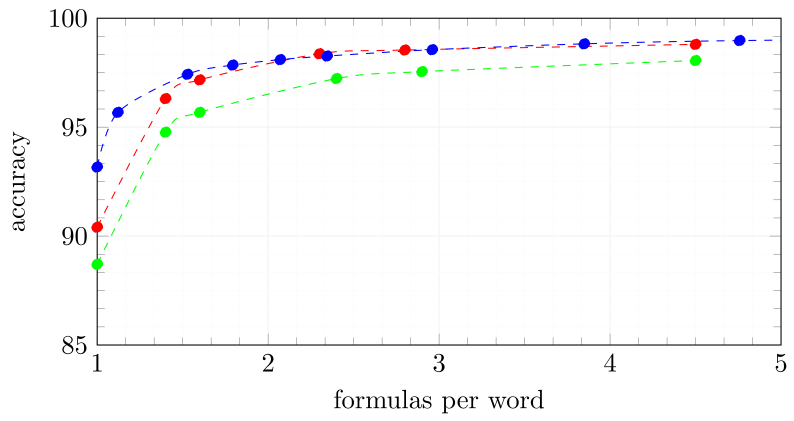


Figure 6.2: Comparison of the maximum entropy versus the LSTM supertagger

these constructions are quite rare (they occur in a bit over 1% of sentences) the supertagger generally does not assign a high enough probability to the gapping type to allow us to find the correct analysis[7].

The supertagger can be found at the following address.

https://github.com/RichardMoot/DeepGrail

6.4 Supertag-factored parsing

Once we have found a set of formulas for each word in the sentence, the next step consists of finding a type-logical analysis using these formulas. The supertagger described in the previous section has assigned each word a limited number of formulas, assigning each formula a probability. For a supertag-factored parser (**?**, **?**), we only use the supertag probabilities for deciding the most likely parse: the parser actions themselves are not assigned weights, they simply combine the probabilties of the supertags (treating these probabilities as independent), although we can assign preferences to, say, right-branching over left-branching structures.

The parser of **?** is an instance of the 'deductive parsing' technology of **?**, using the same core parsing engine with a number of special rules for the treatment of extraction, gapping and related phenomena.

The current chart parser was originally introduced as a preprocessing step for a proof net algorithm (**?**). However, this preprocessing step turned out to be so effective that it soon handled a bit under 98% of the complete French

[7]If we are really interested in gapping, we can of course fix this at the machine learning level, for example by adding significantly more gapping examples or by more severely penalising incorrectly tagged gapping constructions. However, this trade-off will likely leave parser performance worse in many constructions unrelated to gapping.

Type-Logical Treebank and therefore it made sense to add additional chart rules to handle the remaining few percent as well.

Each chart item is assigned a probability, the lexical entries get their probability from the supertagger and the rules then specify how these probabilities are combined. As is fairly standard, the implementation doesn't use probabilities and then multiply them, but rather uses log probabilities and adds them. This prevents problems with precision and numerical underflow which occur when we multiply many small probabilities.

The current implementation is quite flexible with respect to these combinations: in supertag-factored mode, the log-probabilities of the rule premisses are simply added for the rule conclusion. In bootstrap mode, the parser uses the output of a context-free parser (such as the Stanford parser or the Berkeley parser) and adds extra penalty term for the number of brackets of the bootstrap parse the resulting constituent crosses.

Following **?**, we have also implemented an A* parser. A* search is a standard AI algorithm (**?**, Section 3.5.2) (**?**, Chapter 12), a type of best-first search using an estimation of the best possible path to the goal from each state. For searching a shortest path in a geographical information system, we know that we can never do better than the Euclidean distance between the current point and the goal. Similarly, for a supertag-factored parser, we can never do better to complete the current parse state than using the highest probability supertags for each of the non-yet-parsed words.

The chart parser can be found at the following address.

https://github.com/RichardMoot/GrailLight

6.5 Neural proof nets

The chart parser described in the previous section may seen a bit anti-climactic: we spent many chapters extolling the virtues of proof nets, and then, finally, when the need arose to move to large-scale analysis, we reverted back to classic chart parser technology, and, moreover, using an incomplete implementation of the logic!

Can we combine the benefits of deep learning with the benefits of proof nets? **?** show that we can. As we have already seen in several places, given a statement $\Gamma \vdash C$ the search space of proof search using proof nets can be summarised by a set of square matrices, one for each atomic formula, with the positive and negative occurrences of the atomic formulas the rows and columns of the matrices. Each perfect matching of the atomic formulas is a proof structure, a potential proof of the input sequent. In the context of wide-coverage parsing, we are not only interested in whether there exists a proof net for the input, but we also want the 'right' proof net, that is the proof net which corresponds to the intended reading of the sentence.

The task for neural proof nets is therefore the following: given an input sentence, we transform the words into formulas (that is, supertagging, as we have already seen in in Section **??**), we unfold the formulas as usual, then

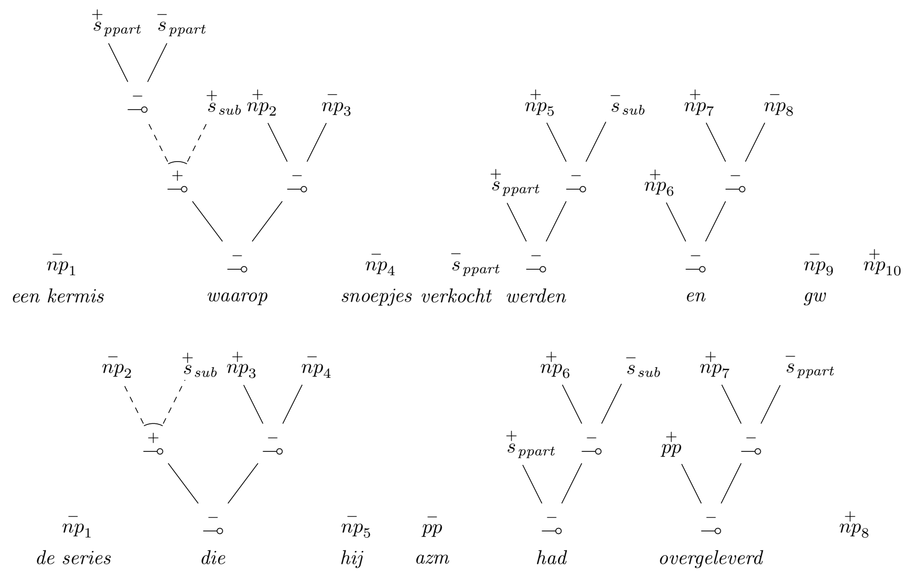


Figure 6.3: Formula unfolding for Examples **??** and **??**

have the neural network learn the intended reading (that is, matching of the atomic formulas) based on examples from an annotated corpus.

To makes this discussion more concrete, Examples **??** and **??** present two noun phrases (both part of larger sentences) from the Æthel treebank.

(3) een kermis waarop snoepjes verkocht werden en gedroogde
a fair on which candy sold were and dried
wijting
whiting
a fair where candy was sold and dried whiting

(4) de series die hij aan zijn medewerkers had overgeleverd
the series which he to his collaborators had passed on
the series which had had passed on to his collaborators

Figure **??** shows the formula unfolding (given the correct formulas/supertags). The *np* formulas have been given integer indices to facilitate referring the specific atomic formulas. To save space "een kermis" (*a fair*), "gerookte wijting" (*dried whiting*), "de series" (*the series*), and "aan zijn medewerkers" (*to his collaborators*) have been treated as atomic constituents, all except the last one of type *np*; "aan zijn medewerkers" is analysed as the prepositional phrase *pp* labeled "azm" and "gerookte wijting" as the np labeled "gw". The actual annotation of these examples also includes semantic role labelling (subject, object, modifier, etc.) for each dependency.

As is clear from the two unfoldings the main difficulty lies in matching the *np* formulas. Our model (**?**) computes Dutch BERT embeddings (**?**) as a first step, and uses a transformer for supertagging[8]. The supertag output can be manually inspected, but is passed directly to the axiom linking component. The transformer architecture learns a vector for each atom in the supertagger output; since it is a transformer it can use the full context (in the current case, the BERT word information but also the context in its formula and the surrounding formulas). For the noun phrases in Example **??**, the transformer obtains a 5×1 vector for both the positive and the negative noun phrases. We multiply these vectors, the positive with the transpose of the negative to obtain a 5×5 matrix, which we normalise to obtain a probability distribution (that is, all rows and all columns sum to 1 and all values are ≥ 0). We have used square matrices to represent the combinatorics of proof search throughout this book, but now we have a probabilistic version of proof net proof search.

Our model learns the preferred reading of a sentence by sending the square matrices through a Sinkhorn layer (**?**, **?**). A Sinkhorn layer updates the weights in the matrix in such a way that iteration pushes one value in each row and one value in each column closer to one and all other values closer to zero. In terms of proof nets, this forces a specific axiom linking.

It should be noted that the logic is fully commutative, so the neural model has to 'learn' all word order information. In addition, the neural network has to learn the correctness condition for proof nets. However, the count check is automatically enforced by the restriction to square matrices for the Sinkhorn layer.

To summarise, our neural proof nets operate as follows:

1. words are translated into contextualised BERT embeddings,
2. a transformer layer translates the BERT embeddings into formulas; formulas are treated as sequences of characters (allowing the lexicon to suggest unseen formulas),
3. vectors of positive and negative atomic formulas are multiplied to produce square matrices,
4. a Sinkhorn layer produces a matching of the atomic formulas.

Together, the output of the supertagger component — as shown in Figure **??** — and the output of the Sinkhorn layer uniquely determine a proof structure (although not necessarily a proof net). Figure **??** shows the correct proof nets for the unfolding of Figure **??**.

Table **??** summarises the results of the neural proof net parser. The parser uses beam search with parameter b representing the beam size (with beam b, only the b best possibilities are expanded at each step). The first row lists the supertagger performance. Since the Æthel treebank has more distinct

[8]Contrary to previous supertaggers this supertagger outputs the supertags as sequences of symbols. It can therefore propose previously unseen formulas.

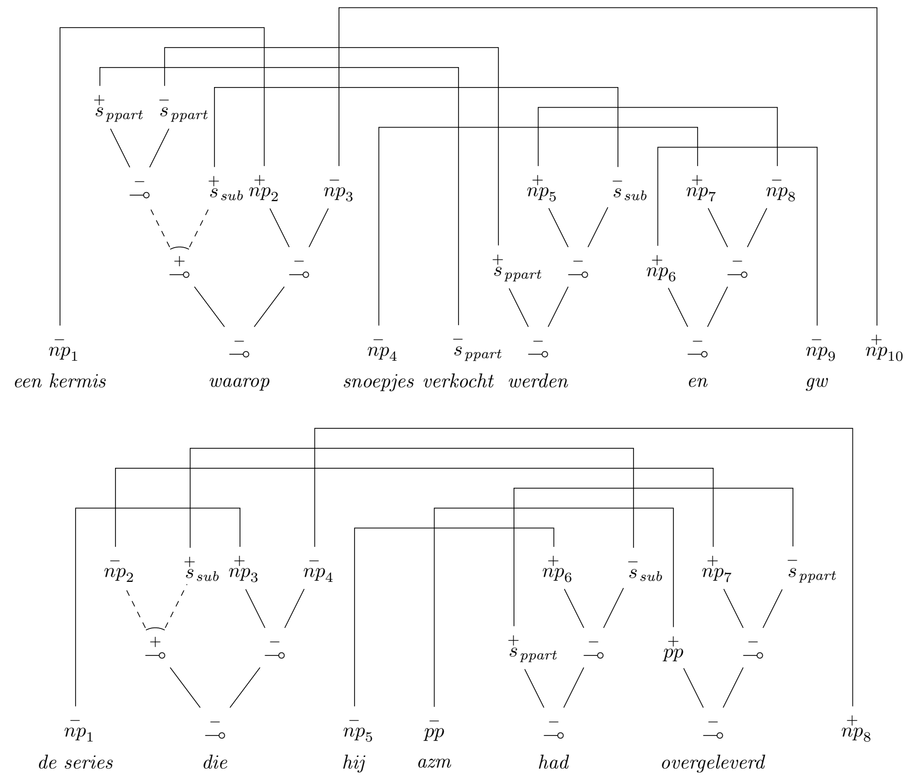


Figure 6.4: Proof nets corresponding to the formula unfoldings of Figure **??**

Metric	Beam size			Baseline
	$b = 1$	$b = 3$	$b = 7$	Alpino
Formula correct	85.5	92.4	93.4	56.2
Sequent correct	57.6	68.0	70.2	n/a
Term correct	60.0	67.7	69.6	45.7

Table 6.4: Results of the neural proof net parser

formulas and much higher lexical ambiguity than the TLGbank, supertagger performance is good. The row 'sequent correct' denotes the cases where all correct formulas have been assigned by the supertagger. The row 'term correct' gives the percentage of sentences for which the neural parser has obtained the correct lambda term. This essentially means that the parser has made no errors (or at least none which could affect the meaning assignment) and it is a *very* strict evaluation: most parsers are evaluated by measuring the percentage of correct parser actions (or correct brackets) rather than only on whether they produce exactly the right structure. As such, getting between 60 and 70% sentences fully correct is quite good.

For comparison, we have used the Alpino parser (**?**), an older, maximum entropy inspired Dutch parser, which uses a large handwritten grammar and which is still quite close to state-of-the-art for Dutch. Running the Alpino parser (without time limits) on our test set and using the methodology of **?** to translate the Alpino parse into a linear logic derivation[9], we can compare the performance of our parser against Alpino. We note that compared to Alpino, our neural proof net parser represents a significant improvement both on the number of correct formulas and with respect to the correctly assigned meaning terms to unseen sentences.

6.6 Semantics

The parsers of the previous sections (the supertag-factored parser and the neural proof net parser) produce linear lambda terms corresponding to the derivational semantics of the input. Although these can be directly used for applications in natural language understanding (**?**), we are also interested in wide-coverage semantics, which requires a way of assigning meanings to the proofs obtained by our parsers.

6.6.1 Three models for the basic units of meaning

In computational linguistics, there are three basic ways to represent meaning, each with their different strengths and weaknesses. For many applications, we use more than one of these.

1. the meaning of a word is a vector in n-dimensional space, giving an abstract relation to other words,

2. the meaning of a word is a node in a lexical network, specifying its relations to other words (its synonyms, antonyms, hyponyms, hypernyms, etc.),

3. the meaning of a word is a lambda term specifying how to construct a logical formula representing its meaning.

[9]This is possible because the Lassy treebank, which we used for the extraction of Æthel, consists of Alpino output which has been manually corrected.

amour	*love*	1.0
tendresse	*tenderness*	0.71
bonheur	*happiness*	0.67
amitié	*friendship*	0.59
platonique	*platonic*	0.57

Table 6.5: Similarity for vector-based meanings

There are many different ways of computing vector-based or distributional meanings for words. In the modern era (**?**) most word representations are estimated from large corpora using some type of neural network, and these include ELMo and BERT we have seen before. As noted by **?**, these distributional meanings do capture some interesting facts of morphology (relating different forms of the same verbs, or adjectives and their corresponding adverbs) and some world knowledge as indicated by calculations like the following.

$$\begin{gathered} \mathit{Paris} - \mathit{France} + \mathit{Italy} = \mathit{Rome} \\ \mathit{sushi} - \mathit{Japan} + \mathit{Germany} = \mathit{bratwurst} \end{gathered}$$

In the equations above '+' and '−' denote vector addition and subtraction, whereas the word on the right hand side of the '=' symbol is the nearest word (according to the embedding) of the vector computed on the left hand side of the equation. We should read these calculations as something like 'Paris is to France as x is to Italy' and 'sushi is to Japan as y is to Germany' with the results $x = \mathit{Rome}$ and $y = \mathit{bratwurst}$ computed through the vector meanings.

Table **??** shows the nearest words for the French word "amour" (*love*), using the embeddings provided by **?**. It's a relatively varied list, and the listed words are semantically close but in different ways.

In spite of the successes of embeddings like BERT for entailment tasks, it does not seem clear to us how to use vector based meanings for individual words can be used to obtain logic-based meanings except in the rather roundabout way we have discussed in the previous sections of this chapter: use the embeddings as input to compute the formulas (Section **??**) and then the proofs (Sections **??** and **??**). This confirms better to our initial goal of our natural language processing framework providing explainable intermediate steps.

The second theory of meaning assigns each word a node in a lexical network. In this context, a small part of the meaning of "amour" (*love*) as provided by the lexical network JeuxDeMots (**?**) is shown in Figure **??**. The lexical network provides useful information such as the fact that "love" is a "feeling" and an antonym to words like "hate" and "indifference" but also that typical patients[10] of love are wives and children.

[10]The term 'patient' is a thematic relation denoting who or what undergoes an action (often the grammatical object), whereas the 'agent' performs the action.

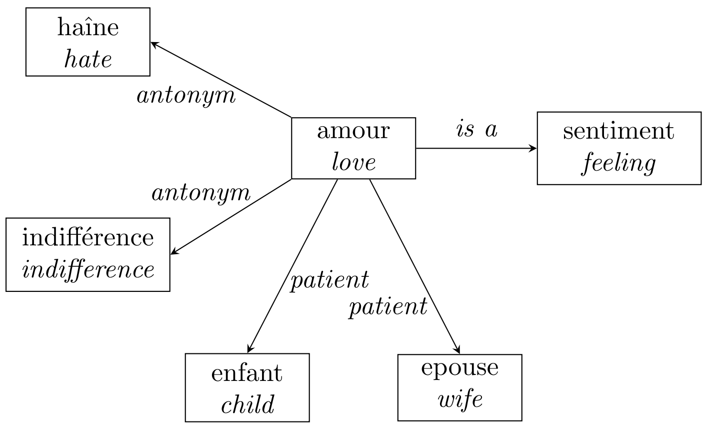


Figure 6.5: The meaning of "amour" (*love*) according to JeuxDeMots

The information in the lexical network is directly useful for entailment. We can infer that 'x loves y' contradicts 'x is indifferent towards y', using the antonym or that it contradicts 'x has no feelings towards y' using the fact that love is a feeling. The typical patients can also be used as information for an anaphora resolution system.

The final theory of meaning does not have a great deal to say about the meaning of "love". According to **?** the noun "amour" is simply something that is true or false of certain entities where as the verb "aimer"(*to love*) is an arbitrary relation between two entities in the domain. **?** adds both meaning postulates and possible worlds to this picture, but this still does not provide a particularly profound theory of meaning. From the abstract point of view of translating natural language expressions into formal logic, it makes perfect sense to use logical predicate symbols for words such as verbs and nouns, of course. And this does have the advantage that it is easy to scale up. When we say that for every transitive verb v, its meaning is the binary predicate v' we simply sidestep (and leave for later, or for other people to resolve) the hard linguistic and philosophical questions of when we can accurately use this verb to describe a situation, and what this really means.

Montague semantics is mostly concerned with those words which use logical connectives as part of their translations, and it is these words about which it has the most interesting things to say. The interesting thing is that these are mostly closed-class words: "et" (*and*), "chaque" (*every*), "que" (*which*) for which we can therefore list the semantic terms.

6.6.2 Montague style wide-coverage semantics

The advantage of Montague semantics for applications in wide-coverage semantics is that we can treat many types of entries with one generic entry. Suppose we see a new noun n in the text. Without knowing anything specific about this noun, we can simply assign it the lambda term $\lambda x.(n'x)$ (or even the equivalent, eta short term n').

Adjectives Adjectives are already a bit more complicated. Given an adjective a and a noun n there are (at least) the following two possibilities for combining their meanings[11].

$$\lambda x.((a\,n)\,x) \wedge (n\,x) \tag{6.1}$$

$$\lambda x.((a\,n)\,x) \tag{6.2}$$

A typical example of **??** would be a "big insect", which would be an insect and big compared to other insects although not necessarily a big animal. A typical example of **??** would be an "alleged murder" who is not necessarily a murderer, but only accused or suspected of being one. Our solution is to use **??** as the the default, and list all exceptions requiring **??** explicitly (this is quite a short list containing adjectives like "possible" and "présumé" *presumed*).

Verbs With respect to verbs, the basic treatment is again simple. An unseen transitive verb v can be assigned the lambda term $\lambda y \lambda x.((v'\,y)\,x)$. However, as already noted by **?**, some transitive verbs like "seek" do not necessarily entail the existence of their object and so "John seeks a unicorn" need not entail the existence of unicorns. These are not the only special properties verbs can have. For example, the following two sentences are structurally very similar.

(5) John promised Mary to leave.

(6) John persuaded Mary to leave.

Sentence **??** means that John promised Mary that *John* would leave. Sentence **??**, on the other hand, means that John persuaded Mary causing *Mary* to leave. In higher-order logic, this difference in meaning is easily expressed by the following difference in meaning assignments, where only the argument of the nested predicate P has changed.

$$\lambda y.\lambda P \lambda x.promise(x, y, (P\,x)) \tag{6.3}$$

$$\lambda y.\lambda P \lambda x.persuade(x, y, (P\,y)) \tag{6.4}$$

There is a limited number of verbs allowing constructions like **??** or **??**, and they are listed with the appropriate semantic term in our lexicon.

[11]Many authors have one or two additional types, the intersective adjectives $\lambda x.(a\,x) \wedge (n\,x)$ (a possible example would be "red" denoting the things that are both red and satisfy the noun meaning, although it is reasonable to argue that even for "red" the meaning depends on the noun a "red apple" and "red wine" have rather different colours) and the privative adjectives $\lambda x.(a\,x) \wedge \neg(n\,x)$ (this last one is more debatable). Note that for both of these we have changed the type of a from $(e \rightarrow t) \rightarrow e \rightarrow t$ to $e \rightarrow t$ (although one possibility would be to define the second from the first using $a_2 \equiv (a_1(\lambda y.y = x))$).

Factives and presuppositions Another important class of verbs are the so-called factive verbs. These verbs have an embedded phrase as argument, which is entailed even when in normally non-entailing contexts, such as in the scope of negation (as in **??** and **??** below) and or a question (as in **??**), or when in the antecedent of an implication (as in **??**).

(7) John didn't know that Mary had filed for divorce.

(8) Mary didn't regret filing for divorce.

(9) Does Mary regret filing for divorce?

(10) If John knows Mary has filed for divorce, he will probably ask her out.

All of Sentences **??** to **??** entail that Mary filed for divorce. However, naive computation of a logical form will not immediately give these entailments. For example, a slightly simplified but otherwise standard meaning for **??** is shown below as **??**.

$$\neg \mathit{know}(j, \mathit{file\_for\_divorce}(m)) \tag{6.5}$$

Even when we add a higher-order axiom (or meaning postulate) such as **??** below, we cannot derive *file_for_divorce*(m) as we would like to.

$$\forall x \forall P \mathit{know}(x, P) \Rightarrow P \tag{6.6}$$

There has been a lot of discussion in the literature about how to obtain a meaning like **??** for a sentence like **??**.

$$\mathit{file\_for\_divorce}(m) \wedge \neg \mathit{know}(j, \mathit{file\_for\_divorce}(m)) \tag{6.7}$$

The actual properties of factive verbs, and more generally the larger class of presuppositions, are fairly complex but there are a number of solutions proposed in the literature (**?**, **?**, **?**). Our implementation is a simplified version of the one proposed by **?**. Presuppositions are used for proper names like "Jean" but also for the definite article "le/la" (*the*) which all entail the existence of the referenced object even in a non-entailing context. Factive verbs and a number of other words and constructions having presupposition-like behaviour are indicated as such in our lexicon.

Event semantics For the treatment of adverbs, we have followed the approach of **?**. Davidson proposes we analyse sentences like Sentence **??** roughly as shown in **??**.

(11) Jones buttered the toast in the bathroom with a knife at midnight.

$$\exists e.\mathit{butter}(e, j, t) \wedge \mathit{in\_the\_bathroom}(e) \wedge \mathit{with\_knife}(e) \wedge \mathit{at\_midnight}(e) \tag{6.8}$$

The event variable e denotes an event, essentially a slice of space-time. It is used as an extra argument of "buttered", so $\exists e.\mathit{butter}(e, j, t)$ indicates that there is an event e of Jones (j) buttering the toast (t). The advantage of this is that the different adverbs now only have this event variable e as argument and this gives us some useful entailments for free. For example, from **??** both **??** and **??** are entailed.

(12) Jones buttered the toast with a knife.

(13) Jones buttered the toast in the bathroom and midnight.

Inversely, with the additional information that **??** and **??** describe the same event, these two sentences together entail **??**. This type of reasoning is much more complicated with a higher order formula such as **??** below.

$$\mathit{at\_midnight}(\mathit{with\_knife}(\mathit{in\_the\_bathroom}(\mathit{butter}(j,t)))) \tag{6.9}$$

Such an approach is also easily integrated with a theory of tense and temporal semantics, with the temporal logic providing predicates over the events (or at least the temporal aspect/interval of the event). For example, the past tense would indicate that the event took place before the current time n (or at least *started* before it[12]).

Discours Representation Structures Finally, instead of higher-order logic for our semantic representation language, we have chosen Discourse Representation Structures (**?**). These provide a convenient, graphical representation of logical formulas with dynamic binding (**?**). In the form of a Discourse Representation Structure, formula **??** looks as follows.

$$\begin{array}{|l|}\hline e,\ j,\ t \\ \hline \mathit{butter}(e,j,t) \\ \mathit{in\_the\_bathroom}(e) \\ \mathit{with\_knife}(e) \\ \mathit{at\_midnight}(e) \\ \hline \end{array}$$

The constants and existentially quantified formulas are shown in the top part of the box and the predicates over these variables and constants are shown in the bottom part. These are atomic formulas, implicitly joined by conjunction. Other logical connectives such as implication, disjunction or negation produce nested boxes. For example, the two readings of "every man loves a woman" are shown below, with the "every" wide scope reading on the left, and the "a" wide scope reading on the right.

$$\begin{array}{|c|}\hline \\ \hline \begin{array}{|l|}\hline x \\ \hline \mathit{man}(x) \\ \hline \end{array} \Rightarrow \begin{array}{|l|}\hline y \\ \hline \mathit{woman}(y) \\ \mathit{love}(x,y) \\ \hline \end{array} \\ \hline \end{array} \qquad \begin{array}{|l|}\hline y \\ \hline \mathit{woman}(y) \\ \begin{array}{|l|}\hline x \\ \hline \mathit{man}(x) \\ \hline \end{array} \Rightarrow \begin{array}{|l|}\hline \\ \hline \mathit{love}(x,y) \\ \hline \end{array} \\ \hline \end{array}$$

[12]Temporal semantics, like presupposition, is another big topic in semantics (**?**, **?**, **?**, **?**). For our implementation of tense (**?**), we have followed **?**.

One of the main advantages of Discourse Representation Theory is that is provides a convient way of representing the variables accessible for anaphoric reference. For example, variable y (corresponding to *woman*) is only accessible in the rightmost DRS. This means that continuing the previous sentence with a sentence like “she is getting very tired of all the attention” (with “she” corresponding to the woman y) is only valid for the rightmost DRS and not for the leftmost one.

6.6.3 The semantic lexicon

The current implementation contains 341 lexical patterns to assign formulas to words given their lexical root form, part-of-speech tag and formula. This includes nouns, adjectives, verbs, prepositions, etc. but also special cases for things like factive verbs and weather verbs[13]. In addition to these generic lexical lambda term recipes, there are 722 specific lexical entries for words whose meaning is not as generic. This includes verbs like “être” (*to be*), the different relative pronouns “que” (*what/which*), “qui” (*who*), the logical connectives “et” (*and*) and “ou” (*or*) with their different syntactic formulas, logical determiners “chaque” (*each*), “un/une” (*a*), “le/la” (*the*).

The lexicon gives a rudimentary treatment of a number of interesting semantic phenomena: coordination, relative pronouns and extraction, presupposition, tense, gapping. For generalised quantifiers, only the surface order (that is, subject wide scope) is computed. This is because the French Treebank does not contain annotations for the relative scope of quantifiers, and adding this information is a complex and laborious task. To improve the usefulness of the computed representation for entailment tasks, our semantic representation would benefit from the incorporation of a designated word sens disambiguation component, which matches the predicate symbols of our meaning representation to nodes in a lexical network, thereby allowing us to distinguish “souris” (*mouse*) as a type of rodent from “souris” (*mouse*) as as a computer input accessory.

Although the current lexicon does not yet cover all 1,101 formulas of the TLGbank, it allows us to compute the meaning for a large chunk of the treebank.

6.7 Conclusion

In this chapter, we have presented our work on wide-coverage syntax and semantics for type-logical grammars. Contrary to many popular approaches today, we have been interested in using machine learning as little as possible, using it only for lexical lookup and parsing. This means we produce a number of intermediate structures — formulas corresponding to words, proofs of grammaticality of sentences and logical representations of sentence meanings — which can be independently verified an explained. When a logical meaning

[13]Weather verbs like “pleuvoir” (*to rain*) are special only in that “il pleut” (*it rains*) is translated as “*pleuvoir*(e)” without only an event argument and no subject.

is produced, we trace back the choices which led to it. We have evaluated the performance of our supertagger and neural proof net parser and have found they performed well, with 92.2% of formulas correctly assigned for French (Section **??**), and with the correct proof for 69.6% of all sentences for our Dutch treebank (**?**)

Although the final output of our system is a logical structure mostly of interest to the formal semanticist, I believe such structures can be fruitfully used to natural language understanding tasks such as textual entailment and question answering (as shown in an only slightly different context by **?**, **?**).

7 Conclusions and future directions

After this brief tour through the landscape to type-logical grammars, it is time to conclude. We have seen that type-logical grammars are a systematic way of using logical calculi for natural language analysis, which puts special emphasis on the syntax-semantics interface.

The failures of the Lambek calculus have inspired a number of variants and extensions. On the surface, there appear to be many unreconcilable differences between these logical calculi. However, when we look below the surface at the underlying linear logic proofs used for calculating meaning, there is general agreement. In other words, the 'deep logic' (intuitionistic linear logic) is relatively uncontroversial, but there is disagreement mainly about the 'surface logic', with different logical connectives, differently structured sequents, and different structural rules.

I have presented two main ways to adapt proof nets to modern type-logical grammars. These are two general frameworks for proof nets which together capture all modern type-logical grammars.

The first framework, presented in Chapter 4, is a rather simple and standard fragment of linear logic, the first-order multiplicative fragment (**?**, **?**). As already shown by **?**, first-order linear logic has the Lambek calculus as a natural fragment, but this can be extended to lambda grammars, hybrid type-logical grammars, and (a large part of) the Displacement calculus (**?**, **?**, **?**).

The second framework, presentend in Chapter 5, is inspired by the interaction nets of Lafont (**?**, **?**) and the multimodal proof nets of **?**. It uses a simple form of graph rewriting to represent logical statements and verify their correctness. Compared to first-order linear logic, which needs to summarise all structures and operations with a finite number of variables and constants, the graph rewrite perspective is more powerful and flexible, and as such it can represent all modern type-logical grammars. In addition to the formalisms representable in first-order linear logic, we also represent multimodal type-logical grammars, NL_λ, the extended version of hybrid type-logical grammars of **?**, and the Lambek-Grishin calculus (**?**, **?**, **?**, **?**, **?**).

These new and/or alternative proof systems for type-logical grammars provide us with a number of benefits, besides providing redundancy-free representations of proofs.

1. In many cases we get new results with respect to computational complexity essentially for free. For example, we have simple NP completeness proofs for the Displacement calculus and hybrid type-logical grammars.

2. Representing different formalisms in a single framework facilitates the comparison of different linguistic analyses in these frameworks. In first-order linear logic, we can use the derivability relation between translations of formulas in different frameworks. For example, the analysis of gapping in the Displacement calculus and in hybrid type-logical grammars turn out to produce equivalent formulas upon translation in first-order linear logic. In the graph rewriting perspective, we can show isomorphism between structures, and equivalence between rewrite operations. This, for example, allows us to show that some of the iconic analyses in hybrid type-logical grammars (such as gapping) and in NL_λ (such as the analysis of "same") are intertranslatable; this gives new analyses of these phenomena in the translation targets.

3. We can, in many cases, *combine* multiple logics into a single logic. For example, with respect to linguistic phenomena like across-the-board extraction[1], which are a problem for hybrid type-logical grammars, we can choose to combine the first-order translations of hybrid grammars with those of the Displacement calculus, or alternatively, we can combine the graph-theoretic operations of hybrid grammars with those of NL_λ.

4. We can use a single theorem prover engine for multiple grammatical frameworks, as illustrated by **?**. This theorem prover uses proof nets for first-order linear logic as an underlying engine for finding proofs in lambda grammars, Lambek grammars, hybrid type-logical grammars and Displacement grammars. If the grammar writer wants, the first-order linear logic proofs can be completely hidden, and both the input grammar and output proofs can be provided in the desired logic. Unfortunately, there is, as of yet, no similar engine for the graph rewriting perspective.

Taken together, the two proof net frameworks for modern type-logical grammars show that, despite starting for different logical primitives, there

[1] We have seen across-the-board extraction as example (17) in Section 1.5.2.

is much greater convergence than has previously been assumed. There is a 'common core' of phenomena which most formalisms can treat elegantly, but also some differences around the edges which require more careful investigation to evaluate the relative benefits of one formalism over another.

Finally, in Chapter **??**, we have shown how to use type-logical grammars for wide-coverage syntax and semantics. While the logical and formal study of our grammar formalisms is very important, it is also important to show how, at least in principle, these formalisms are useful for some of the standard tasks of natural language understanding. This includes systems for natural language question answering and for recognising textual entailment.

Doing such research requires a certain amount of machine learning and a large set of annotated examples. We have developed two such datasets, the French TLGbank (**?**) and the Dutch Æthel treebank (**?**). We use machine learning models trained on these treebanks both for lexical lookup (so-called supertagging) and for proof search, culminating in a system of neural proof nets (**?**). Both the supertagger and neural parser receive very good results — over 93% correct supertags (**?**) and 60-70% of proof net parses essentially without errors (**?**) (that is, the correct lambda term was obtained). The final lambda terms (or the logical semantic representation) can then be used for downstream tasks in natural language understanding, something we will leave for future research. However, we have shown that, far from being just a theoretical tool, type-logical grammars can also be used, at least in principle, for applications in natural language understanding.

7.1 Open questions and future research

Formal language theory One of the big open questions for the different type-logical grammars is to obtain a precise characterisation of the language classes generated. With the exception of multimodal categorial grammars and the Lambek calculus (which generate the context-sensitive and context-free languages respectively) we know next to nothing about upper bounds for language classes. Most formalisms can easily be shown to generate some variant of the mildly context-sensitive languages, but the techniques used in the proof of **?** are difficult to generalise to other formalisms:

1. **?** shows that, whereas for Lambek grammars we can always guarantee that interpolants do not increase the maximum size of formulas, this can no longer be guaranteed in a commutative context.
2. **?** show by a simple counting argument that a meaning-preserving translation such as Pentus' cannot translate a type-logical grammar treating quantifier scope into a mildly-context sensitive grammar with at least one derivation for each reading (essentially because an n quantifier sentence requires n pointers, whereas mildly context-sensitive formalisms can only use a constant number of them).
3. **?** shows that type-logical grammars easily generate language classes closed under permutation, because the same mechanism used to handle

medial extraction allows us to move arbitrary words to the front of the sentence, thereby generation arbitrary permutations. However, the formal language classes known to handle permutation closure such as Range Concatenation Grammars (**?**) also allow other operations such as copying which appear to be outside the language classes handled by type-logical grammars[2].

By only looking at the Horn clause fragments of type-logical grammars (that is, without embedded implications, allowing formulas only of the following form $F ::= p \mid p \multimap F$, for p atomic) we avoid all of the problems listed above and obtain some equivalences (and thereby lower bounds for the general logics) (**?**, **?**). However, restricting ourselves to the Horn clause fragment amounts to giving up on all the strong points of type-logical grammars at the level of the syntax-semantics interface.

Beyond multiplicatives We have only considered the multiplicative connectives throughout this book. However, there is a good case to be made for using a more expressive logic for natural language analysis. The linear logic additives, exponentials, and second-order quantifiers all have some proposed linguistic applications (**?**, Section 2.5). The problem with adding them is that these groups of connectives increase complexity of the proof theory and the computational complexity of the logic. **?** provides a logic with additives, exponentials (and many other connectives: there is a total of 49 different connectives) but without second-order quantifiers. I would like to use cleaner and more standard logic for natural language analysis. One promising candidate for such a logic would be a version of soft linear logic (**?**). This logic has additives, exponentials and second-order quantifiers, to which I would propose to add the first-order quantifiers. To find a decidable fragment of this extension of soft linear logic, we could look at restrictions on the application of the second-order quantifiers, similar to the results of **?**.

For the linguistic applications of the additives, it appears we handle most or all of the applications using complex axioms of the following forms.

$$A_1 \& \ldots \& A_n \vdash B_1 \& \ldots \& B_m \tag{7.1}$$

$$B_1 \oplus \ldots \oplus B_m \vdash A_1 \oplus \ldots \oplus A_n \tag{7.2}$$

With the condition that each of the B_i is identical to one of the A_j formulas, some valid instances would be the following.

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

In such a setup we can avoid the (proof theoretically) complicated $\oplus E$ and $\& I$ rules. For example, it is easy to add such restricted additives to a proof net theorem prover as additional non-determinism in the formula unfolding

[2] I conjecture that type-logical grammars generate mildly context sensitive languages and their permutation closures.

stage. Although as a logician, I am never happy about removing logical rules, with respect to the linguistic applications, we do not seem to need them[3]: neither **?** nor **?**, even mention the $\oplus E$ and $\& I$ rules, even though they handle all the standard linguistic examples.

Comparing proof systems, comparing analyses While we have made a number of comparisons between different type-logical grammars as well as their analyses of different phenomena on the syntax-semantics interface, many questions remain open here as well. While the question ‘which type-logical grammar is the best for natural language analysis?’ is unlikely to have a definite answer, the differences between formalisms and analyses, especially cases where one formalism has a clear advantage, require further study. This can be done linguistically (for example by providing an appealing analysis of certain phenomena in the chosen framework and challenging other formalisms to provide an analysis which works as well) but also logically (by showing that certain analyses have no translation in some formalism).

Graphs and theorem proving The graph rewriting perspective on proof nets, at least in the general form presented in Chapter 5, has been implemented only for the multimodal case (**?**, **?**), where components are rooted trees and not more general graphs. Moving from trees to graphs amounts to a non-trivial (to say the least) change in the code base. One promising avenue of research would be to write a proof net module for an existing graph rewrite tool. There are a number of tools which use interaction nets (or some extension of it) as their basis (**?**, **?**), so these would — at least a priori — have the required functionality.

Neural proof nets as graph neural networks The neural proof nets of Section **??** integrate proof net proof search with modern deep learning methodology. As an alternative to the approach of **?**, it seem promising to use one of the many variants of graph neural networks (**?**, **?**) for our neural theorem prover.

Applications in natural language entailment Wide-coverage semantics, as discussed in Chapter **??**, is a first step in a treatment chain for classical tasks in natural language processing, such as entailment. There are several possible ways to combine the output of our wide-coverage parser with a ‘standard’ theorem prover for first-order or higher-order logic, and use this combination for natural language entailment tasks[4]. There are several ways

[3]The additives, together with associativity, also allow us to compute intersections of string languages, which is a powerful operation in formal language theory since it moves us outside of the mildly context sensitive formalisms (**?**).

[4]Many entailment datasets have logically strange notions of entailment and contradiction. For example, “two dogs sit” is assumed to *contradict* “three cats play” (**?**) which would be a strange result for a logical system.

to do this. We can feed the formulas representing the meaning to an off-the-shelf theorem prover and hope it terminates (**?**). We can also program a set of tactics suited to natural language inference in a general-purpose proof assistant such as Coq, as done by **?**. Finally, we can restrict ourselves to a decidable logical fragment and use this to calculate entailments (**?**).