Tableau Algorithms for Categorial Deduction and Parsing

Saturnino Luz

PhD Thesis

University of Edinburgh

October 1997

Declaration

I declare that this thesis has been composed by myself and that the research reported here has been conducted by myself unless otherwise indicated.

Saturnino Luz

Edinburgh, November 11, 1998

Acknowledgements

I would like to thank my supervisor Jo Calder for his support, for providing much useful advice and feedback, and for his patience and dedication in sometimes doing so under adverse temporal and geographic constraints.

This thesis benefited from discussions with Robin Cooper (my former supervisor), Massimo Poesio (my second supervisor), the members of the non-standard and applied logic group and the participants of the dialogue workshops at the Centre for Cognitive Science. I would like to extend special thanks to my former office mate Patrick Sturt (now "on the other side") for introducing me to categorial grammar and providing feedback on my drafts.

I thank my office mates David Tugwell and Siamak Rezaei for (nearly always) putting up with my disorganisation, Kentaro Takahashi for a last minute comment on the thesis, and my brother Ricardo for acting as an interface between Brazil and me.

My parents, Saturnino Luz and Maria Carvalhaes, also deserve thanks for their tolerance while I was home and constant support since I came to Edinburgh. So does my friend Vania, who have always believed in me; even when I didn't. So do the Lola family, whose kindness and warm hospitality provided me with the excellent environment in which I completed part of my writing. Of them I would like to single out Theodora Lola for special thanks: for her love and support, for cheering me up when the PhD depression struck, and for keeping me company in the lab until three o'clock in the morning for a couple of days over the last two weeks.

Abstract

This thesis investigates the computational properties of a class of substructural calculi, those located between the non-associative Lambek calculus and the implicational fragment of intuitionist logic, which have been used for linguistic description and parsing. The investigation is set in the context of labelled analytic deduction. Parsing in Lambek calculi and labelled approaches to generalised deduction in most substructural logics have both been shown to be costly, and in fact intractable in many common applications. In this thesis we develop automated deduction mechanisms designed to keep complexity of categorial parsing under control while preserving the levels of uniformity and coverage one finds in labelled deductive systems.

First, we define the hierarchy of calculi whose computational treatment is addressed in the thesis, review the main issues and linguistic motivations behind proof-theoretical features of each calculus and describe the correspondence between proofs and semantic interpretation with respect to lambda terms.

Next we introduce the rules and algorithms of a deductive system based on analytic tableaux which covers the whole hierarchy of categorial calculi presented. Completeness and termination results are shown. We then impose syntactic constraints on the calculi and elaborate label unification procedures aimed at limiting the system's complexity. Alternative proof-search strategies are discussed and a technique for recovering syntactic structure from tableau derivations is developed.

In the last chapters we compare our system with other methods used in categorial deduction, discuss design issues, heuristics and extensions, and link categorial deduction with theorem proving in recently developed logics of information flow such as channel theory.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis we discuss analytic deduction techniques for automated theorem proving in categorial logics and its tailoring to perform parsing in categorial grammars. The discussion is set in the framework of labelled tableaux and is intended as a contribution to the study of algorithmic properties of labelled categorial systems.

This is, to our knowledge, the first attempt to apply tableau techniques to this kind of task. Tableau and labelled deduction have been used in theoretical approaches to substructural logics and automated deduction techniques have previously been employed in categorial grammar parsing. However, one encounters severe complexity problems when trying to implement general labelled systems or develop theorem provers to cover the variety of calculi that categorial grammar writing seems to demand. The work presented here seeks to provide a feasible application for the former and an environment for the implementation of the latter. The thesis is focused on issues which are specific to categorial deduction but which are also of interest for theorem proving tasks in the domain of information-oriented logic.

A full, general decision procedure is defined which enables us to cover a whole hierarchy of categorial calculi via setting of algebraic constraints, without having to implement separate provers for each calculus. Techniques which guarantee tractability in linguistically-relevant categorial subsystems are presented and tested in practice at the level of implementation.

Comparisons with other approaches to categorial deduction are made and the requirements of this kind of system are discussed. Towards the end of the thesis the issue of efficient categorial parsing develops into an account of type polymorphism within the tableau framework and an analysis of the connections between theorem proving in Lambek calculi and more general logics of information flow.

In what follows we overview the history of categorial grammar, situate the aims of this thesis with respect to previous research and describe its structure to set the scene for later chapters.

1.1 A brief historical excursion

This section is meant as brief overview of the developments in the field categorial grammar and categorial logics over a period of almost 70 years. The reader is referred to (van Benthem, 1986; van Benthem, 1991; Buszkowski, 1986) for more detailed discussions and pointers to the relevant literature.

Categorial grammar (CG for short) is an approach to language description in which all syntactic information is encoded in the lexicon. Lexical items are put in correspondence with *logical types* whose structure essentially determines which combinations are classified as grammatical and which are ruled out. This approach has its origins in logical semiotics and can be traced back to Frege and Husserl. The first formalisations of categorial grammar are due to (Ajdukiewicz, 1935) — which introduces an algorithm to decide well-formedness of an arbitrary string in Lesniewiski's system of "semantical types" — and (Bar-Hillel, 1953). We refer to them as the AB calculus. The symbols involved in these systems receive interpretations which are independent of any particular language and therefore the grammar may be regarded as a form of propositional calculus. Still stronger similarity with the propositional calculus is exhibited by the logic introduced by Lambek (Lambek, 1958) which extends the deductive apparatus of the AB calculus up to a logical system whose presentation resembles that of intuitionist logic.

After these pioneering works categorial grammars received little attention in the formal lin-

guistics community for many years — then dominated by Chomsky's *transformational* approach. With the advent of Montague's semantic enterprise (Montague, 1974; Dowty, 1988), CG started to attract more research (Geach, 1972), experiencing a strong revival in the 80's with the works of Ades and Steedman (Ades and Steedman, 1982; Steedman, 1987), Flynn (Flynn, 1983) Buszkowski and van Benthem (van Benthem, 1986; Buszkowski, 1986) among others. Many variants of the original systems were proposed to tackle different phenomena. The renewed interest in *substructural logics* — i.e logics which treat formulae a "resources" and are sensitive to, for instance, the order of premises in a deductive step — brought about by *linear logic* (Girard, 1987) within theoretical computer science added a new impulse to the strictly logicalist branch of CG. Linguistic motivations for calculi situated beyond the original Lambek calculus in deductive power were pointed out (van Benthem, 1991) and several versions of Lambek calculi received finer-grained mechanisms for structural control, such as the ones proposed by the Edinburgh-based group (Barry and Morrill, 1990). These have achieved a stage of maturity such that comprehensive presentations of CG systems covering substantial aspects of natural language and their connections with logic and type theory are starting to appear (Morrill, 1994; Carpenter, 1997).

These developments situate CG within the field of new *unified* approaches to logic, computational and cognitive processes based on a notion of *information flow* which encompasses works in semantics (Barwise, Gabbay, and Hartonas, 1995; Allwein and Dunn, 1993), proof-theory (Gabbay, 1994) and informatics (van Benthem, 1996; Girard, Lafont, and Regnier, 1995).

1.2 Aims of this thesis

Of the so called "lexicalist" approaches to grammar description, categorial grammar seems to be one which best encompasses the elements of the paradigm in its purest form:

- characterisation by types at the lexical level
- functor-argument structure
- clear and elegant compositional syntax-semantics interface

It has also been claimed that CG offers the possibility of *decoupling* the theory of parsing from the theory of competence. Furthermore, if one narrows the range of categorial systems down to the family logics started by Lambek (Lambek, 1958; Lambek, 1961) and augmented by the logics derived from those by means of deductive extensions towards intuitionist systems one also achieves a pure realisation of the paradigm of "parsing as deduction" (Shieber, Schabes, and Pereira, 1994). With only minor additions theorem proving in Lambek logics becomes (equivalent to) parsing in categorial grammar. In spite of this, although linguistic research has seen considerable theoretical activity in the field of categorial grammar[1], this research has failed to have the impact one would expect from a framework with such characteristics in the mainstream of computational linguistics[2]. We believe one of the reasons for this to be the disproportionate amount of research effort invested into the logical properties of these calculi in comparison with the research into the complex automated deduction techniques which these calculi seem to demand. This thesis is the result of our efforts to tackle the latter issue in a systematic manner.

There are several systems of CG. The ones in which we will be mostly interested here are those developed around the original Lambek calculus and its non-associative variants. We will not deal, for instance, with combinatory categorial grammar (CCG) or weaker combinatory systems such as the one introduced by the pioneering works of Ajdukiewicz and Bar-Hillel. All calculi addressed in this thesis share the property of being describable by Gentzen sequent systems which enjoy a cut-elimination property and therefore provide an effective method for deciding grammaticality. A number of calculi can be obtained from the original Lambek calculus by varying the degree to which the logic is sensitive to order and quantity of strings. From these, other calculi can be obtained by refining the constraints on order and number even further and/or combining them into more expressive, hybrid frameworks. The CG community does not seem to have reached a consensus regarding the degree of such resource sensitivity needed in natural language description or which hybrid framework would be the most adequate for the task. There is however an agreement, or at least a common working hypothesis, as

[1] In addition to the fact that the categorial approach has inspired increasingly more popular theories such as HPSG, which present themselves as alternatives to Chomsky's GB framework. Unlike GB, both HPSG and CG treat dependency in a purely *non-transformational* way.

[2] We refer mainly to systems which preserve the core logical characteristics set forth in (Lambek, 1958) as opposed to systems based on purely combinatory techniques.

to the correspondence between certain logical (structural) operations and certain natural language phenomena. In this thesis we will focus on these common logical features rather than on a particular hybrid system.

In his recent work on linear logic programming deduction for categorial logics, Glyn Morrill states that

> Automated deduction for Lambek Calculi is of interest in its own right but solution of the parsing problem for categorial logic allowing significant linguistic coverage demands automated deduction for more than just individual calculi. There is a need for methods applying to whole classes of systems in ways which are principled and powerful enough to support the further generalisations that grammar development will demand. (Morrill, 1995b)

We tend to endorse this view. In fact, the work to be presented in the next chapters is intended as a contribution to this research programme. An important point not made explicit in the statement above must be emphasized, however. Automated deduction techniques for CG should aim at efficiency and manageable computational complexity as well as generality. Efficiency and generality are two aspects which normally pose a trade-off for practical systems. In this thesis we will express this trade-off between coverage of a wide range of logics and tractability in the framework of tableau-based labelled deduction. We will borrow labelling techniques developed originally for full (generally intractable) substructural logics (Gabbay, 1994; D'Agostino and Gabbay, 1994) and adapt them to the specific case of linguistic description in CG, showing that a compromise may be achieved which does not impair the logical features of the Lambek calculi and opens up the possibility for the use of CG in efficient natural language applications.

The present work will focus on parsing issues mainly from a syntactic perspective. Although we recognise their importance in CG, semantic issues such as the labelling of derivations arising from the Curry-Howard isomorphism will only be discussed to the extent that they have an impact on proof search itself — for instance, in the case of *spurious ambiguity* (Hepple, 1990).

We will concentrate on theorem proving techniques for the core structural features of the class of a hierarchy of Lambek calculi to be defined in chapter 2 aiming at modularity and scalability. Structural modalities (Morrill et al., 1990; Hepple, 1990; Versmissen, 1994) and hybrid operators (Moortgat and Oehrle, 1993; Hepple, 1995) will not be directly addressed but we suggest in chapter 5 that the theorem proving mechanisms developed here for the Lambek hierarchy can be straightforwardly extended to deal with them.

Finally, we should remark that we will not attempt to provide an account of any particular kind of linguistic phenomena. However, unlike most approaches to automated deduction in categorial logics we will keep in mind that our primary application domain is natural language processing and that the deductive system to be presented should benefit from this fact by incorporating domain-specific knowledge.

1.3 Synopsis of chapters

The core of the thesis is presented in five main chapters followed by a concluding summary and appendices containing sample categorial proofs generated by the prototype implementing the techniques discussed in the main chapters and an analysis of the system's run-time profile. In all chapters we have tried to start by giving an informal overview of the issues to be tackled before moving on to the more formal presentation. The content of the main chapters is arranged as follows:

Chapter 2: Mathematical, linguistic and computational backgrounds

This chapter introduces the framework of CG in general terms intercalating logical and algebraic tools with the natural language phenomena which motivates their introduction. The basic type syntax to be used in the following chapters is defined here. We then overview Ajdukiewicz/Bar-Hillel's systems, the Lambek calculus and its cut-elimination result. There follow a discussion of the role played by structure and resource sensitivity in the notion of grammaticality posited by Lambek calculi, the definition of a hierarchy of categorial calculi

with respect to proof structure and a survey of the algebraic and relational semantics for the calculi along with the Curry-Howard isomorphism. Issues briefly mentioned in this chapter include polymorphic types and the recognising power of CGs. We close the chapter by introducing the issue of parsing as deduction in Lambek calculi and commenting on automated deduction approaches to categorial logics.

Chapter 3: Automated Substructural Deduction for CG

Here we describe our tableau-based approach to categorial deduction, presenting the general architecture of the parsing model distinguish between two main modules: *syntactic tableau expansion* and *labelling algebra manipulation*. We focus on tableau expansion rules and algorithms presenting a generalised decision procedure for the family of logics defined in chapter 2, and sketching soundness completeness results.

Chapter 4: Syntactic Structure and Labelling

In chapter 4 we deal mainly with issues related to the labelling algebra. We introduce the bookkeeping strategies adopted by the system, discuss spurious ambiguity arising from combinatorial features and point out their relationship with one of the most distinctive features of the tableau system employed in this thesis: variable introduction in the labelling algebra via tableau branching rules. Our goal here is to present a study of how the system's efficiency would change by varying the division of labour between its two main modules. This is accompanied of the presentation of a technique to *recover* syntactical information from proof trees and time complexity results.

Chapter 5: Redundancy in Labelled and non-Labelled CG Deduction

In this chapter we return to the issue of redundancy in categorial proof search, this time in sequent-based and proof-net implementations. We compare the manifestations of the phenomenon in our tableau system with strategies to deal with it in other approaches. The

discussion is set against a background of general theorem proving rather than CG-specific applications.

Chapter 6: Polymorphism and Information Flow

We close the body of thesis with a more speculative chapter which addresses two extensions of the framework: quantification — or the handling of polymorphic types in CG — and theorem proving in the general *information networks* of (Barwise, Gabbay, and Hartonas, 1994; Barwise, Gabbay, and Hartonas, 1995).

Chapter 2

Mathematical, linguistic and computational background

In this chapter we introduce the basic formal apparatus and terminology of categorial grammar, describe its linguistic features and motivations, demonstrate the main mathematical properties of the formalism and introduce the issue of parsing as theorem proving in CG. The chapter has the format of a general overview of the main categorial systems rather than an in-depth analysis of any particular system. Emphasis has been given to structural properties whereby a hierarchy of calculi encompassing the main logical features to be addressed in the remaining of this thesis is defined. Towards the end of the chapter we present an overview of early implementations of parsers for Lambek systems, pointing out the main problems to be dealt with in this thesis.

2.1 Initial setup: the AB calculus

We mentioned that in categorial grammar a great deal of syntactic information is encoded in the lexicon. The (logical) proof theory is thus in charge of determining how lexical items combine to build up complex structures. In what follows we define the basic machinery which

allows us to encode such information.

Lexical entries (words) are ascribed to *syntactic types*[1] which describe, encode a word's function. Types can be primitive, such as "NP", "PP", "AP" etc, or built from primitive types through binary operators (connectives) to form complex types such as "S/NP" "$(NP\backslash S)/NP$". We call the number of connectives in a type the *degree* of that type and define the set of types in definition 2.1.

Definition 2.1 *The set of of well-formed types, $\mathcal{C}$, is the closure of the set of primitive types $\mathcal{P} = \{A, B, C, N, NP, AP, PP, ...\}$ (with or without subscripts) under the following rules:*

2.1(i) If $X \in \mathcal{P}$ then $X \in \mathcal{C}$

2.1(ii) If $X \in \mathcal{C}$ and $Y \in \mathcal{C}$, then $(X * Y) \in \mathcal{C}$, where $* \in \{/, \backslash, \bullet\}$. We normally omit the outermost brackets.

It should be remarked that in most linguistic applications of the syntax specified in definition 2.1 the symbol "$\bullet$" does not occur in types assigned to lexical entries but only in formulas built from these types. In addition, as noted in (Zielonka, 1981), occurrences of formulae of the form $(X \bullet Y)/Z$ are often limited to intermediary steps of derivations, a fact which has been exploited by (Cohen, 1967) in the definition of a strictly product-free calculus.

The most basic form of combination of syntactic types is *function application*. For instance, an NP (a noun phrase such as John) could combine with a type NP\S (an intransitive verb such as sleeps), yielding the sentence S: John sleeps. This will be represented by: NP $\bullet$ NP\S $\vdash$ S, where $\vdash$ stands for syntactic entailment, which will vary according to the characteristics of the logic being used. Operators on syntactic types are called *functors* and the elements they combine with (the elements appearing under the division bars) are called *arguments*. The rule below summarises this:

[1] We follow (Lambek, 1988) in calling our formulae "types" as opposed to "categories" (Moortgat, 1988) in order to avoid confusion with the usage of the latter in Category Theory (MacLane, 1971). Our usage of the term also agrees with recent systematisations in the area of *type-logical* syntactic and semantics analysis of natural language — e.g. (Morrill, 1994), (Carpenter, 1997).

$$\underline{\textit{Application}:} \qquad \begin{array}{l} X/Y, Y \vdash X \\ Y, Y\backslash X \vdash X \end{array} \tag{2.1}$$

Rule (2.1) plus identity, (2.2) below, suffice to characterise the weakest categorial system; the first to be introduced (Ajdukiewicz, 1935; Bar-Hillel, Gayfman, and Shamir, 1960) which is motivated by linguistic considerations: the calculus aptly named AB.

$$\underline{\textit{Identity}:} \qquad X \vdash X \tag{2.2}$$

2.1.1 Extending the basic machinery

Although linguistically sound, the system defined above is too weak to cope with a variety of phenomena. In (Luz and Sturt, 1995), we note that much of the interest in using categorial grammars for linguistic research derives from the possibilities they offer for characterising a flexible notion of constituency, and also that this has been found particularly useful in the development of theories of coordination, and incremental interpretation. Consider the following right node raised sentence (Moortgat, 1988), for example:

(e.1) [John resents $_{S/NP}$] and [Peter envies $_{S/NP}$] Mary

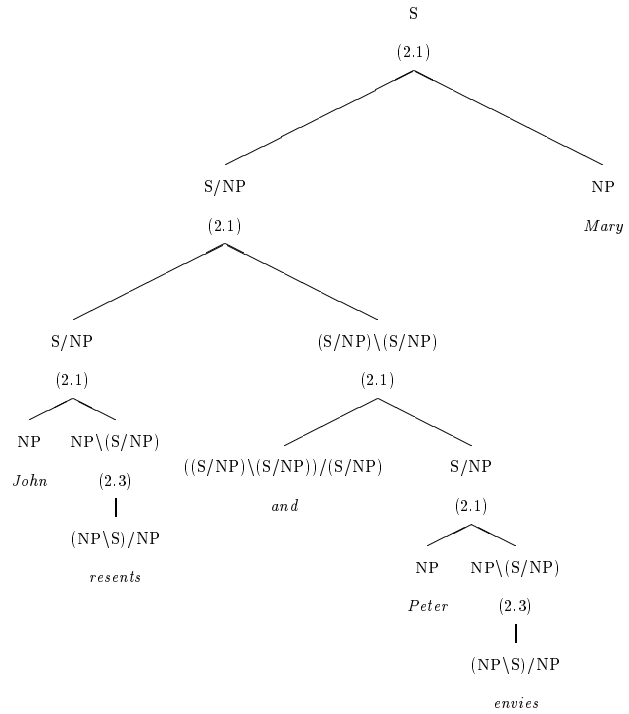
Under standard lexical type assignments, in which transitive verbs are assigned the type $(NP\backslash S)/NP$, (e.1) cannot be derived in AB. In order for the sentence to be derivable, a new rule must be used. The associativity rule shown in (2.3) plays the required role.

$$\underline{\textit{Associativity}:} \qquad \begin{array}{l} (Z\backslash X)/Y \vdash Z\backslash(X/Y) \\ Z\backslash(X/Y) \vdash (Z\backslash X)/Y \end{array} \tag{2.3}$$

In a system which includes associativity, with each conjunct assigned the type indicated in square brackets, the sentence of example (e.1) will receive the derivation shown on the tree

in (e.2)[2].

(e.2)



A calculus which includes composition, (2.4), will allow a function to apply to an unsaturated argument, and it is this property which allows Ades and Steedman (Ades and Steedman, 1982) to treat long distance dependencies, and motivates much of Steedman's later work on incremental interpretation.

$$\underline{\textit{Composition}:} \qquad \begin{array}{l} X/Y \bullet Y/Z \vdash X/Z \\ Z\backslash Y \bullet Y\backslash X \vdash Z\backslash X \end{array} \tag{2.4}$$

Even more drastic examples of non-constituent coordination can be handled if a rule of

[2]The numbers in brackets which occur on certain nodes indicate the rules applied to the daughter(s) so as to derive the mother node.

lifting, (2.5), is added to the above.

$$\underline{Lifting:} \qquad \begin{array}{l} X \vdash Y/(X\backslash Y) \\ X \vdash (Y/X)\backslash Y \end{array} \tag{2.5}$$

Dowty (Dowty, 1988) uses the combination of (2.4) and (2.5) to derive (e.3). However, it has been argued that the power which gives non-applicative categorial grammar its notion of flexible constituency also has to be constrained. For example, as Pickering and Barry point out (Pickering and Barry, 1993), a system which includes lifting and composition will allow ungrammatical coordinations such as (e.4), since both of the bracketed strings can be assigned the type S/(NP\S).

(e.3) John gave Mary a book and Susan a record.

(e.4) *[I believe that John $_{S/(NP\backslash S)}$] and [Mary $_{S/(NP\backslash S)}$] is a genius

This leads them to propose the Dependency Categorial Grammar calculus D, which forbids lifting through a global constraint on derivations. The resulting notion of constituency (Dependency Constituency) is shown to have applications not only in the description of coordination phenomena, but also in modelling certain aspects of human sentence processing, in particular allowing for a categorial characterisation of the notion of *head-driven* parsing. Milward (Milward, 1995) shows that Composition can result in over-generation, allowing ungrammatical sentences such as

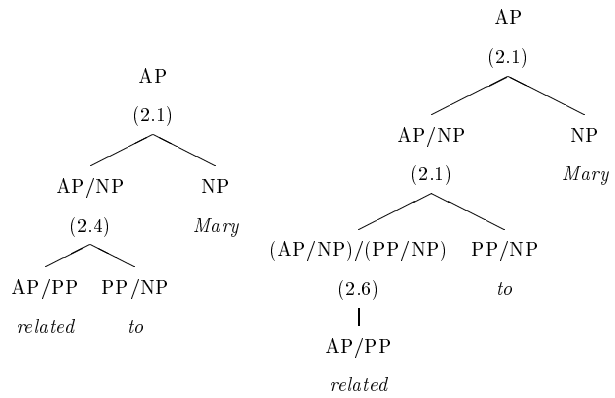
(e.5) *Children [reluctantly $_{(NP\backslash S)/(NP\backslash S)}$] [who came from far away $_{NP\backslash NP}$] [arrived $_{NP\backslash S}$].

Assuming the types as indicated, the problem is that the relative clause who came from far away can combine with the intransitive verb arrived through backward composition, to derive a new constituent of type NP\S, which can then be modified by the predicate adverb reluctantly, through backward application. Problems of over-generation such as these are part

of the motivation for Milward's definition a categorial grammar, AACG, which is equivalent to AB plus associativity (Milward, 1995). Ignoring over-generation problems for the time being, we add to the set of rules the division schemes (2.6) which permit an AP such as related$_{AP/PP}$ to$_{PP/NP}$ Mary$_{NP}$ to receive an alternative left-branching analysis by means of purely *unary rules* — i.e. rules in which the operator "•" does not appear — in addition to function application (2.1). (Moortgat, 1988) remarks that this kind of unary analysis might favour incremental interpretation. Compare the two left-branching derivations shown in (e.6).

$$\underline{Division(mainfunctor):} \qquad \begin{array}{l} X/Y \vdash (X/Z)/(Y/Z) \\ Y\backslash X \vdash (Z\backslash Y)\backslash(Z\backslash X) \end{array} \tag{2.6}$$

(e.6)



Finally, division can also occur on the subordinate functor as in (2.7) turning it into a higher-order functor:

$$\underline{Division(subordinatefunctor)} \qquad \begin{array}{l} X/Y \vdash (Z/X)\backslash(Z/Y) \\ Y\backslash X \vdash (Y\backslash Z)/(X\backslash Z) \end{array} \tag{2.7}$$

It is possible to define a hierarchy of logical calculi, each of which admits one or more of (2.1)–(2.7) as theorems; from the purely applicative calculus AB, of Ajdukiewicz and Bar-

Hillel, which supports only (2.1), to the full Lambek calculus L, which supports all the above laws. Calculi apparently intermediate in power between AB and L have been explored (e.g. Dependency Categorial Grammar (Pickering and Barry, 1993)), as well as stronger calculi. Although (2.1)–(2.7) may be regarded as theorems of the L, they do not suffice to characterise the calculus. In fact, it has been shown (Zielonka, 1981) that no extension of (2.1) in the form of a finite number of cancellation schemes is equivalent to L. It is possible, however, to define a Gentzen system which describes the calculus.

2.2 Gentzen presentation of L

Rules (2.1)–(2.7) describe incompletely the behaviour of the entailment relation "⊢". The next step is to present a procedure which will enable us to verify, given an entailment relation, whether or not that relation holds between formulae of the particular categorial system being considered. This is done by means of *sequents*, a device used by Gentzen in his intuitionist logical system (Gentzen, 1969). A sequent is a pair (Γ, Δ) of finite, possibly empty, sequences of types between which an entailment relation holds — i.e. relations of the form $\Gamma \vdash \Delta$, where $\Gamma = [A_1, ..., A_m]$ and $\Delta = [N_1, ..., A_n]$. In L the further requirement that $n = 1$ is enforced. In a sequent, Γ is called the *antecedent* while Δ is called the *succedent*. We denote sequences of types by capital Greek letters (Γ and Δ non-empty) and use commas to denote type juxtaposition. The following sequent rules were proposed in (Lambek, 1958) to describe the behaviour of "⊢" in the calculus named after Lambek:

$$
\begin{array}{ll}
(\mathrm{R}/)\dfrac{\Delta, A \vdash B}{\Delta \vdash \mathrm{B/A}} & (\mathrm{L}/)\dfrac{\Gamma \vdash C \quad \Psi, A, \Phi \vdash B}{\Psi, A/C, \Gamma, \Phi \vdash B} \\[2ex]
(\mathrm{R}\backslash)\dfrac{A, \Delta \vdash B}{\Delta \vdash A\backslash B} & (\mathrm{L}\backslash)\dfrac{\Gamma \vdash C \quad \Psi, A, \Phi \vdash B}{\Psi, \Gamma, C\backslash A, \Phi \vdash B} \\[2ex]
(\mathrm{L}\bullet)\dfrac{\Psi, A, C, \Phi \vdash B}{\Psi, A \bullet C, \Phi \vdash B} & (\mathrm{R}\bullet)\dfrac{\Delta \vdash A \quad \Gamma \vdash C}{\Delta, \Gamma \vdash A \bullet C} \\[2ex]
(\mathrm{cut})\dfrac{\Gamma \vdash A \quad \Psi, A, \Phi \vdash C}{\Psi, \Gamma, \Phi \vdash C} & (\mathrm{Id})\dfrac{}{A \vdash A}
\end{array}
\tag{2.8}
$$

Semantically, these operators correspond to the operations of right division (/), left division (\) and multiplication (•) on the subsets of a semigroup M (Lambek, 1988), as shown in (2.9)–(2.11). The calculus has been shown to be complete with respect to this free semigroup interpretation in (Pentus, 1994b)[3] — i.e. if the $X \vdash Y$ can be derived through the system of rules (2.8) then the following is the case with respect to the algebraic structure of (2.9)–(2.11): $[\![X]\!]^M \subseteq [\![Y]\!]^M$, and vice-versa. If M is a non-associative multiplicative system instead of a semigroup, then we obtain the non-associative calculus NL.

$$A \bullet B = \{x \bullet y \in M | x \in A \wedge y \in B\} \tag{2.9}$$

$$C/B = \{x \in M | \forall_{y \in B} x \bullet y \in C\} \tag{2.10}$$

$$A\backslash C = \{y \in M | \forall_{x \in A} x \bullet y \in C\} \tag{2.11}$$

Returning to (2.8), proofs in Gentzen systems can be interpreted "bottom-up" as starting off with axioms, (Id)s, and constructing the sequent one wants to prove via finite number of applications of (R/)–(CUT) — notice a potential source of confusion here: "bottom-up" actually refers to the way the sequents are built rather than the orientation in which proofs are displayed in Gentzen notation.

Perhaps a more practical way of viewing a derivation is "top-down". Sequent rules are interpreted top-down as breaking the formulae into progressively smaller ones (i.e. formulae of smaller degree) until each leaf contains either an (Id) sequent or a sequent to which no rules can be applied. Pushing this interpretation into a method of proof by refutation, we can assume the formulae in the antecedent to have *positive* polarity while succedent types receive *negative* polarity. Connectives "/" and "\" can thus be seen as forms of left and right implication respectively and "•" as a form of conjunction. A rule such as (R/) under this framework will be then be read top-down as saying: if B/A has negative polarity, then A is assigned positive polarity *and* B negative polarity, provided that the structure of Δ is preserved. (L/) will read: if A/C is positive then *either* C is negative *or* A is positive,

[3] Buszkowski (Buszkowski, 1986) gives a completeness proof for the product-free calculus

with the same proviso as in (R/), and so on. As noted in (Fitting, 1990), if polarities are interpreted as Boolean values this approach corresponds to generating counter models of the initial sequent as in semantic tableaux (notice that (Id) expresses a contradiction in this kind of interpretation). An example of a proof in a Gentzen system is the proof of (2.3), given in example (e.7).

(e.7)
$$\dfrac{\dfrac{C \vdash C \qquad \dfrac{A \vdash A \qquad B \vdash B}{A, A\backslash B \vdash B}\,(L\backslash)}{A, (A\backslash B)/C, C \vdash B}\,(L/)}{\dfrac{A, (A\backslash B)/C \vdash B/C}{(A\backslash B)/C \vdash A\backslash(B/C)}\,(R\backslash)}\,(R/)$$

Decidability of L was first proved in (Lambek, 1958). Since, apart from (CUT), all rules in (2.8) obey the subformula principle, i.e. the resulting formulae contain only subformulae of the formulae to which the rule is applied, it suffices to show that cut can be eliminated in order to prove that given a sequent, a finite number of applications of the rules terminates with a positive or negative answer to whether the sequent is a theorem of L or not. Lambek's result (theorem 2.1) therefore shows that the set of theorems of L does not decrease if (CUT) is eliminated[4].

Theorem 2.1 (Cut elimination) *Any sequent derivable in the system* (2.8)} *is also derivable in the same system without (CUT).*

Proof. The proof is obtained by defining the *complexity* of a cut as the sum of the degrees of the formulae and sequences occurring in it and then showing by induction that any cut can be either removed or replaced by a cut of smaller complexity. Since cuts can never have negative complexity we conclude that all occurrences of (CUT) can be eliminated from any derivation. ■

[4]Whether or not this is a sensible thing to do in automated deduction is a different issue which will be discussed in some detail in chapter 3.

2.3 Lexical items as logical resources

Analysis of the system presented in (2.8) shows that classical theorems of standard logic do not hold in the Lambek calculus. Modus ponens, for example loses its commutative character: interpreting \ as standard implication we would be able to derive a sequent such as (e.8.a) which does not hold in L. For the same reason (e.8.b) fails in L even though it is derivable in standard propositional logic.

(e.8) a. $A\backslash B, A \vdash B$

b. $(A\backslash B)\backslash C \vdash B\backslash(A\backslash C)$

(e.9) a. $A, A, A\backslash B \vdash B$

b. $A \vdash A\backslash A$

(e.10) $A\backslash(A\backslash B) \vdash A\backslash B$

This shows that order is relevant in L, a property which is supposed to reflect a characteristic of natural language syntax: namely, the one which says that we cannot to change the positions of our words in a sentence and always end up with a grammatical construct. Other theorems of propositional logic which are non-theorems in L are (e.9) and (e.10). The former are provable in logics which allow a sequent to be *expanded*, where (b) exhibits a form of type raising which would enable unrestricted duplication of lexical items and therefore is not allowed in L. The latter would permit arbitrary deletion of words, which seems to be equally undesirable.

2.3.1 Categorial logics as (bi)linear logics

If one compares Gentzen's system for the implicational fragment of intuitionist logic with the system defined in (2.8) one realises that the sequent rules of the latter are a subset of the rules of the former. In fact, (2.8) corresponds to the *operational rules* of a Gentzen sequent system (Gentzen, 1969). Operational rules are those which describe the behaviour of logical operators with respect to the entailment relation. What is missing is precisely the so called *structural*

rules — that is, the schemes which govern the proof steps required to derive theorems such as (e.8)–(e.10). These rules are shown in (2.12).

$$\text{(P)}\ \frac{\Phi, A, B, \Psi \vdash C}{\Phi, B, A, \Psi \vdash C} \qquad \text{(E)}\ \frac{\Phi, A, \Psi \vdash B}{\Phi, A, A, \Psi \vdash B}$$

$$\text{(C)}\ \frac{\Phi, A, A, \Psi \vdash B}{\Phi, A, \Psi \vdash B} \qquad \text{(M)}\ \frac{\Phi, \Psi \vdash B}{\Phi, A, \Psi \vdash B} \tag{2.12}$$

If structural properties are taken into account, then lexical items can be viewed as resources which will be "consumed" during the parsing of the sentence, which by its turn may be regarded as a proof. This is in total agreement with the paradigm of "parsing as deduction" (Shieber, Schabes, and Pereira, 1994). Furthermore, from this point of view it is fair to regard the Lambek-style categorial grammar as a precursor of Girard's linear logic (Girard, 1987; Girard, Lafont, and Regnier, 1995) with bidirectional implication, a "bilinear logic" as (Lambek, 1995) calls it[5].

2.3.2 Linguistic aspects of proof structure

Van Benthem (van Benthem, 1988) and Moortgat (Moortgat, 1988) discuss linguistic motivations to increase the power of L by adding the structural transformations *Permutation*, *Contraction* and *Expansion* and the logical aspects of doing so. The derived calculi are named: LP, LPC, LPE and LPCE. The structural transformation *Permutation*, which removes the restrictions on the linear order of types, allows us to go beyond the purely concatenative derivations of L. This allows us to deal with sentences exhibiting non-standard constituent order. For example, Moortgat suggests using permutation for dealing with *heavy NP-shift* in examples similar to the following (Moortgat, 1988):

[5]Except that Girard appeals to somewhat more mundane motivations than the discovery of universal language principles in his advocacy of structural control in logic. His examples include formalisation of computational processes and even cigarette vending machines.

(e.11) John gave [to his nephew $_{PP}$] [all the old comic books which he'd collected in his troubled adolescence $_{NP}$].

In (e.11), the bracketed constituents can be "rearranged" via permutation so that a derivation is possible that employs the standard type ((NP\ S)/PP)/NP for the ditransitive verb *gave*.

In L, while it is possible to specify a type missing an argument on its left or right periphery, it is not possible to specify a type missing an argument "somewhere in the middle", making it impossible to deal with non-peripheral extraction. However, as Morrill et al show, permutation provides the additional power necessary to account for this phenomenon (Morrill et al., 1990).

In addition to permutation, there are also linguistic examples which motivate contraction and expansion (Moortgat, 1988; Morrill, 1994) combined with (P). An example of syntactic transformation which is accounted for by means of deletion of types (in the sentence) via (C) is *gapping*, as shown in derivation (e.12)[6]. Notice that, read top-down, the transformation actually corresponds to duplication of types in a Gentzen proof — the types which undergo duplication (contraction, if read bottom-up) and change position are shown in boldface in (e.12).

(e.12)
$$\text{(C)}\ \frac{\text{(P)}\ \dfrac{\text{(L/)}\ \dfrac{\text{(L/)}\ \dfrac{NP \vdash NP \quad \text{(L\)}\ \dfrac{VP \vdash VP \quad \dfrac{NP \vdash NP \quad S \vdash S}{NP, NP\backslash S \vdash S}}{NP, (NP\backslash S)/VP, VP \vdash S}}{NP, ((NP\backslash S)/VP)/NP, NP, VP, \vdash S} \quad \text{(L/)}\ \dfrac{NP \vdash NP \quad \ldots}{NP, \ldots, S\backslash S \vdash S}}{NP, ((\mathbf{NP}\backslash \mathbf{S})/\mathbf{VP})/\mathbf{NP}, NP, \mathbf{VP}, (S\backslash S)/S, ((\mathbf{NP}\backslash \mathbf{S})/\mathbf{VP})/\mathbf{NP}, NP, \mathbf{VP} \vdash S}}{NP, ((\mathbf{NP}\backslash \mathbf{S})/\mathbf{VP})/\mathbf{NP}, ((\mathbf{NP}\backslash \mathbf{S})/\mathbf{VP})/\mathbf{NP}, NP, \mathbf{VP}, \mathbf{VP}, (S\backslash S)/S, NP \vdash S}}{\underset{\textit{Jo}}{NP},\ \underset{\textit{promised}}{((\mathbf{NP}\backslash \mathbf{S})/\mathbf{VP})/\mathbf{NP}},\ \underset{\textit{Mary}}{NP},\ \underset{\textit{to stop smoking}}{\mathbf{VP}},\ \underset{\textit{and}}{(S\backslash S)/S},\ \underset{\textit{Fred}}{NP},\ \underset{\textit{Sue}}{NP} \vdash \text{S}}$$

Examples of structural expansion are provided by the linguistic phenomenon of *right dislocation* (Moortgat, 1988) — see example (e.13) where two NP resources could satisfy the same

[6]Part of the branch on the right-hand side of this derivation has been omitted here due to lack of space. The missing part is identical to the left-hand side branch except for the extra application of (L\) which introduces S\S on the antecedent. For the same reason the two applications of (C) and (P) are shown as as single step rather than four.

function in the thematic structure.

(e.13)
$$\dfrac{\dfrac{}{NP \vdash NP}(\mathrm{L}/) \qquad \dfrac{\dfrac{}{AP \vdash AP} \qquad \dfrac{\dfrac{}{NP \vdash NP} \quad \dfrac{}{S \vdash S}(\mathrm{L}\backslash)}{NP, NP\backslash S \vdash S}(\mathrm{E})}{NP, (NP\backslash S), NP \vdash S}(\mathrm{L}/)}{\underset{\textit{He}}{NP}, \underset{\textit{considers}}{((NP\backslash S)/AP)/NP}, \underset{\textit{them}}{NP}, \underset{\textit{incompetent}}{AP}, \underset{\textit{those candidates who...}}{NP}} \vdash S$$

Finally, there is evidence for the relevance of a structural property which has not been explicitly stated in (2.8): associativity. As Morrill points out (Morrill, 1994), although associativity can be seen as a desirable property in cases where all possible bracketings of a sentence constitute the specification of its possible prosodic readings (Steedman, 1991), as in (e.14), it sometimes leads to impossible divisions, as shown in Steedman's example (e.15).

(e.14) a. (Bill) (thinks John walks).

b. (Bill thinks John) (walks).

c. (Bill thinks) (John walks).

(e.15) *Three mathematicians (in ten derive a lemma).

The system NL (Lambek, 1961) is a version of L in which associativity is strictly forbidden. Along with the other calculi described in this section NL define a hierarchy of linguistically motivated categorial logics which can be presented as in table 2.1.

2.3.3 Syntactic encoding of structural properties: modal operators

In spite of the theoretical significance of the "pure" substructural hierarchy summarised in table 2.1, it has been widely recognised that a system employing the unrestricted use of structural transformations would be far too powerful for any useful linguistic application. Arbitrary word order variation, copying and deletion are not characteristics which could be

		NL	$\subset$	NLP	$\subset$	NLPC NLPE	$\subset$	NLPCE
AB	$\subset$	$\cap$		$\cap$		$\cap$		$\cap$
		L	$\subset$	LP	$\subset$	LPC LPE	$\subset$	LPCE

Table 2.1: Hierarchy of Categorial Calculi

claimed to hold for any natural language. For this reason, a goal of current research is to build systems in which the resource freedom of the more powerful calculi can be exploited when required, while the basic resource sensitivity of L (or NL) is retained in the general case. In this section we briefly survey some of the approaches to achieving this goal. A warning should be given here: the area is still undergoing intense research and therefore no complete agreement has been reached which could make possible a comprehensive presentation. Good surveys on linguistically motivated calculi are found in (Morrill, 1994) and in (Moortgat, 1994a; Moortgat, 1995). On linear and substructural logics in general, including model-theory, with applications to other branches of computer science see (Girard, Lafont, and Regnier, 1995) and (van Benthem, 1996).

Edinburgh structural modalities

The most straightforward way to reintroduce controlled structural operations is to allow structural operators in the syntax. In linear logic the power of intuitionist and classical logic is regained by means of "exponentials" ("!" and "?") which deal with monotonicity, described by Girard as the rule which "opens the door for fake dependencies" and contraction (C), "the fingernail of infinity in propositional calculus" (Girard, 1995). A similar approach is adopted in (Morrill et al., 1990; Hepple, 1990), where modal operators, the so-called *Edinburgh structural modalities*, explicitly mark those types which are permitted to be manipulated by specific structural transformations. In Lambek systems which do not enjoy (P), a further modality to allow word order to be commuted is introduced. For example, consider the Gentzen rules (2.13) describing a modality $\triangle$ which licenses permutation in L (the symbol

$(\Gamma)^{\triangle}$ stands for a sequence of formulae, each of which preceded by $\triangle$).

$$(\triangle p)\frac{\Phi, \triangle A, B, \Psi \vdash C}{\Phi, B, \triangle A, \Psi \vdash C} \qquad (\triangle r)\frac{(\Gamma)^{\triangle} \vdash A}{(\Gamma)^{\triangle} \vdash \triangle A} \qquad (\triangle l)\frac{\Gamma, B, \Psi \vdash A}{\Gamma, \triangle B, \Psi \vdash A} \tag{2.13}$$

It is demonstrable that the Lambek calculus with the triangle modality enjoys cut-elimination and is therefore decidable. A semigroup interpretation for the system was first proposed in (Hepple, 1990) which is sound but not complete (Lincoln et al., 1990). Versmissen (Versmissen, 1994) proposes relaxing $(\triangle r)$ in (2.13) so as to make the resulting calculus a sound and complete one with respect to the algebraic semantics.

Example (e.16) shows the permutation modality in action (Morrill, 1994) in a case of *non-peripheral extraction*. Note that commutativity is signalled *by* the relative pronoun rather then *at* the noun ("book") itself.

(e.16)
$$(\mathrm{L}/)\frac{\begin{array}{c}(\ldots)\ \dfrac{\vdots}{NP, (NP\backslash S)/NP, \mathbf{NP}, (NP\backslash S)\backslash(NP\backslash S) \vdash S}\\ (\triangle l)\ \dfrac{NP, (NP\backslash S)/NP, \triangle\mathbf{NP}, (NP\backslash S)\backslash(NP\backslash S) \vdash S}{NP, (NP\backslash S)/NP, (NP\backslash S)\backslash(NP\backslash S), \triangle\mathbf{NP} \vdash S}\\ (\triangle p)\ \dfrac{}{(\mathrm{R}/)\ \dfrac{}{NP, (NP\backslash S)/NP, (NP\backslash S)\backslash(NP\backslash S) \vdash \mathbf{S}/\triangle\mathbf{NP}}}\end{array} \qquad (\mathrm{L}/)\frac{N \vdash NN \vdash N}{N, N\backslash N \vdash N}}{(\mathrm{L}/)\ \dfrac{N, (\mathbf{N}/\mathbf{N})/(\mathbf{S}/\triangle\mathbf{NP}), NP, (NP\backslash S)/NP, (NP\backslash S)\backslash(NP\backslash S) \vdash N \qquad NP \vdash NP}{\begin{array}{cccccccc} NP/N, & N, & (N/N)/(S/\triangle NP), & NP, & (NP\backslash S)/NP, & (NP\backslash S)\backslash(NP\backslash S) & & \vdash \mathrm{NP}\\ \textit{the} & \textit{book} & \textit{which} & \textit{Jo} & \textit{read} & \textit{today} & & \end{array}}}$$

As before we indicate the types which undergo movement across the proof tree in (e.16) by boldface.

Hybrid and embedding systems

Structural modalities can help define translations between weaker and stronger systems. The precursors of such translations have been the aforementioned renderings of linear, intuitionist and classical logics into one another. *Embedding logics* are categorial systems in which different forms of product ("$\bullet$), left residuation ("\") and right residuation coexist ("/"), the stronger forms being defined from the weaker ones by means of modalities (Hepple, 1994b; Moortgat, 1994b). In this kind of system, the extra power provided by the modalities is fused into operators which encode linguistic phenomena not captured by the embedded system on its own and made available for grammar specification. It is also possible to define *hybrid calculi* (Hepple, 1994a) by suppressing auxiliary modalities altogether in the final system of embeddings.

The discovery that one could tamper with proof structure to cope with linguistic phenomena unfortunately (or fortunately, some may say) led to a proliferation of modalities, product operators and implication lollipops within frameworks for lexical specification. Notwithstanding the need for structural flexibility which natural language seems to impose, the introduction of operators ought to be tamed if a useful (and usable) categorial architecture is to result. Attempts at defining "minimalist" frameworks motivated by universal principled linguistic assumptions are found in (Kurtonina and Moortgat, 1995) and (Morrill, 1994). The former describes proof and model theory for a lattice of resource-sensitive logics which is claimed to encompass the essentials of grammatical description: linear order, hierarchical grouping (constituency) and dependency. The latter presents a fuller treatment along the same lines which includes Montague-style semantics and draws parallels between the categorial programme and other linguistic frameworks (Morrill, 1994, pp 250–261).

In (Kurtonina and Moortgat, 1995) the properties mentioned above are captured by two general logical principles: *relaxation of structural constraints* (e.g. to allow certain resources to permute) and *control over type instantiation*. A case which exemplifies the need for licensing structural relaxation in the hierarchy of categorial calculi is given by observing the sentence in example (e.16). Type NP cannot be derived in plain L. However, while without the adverb an L–derivation for the concatenation of the remaining types becomes possible, it is still not possible to derive NP in NL. Kurtomina's and Moortgat's proposal is to enrich the type language with unary residuated operators whose semantics is defined in terms of binary, Kripke-style accessibility relations (see section 2.3.4) on frames (Došen, 1992). This approach enables bidirectional operators to be defined which account for the asymmetry between head and dependent constituents.

The result of this is that the hierarchy of calculi which can be reached from NL or L by means of translations and embedding is augmented with DNL, DNLP, DL and DLP, where D stands for "dependency". Completeness with respect to the relational semantics, soundness of embedding and cut-elimination have been proved for these calculi (Kurtonina and Moortgat, 1995).

2.3.4 Kripke-style frames and model theory

As seen above, one way to interpret (Lambek, 1988) the implicational and conjunctive operators of L (and NL) is as corresponding to right division (./.), left division (.\.) and multiplication (.•.) on the subsets of a semigroup M (non-associative multiplicative system, respectively). Other ways include the use of categories in *categorical* logic (Lambek, 1988; Lambek and Scott, 1988) and the relational semantics suggested by van Benthem (van Benthem, 1991) and shown to be complete in (Andreka and Mikulas, 1994). Since the former has not been explored so far in parsing applications we shall focus on the latter which has been employed in most of the systems mentioned above and can be seen as a generalisation of the semigroup semantics.

The interpretation is based upon a Kripke structure $\mathcal{R} =< W, R >$, where $R \subseteq W \times W \times W$ on which a valuation function v obeying the following conditions is defined:

$$\begin{aligned} v(A \bullet B) &= \{z \mid \exists x \exists y : Rzxy \wedge x \in v(A) \wedge y \in v(B)\} \\ v(C/B) &= \{z \mid \forall x \forall y : (Rxzy \wedge y \in v(B)) \Rightarrow x \in v(C)\} \\ v(A\backslash C) &= \{z \mid \forall x \forall y : (Rxyz \wedge y \in v(A)) \Rightarrow x \in v(C)\} \end{aligned} \tag{2.14}$$

The properties of the connectives are thus determined by the restrictions one imposes on $\mathcal{R}$. Obviously, the weaker the restriction imposed, the weaker the corresponding calculus. An unrestricted $\mathcal{R}$ describes NL, $\mathcal{R}$ obeying $\forall xyzu \in W : \exists t(\mathcal{R}xyt \wedge \mathcal{R}tzu) \Leftrightarrow \exists v(\mathcal{R}xyz \wedge \mathcal{R}xvu)$ (i.e. associativity) characterises L, if $\mathcal{R}$ is commutative we obtain NLP, and so on. Dependency–preserving features are expressed by defining two accessibility relations on W rather than one. A frame $\mathcal{R}' =< W, R^3_l, R^3_r >$ suffices to characterise the corresponding left-headed and right-headed products if W is interpreted as the set of linguistic resources and the accessibility relations are viewed as the counterparts of composition operations.

Again, these calculi of embedding and translations enjoy completeness, soundness and cut-elimination (Kurtonina and Moortgat, 1995).

2.3.5 Polymorphic types

Sometimes a lexical type may have different functions and still play similar roles. For instance, conjunctions can be used to coordinate nouns, sentences, and even non-standard constituents as noted above. The main idea behind polymorphism is to capture such generalisations in grammar specification. For the coordination case, the simplest solution appears to be to introduce quantified variables into the logic. The word and, for example, could receive a type such as $(X\backslash X)/X$. However, special worries arise when quantifiers are allowed into a logic. One of these concerns completeness. Emms shows (Emms, 1994) that the polymorphic version of L interpreted under the ternary frame semantics described above is incomplete if we allow quantifiers to range over arbitrary subsets of W.

An alternative to simply introducing variables is given in (Morrill, 1994). Morrill defines *meet* and *join* operators which enable one to specify the argument types which a functor may require. For example, a polymorphic type for the lexical entry from in (e.17) would be the one given in example (e.18):

(e.17) a. a man from Edinburgh

b. a man walks from Edinburgh.

(e.18) $((N\backslash N) \wedge ((NP\backslash S)\backslash(NP\backslash S)))/NP$

Here the operator (.∧.) denotes a kind of substructural disjunction of types which can play the role of the resulting type, hence generalising over the fact that in both cases the functor takes an NP for complement. Morrill also defines a another operator, (.∨.), which is used

to generalise over types occurring in an argument position. We will discuss polymorphism in greater detail in chapter 6, where a tableau treatment of the phenomenon is introduced.

2.4 Curry-Howard isomorphism: the syntax–semantics interface

So far we have talked about the features of categorial grammar which concern syntactic description. We have seen that one of its attractive aspects is that it gives us on the one hand enough expressivity to specify many properties of natural language in a principled way, and mathematical rigour on the other hand. Furthermore, we have seen that structural transformations arising purely from logical, proof-theoretic motivations find natural counterparts in the syntactic structure of natural-language constructs. The syntax-semantics interface of Lambek logics is concerned with extending this correspondence to the semantic level.

Since Montague's (Montague, 1974; D.R. Dowty and Peters, 1981) proposal that natural languages should receive the semantic treatment of formal languages, a great deal of attention has been devoted to specifying translation mechanisms between syntax and a logical (intensional) language whose semantics can be defined model-theoretically. In a nutshell, Montague's approach consisted of defining an extension of the simply-typed lambda calculus encompassing higher-order types which he called *Intensional Logic* (IL) and a function from syntactic types to the semantic types of IL. Given a type a, a set of possible denotations of type a, D_a is defined – e.g. one-place predicates are defined as functions from individuals to truth-values, so assuming individuals to have denotation D_e, the set of objects possibly denoted by one-place predicates is represented by $D_{<e,t>}$. The syntactic apparatus borrowed from the lambda calculus also includes the following devices: (a) function abstraction: $\lambda x\psi$ stands for the function which applied to value v results in the object denoted by ψ when x has value v, provided that $\psi \in \mathcal{T}_1 \rightarrow \mathcal{T}_2$ and x ranges over objects of $\mathcal{T}_2$; (b) function application, where $(\phi\psi)$ denotes the result of applying ϕ to argument ψ; and (c) substitution: $\phi[x \leftarrow \psi]$ standing for term ϕ with all occurrences of variable x replaced by term ψ. The system is

defined in such a way that the following properties hold:

$$\lambda x\psi = \lambda y\psi[x \leftarrow y] \tag{2.15}$$

where no free x occurs in the scope of y in ψ

$$(\lambda x\psi\ \phi) = \psi[x \leftarrow \phi] \tag{2.16}$$

where no free x occurs in the scope of a variable of ϕ.

$$\lambda x(\psi\ x) = \psi \tag{2.17}$$

where no free x occurs ψ

The original Montague system employs a variant of AB to implement the translation procedure. However, an observation well known in the area of functional programming known as the Curry-Howard isomorphism opens up the possibility of using Lambek calculi profitably for the same purpose. The Curry-Howard correspondence states a one-to-one correspondence between proofs in a natural deduction system for intuitionist logic and lambda terms. A Gentzen system that realises the properties of the λ-calculus, (2.15)–(2.17), will have the form of a sequent calculus for the implicational fragment of intuitionist logic, though the one-to-one correspondence is lost due to inherent non-determinism in its rules[7]. Since the Lambek calculi described above are implicational fragments of substructural logics to which the correspondence can be extended their syntax can be nicely coupled to a Montague-style semantic interpretation. The sequents in (2.18) show how this can be done in L (van Benthem, 1986; Moortgat, 1988; Hendriks, 1993).

$$\begin{array}{cc}
(\mathrm{R}/)\dfrac{\Delta, A : x \vdash B : \beta}{\Delta \vdash B/A : \lambda x\beta} & (\mathrm{L}/)\dfrac{\Gamma \vdash C : \gamma \quad \Psi, A : (\phi\,\gamma), \Phi \vdash B : \beta}{\Psi, A/C : \phi, \Gamma, \Phi \vdash B : \beta} \\[2ex]
(\mathrm{R}\backslash)\dfrac{A : x, \Delta \vdash B : \beta}{\Delta \vdash A\backslash B : \lambda x\beta} & (\mathrm{L}\backslash)\dfrac{\Gamma \vdash C : \gamma \quad \Psi, A : (\phi\,\gamma), \Phi \vdash B : \beta}{\Psi, \Gamma, C\backslash A : \phi, \Phi \vdash B : \beta} \\[2ex]
(\mathrm{cut})\dfrac{\Gamma \vdash A : \alpha \quad \Psi, A : \alpha, \Phi \vdash C : \gamma}{\Psi, \Gamma, \Phi \vdash C : \gamma} & (\mathrm{Id})\dfrac{}{A : \alpha \vdash A : \alpha}
\end{array} \tag{2.18}$$

[7] As we shall discuss later, in CG parsing based on Gentzen sequents, the loss of the correspondence gives rise to the problem of *spurious ambiguity*.

Once the logical relationships between λ-terms and the deductive apparatus which controls the syntactic types has been established then the semantics of the (natural) language to be parsed can be totally specified in the lexical level. In addition to being based on sound logical principles, this technique agrees with the CG tradition of lexicalism, corresponding to an implementation of type-driven translation, as defined in (Klein and Sag, 1985).

2.5 Overview of results on recognising power

Recognising power has been an issue intensely studied since the early papers on CG. Already in (Bar-Hillel, Gayfman, and Shamir, 1960) it was shown that the applicative calculus AB is in fact equivalent to a context-free language. Chomsky (Chomsky and Miller, 1963) conjectured that the associative Lambek calculus L is also equivalent to context-free grammars. The strong equivalence between the non-associative calculus NL and the latter was shown by Buszkowski (Buszkowski, 1988). In (Cohen, 1967) it is proved that the generative capacity of each AB-grammar is equivalent to that of some L-grammar. Pentus (Pentus, 1993) completed Cohen's proof showing Chomsky conjecture to be correct and therefore that L and AB are equivalent in weak generative capacity. In (Buszkowski, 1996) a technique was developed which allows one to transform any L-grammar into an AB-grammar by expanding its original type assignment. It has also been shown (van Benthem, 1987; Buszkowski, 1988) that LP recognises all permutation closures of context-free languages (which includes some non-context-free ones). Although it has been conjectured (Buszkowski, 1996) that the converse also holds, a proof of this conjecture hasn't been presented so far. Finally, (Carpenter, 1995) demonstrates that multimodal CG is Turing-complete in weak generative power.

2.6 Computational issues

Due to the diversity of calculi and operations involved Categorial logics often require sophisticated parsing mechanisms. Fine-grained structural control asks for extra bookkeeping tasks whose implementation is not always trivial. Fortunately, there is a vast collection of techniques

which have been developed within the area of automated deduction which can be adapted to use in CG. In the next sections we introduce the issue of theorem proving and review the application of some of its methods to CG parsing. A more detailed treatment of the most relevant ones will be given in chapter 5. The emphasis here will be on presenting a general perspective on the problems encountered by previous approaches to automated deduction in the range of Lambek calculi described above, thus setting the scene for the introduction of our own approach in the following chapter.

2.6.1 Parsing as theorem-proving

In computational linguistics, grammar specification often resembles software design. Linguistic and psychological requirements are analysed and subsequently incorporated into a parser through which the adequacy of specification and assumptions is evaluated in practice. Under the paradigm of *parsing as deduction* (Shieber, Schabes, and Pereira, 1994), as the name suggests, grammatical sentences are identified with theorems of a logic, and therefore parsing corresponds essentially to theorem proving. The research programme of categorial grammar probably represents the most radical attempt to realise this paradigm.

We claim in (Luz, 1996b) that a system for categorial deduction should in general meet the following requirements:

- The framework should to be general enough to accommodate the basic substructural calculi and allow further extensions
- The user should be able to specify and experiment with different theories in a transparent way
- The display of the derivations should ideally reflect the linguistic structures being analysed in an intuitive way
- The system should be able to accommodate domain-dependent heuristics and mechanisms whereby linguistic knowledge may have a positive impact on efficiency

The last item, in particular, is an allusion to the fact that although theorem proving is computationally expensive in general, there is a vast amount of domain-dependent knowledge which have been largely studied in the linguistic literature but have either been neglected or only partially considered in most systems for automated CG deduction. These requirements will be further elaborated in chapters 4 and 5.

2.6.2 Tableau, resolution and other methods

Many proof procedures, originally meant for classical and/or intuitionist logic have been proposed: natural deduction, Gentzen's sequents, analytic (Smullyan style) tableaux, etc. Among these, methods which conform to the *sub-formula principle*[8] are particularly interesting, as far as automation is concerned.[9]. Most of them, along with proof methods developed specifically for resource logics, such as Girard's proof nets (a variant of Bibel's connection method), can be used for categorial logic.

Early implementations of CG parsing relied on cut-free Gentzen sequents implemented via backward chaining mechanisms (Moortgat, 1988). This approach suffers from several problems. Apart from the fact that it lacks generality, since implementing more powerful calculi would involve modifying the code in order to accommodate new structural rules, the theorem proving strategy presents various sources of inefficiency. The main ones are: the generate-and-test strategy employed to cope with associativity, the non-determinism in the branching rules (L/) and (L\), and the ambiguity induced by the fact that different sequences of rules might produce essentially the same proof. To reduce the impact of the latter over efficiency has been the goal of *proof normalisation* (König, 1989; Hepple, 1990). However, even in normal-form proofs a certain degree of non-determinism still remains and the search space is usually of considerable size, though it could be mitigated (in contraction and expansion-free calculi at least) by testing branches for *count invariance* (van Benthem, 1988). As we tackle stronger logics and incorporate structural modalities such problems tend to get much harder.

[8] The sub-formula principle says that all formulae to be introduced in a derivation should be sub-formulae of formulae already in the derivation. The tableau systems of chapter 3 will obey this principle, as do the cut-free sequent calculus.

[9] See (Fitting, 1990) for a survey.

An improved attempt to deal uniformly with multiple calculi is presented in (Moortgat, 1992). In that paper, the theorem prover employed is based on proof nets, and the characterisation of different calculi is taken care of by labelling the formulae as in Gabbay's Labelled Deductive Systems (Gabbay, 1994). For substructural calculi stronger than L, much of the complexity (perhaps too much) is shifted to the label unification procedures. However, as pointed out in (Morrill, 1995a), while proof nets alone are unsuitable for dealing with built-in modalities and non-associativity, the kind of associative unification required by the labelling regime has expensive worst cases. A strategy for improving such procedures by compiling labels into higher-order linear logic programming clauses is presented in (Morrill, 1995b) for NL and L. Furthermore, a comprehensive solution to the problem of binding label unification which arises as we move from sequents to labelled proof nets, has not been presented yet. Moreover, as discussed in (Leslie, 1990), if we consider that the system is to be used as a parser, as a tool for linguistic study, the proof-net style of derivation does not seem to provide a very intuitive display of the proofs.

As far as we are aware, standard tableau systems have not yet being used in categorial parsing[10]. A reason for this may be the fact that Smullyan style tableau systems have been shown to be inherently inefficient (D'Agostino and Mondadori, 1994) as the method fails even to simulate truth-tables in polynomial time[11]. This is because of the fact that many of the Smullyan tableau expansion rules cause the proof tree to branch, thus increasing the search space (enormously in special cases such as pigeon-hole formulae (Cook and Reckhow, 1979)). In addition, keeping track of the structure of the derivations represents an extra source of complexity, which in most categorial parsers (Moortgat, 1992; Morrill, 1995b) is reflected in the bottleneck of unification algorithms employed for dealing with substructural implication. Our approach, as we shall see later, employs a variant of the tableau method which minimises these problems.

[10] Leslie (Leslie, 1990) presents and compares some categorial versions of these procedures for the standard Lambek calculus L, taking into account complexity and proof presentation issues. Although tableau systems are not discussed in (Leslie, 1990), a close relative, the cut-free sequent calculus is presented as being the one which represents the best compromise between implementability and display of the proof.

[11] Notice that the same result applies to cut-free Gentzen systems, which however have been used extensively.

2.7 Summary and further references

This chapter has presented a summary and overview of the main issues in the categorial grammar programme which stems from the calculus of syntactic types developed by Lambek. We have tried to balance formal presentation with the linguistic analysis which motivated the introduction of each formal device. In addition, the main theoretical results have been surveyed and the computational problems related to parsing CGs introduced. We shall elaborate further on the issues concerned with the latter in the next chapters.

From a syntactic point of view, calculi L–LPCE and the modal apparatus described above can be considered augmentations of the original applicative calculus. However, there are alternatives to the approach on which we have focused in this chapter. The most popular ones are the enrichment of AB with unification (Zeevat, Klein, and Calder, 1987) or combinators (Steedman, 1987). These approaches however fall outside the scope of the kind of logic and computational framework to be proposed in chapter 3, and therefore have not been discussed here.

From a (logical) semantic perspective we have not discussed (Barwise, Gabbay, and Hartonas, 1994), who treat lexical items as "information channels" within the framework of channel theory. This work, although theoretically interesting since it relates the Lambek calculus and labelled deduction, can hardly be considered "mainstream" categorial grammar. We have therefore decided to postpone its discussion to chapter 6.

Chapter 3

Automated Substructural Deduction for CG

In this chapter we describe a theorem proving framework for categorial deduction along the lines of the systems discussed in chapter 2. We start by setting up the basic ideas informally, discussing the general approach to proof-search to be adopted: analytic deduction based on tableaux. We then move on to a formal presentation of the theorem proving strategy, describing the main tableau expansion algorithms as well as the algebraic apparatus used to characterise different calculi. We will first give an overview of the system and its different modules followed by a exposition focused on the *syntactic module* (see figure 3.1). A detailed description of the *labelling module*, heuristics, and comparisons with other methods will be the subject of the following chapters.

Completeness and soundness results with respect to the Gentzen sequent presentation of the calculi, adapted from the ones given in (D'Agostino and Gabbay, 1994), are presented and discussed along with computational issues of termination and label introduction. Finally, we point out the problems of non-termination exhibited by the algorithm given in (D'Agostino and Gabbay, 1994) and show how to extend their semi-decision procedure into a full decision procedure for the range of categorial calculi defined in the previous chapter.

3.1 Overview of the parsing architecture

As we have seen in chapter 2, there is a large variety of categorial calculi which have been used in the description of linguistic phenomena. In order for these calculi to make the transition from theoretical tools into applied parsing mechanisms automated deduction techniques are required — that is, of course, if one accepts the Shieber et al paradigm of *parsing as deduction*. In what follows we describe a system, which we will call LLKE[1], designed for performing automated deduction in the range of categorial calculi within the hierarchy of implicational fragments of substructural logics (Došen, 1992).

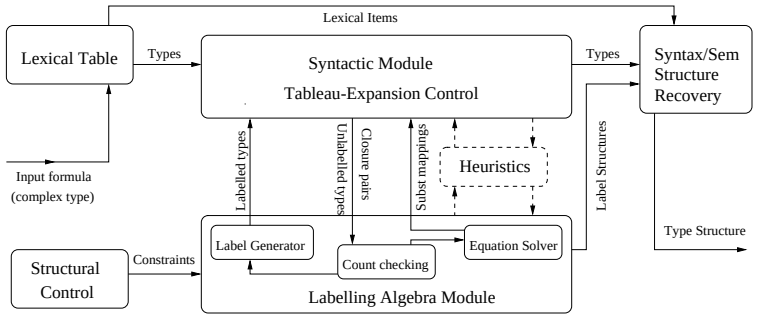


Figure 3.1: LLKE Architecture

The general architecture of LLKE is shown in figure 3.1. The main processing tasks are divided, in consonance with the philosophy of labelled systems, between two main modules:

- the *syntactic module* which controls the operations performed on the proof tree according to the syntactic (in logical terms) structure of CG types. These operations, essentially defined via tableau expansion rules, will be described in detail in the remaining sections of this chapter.
- the *labelling algebra* or *semantic module*, which controls structural features of the proof and ultimately determines the target calculus. By varying the settings of this module one changes the notion of grammaticality against which a given input string will be evaluated. The algebraic module will be partially described in this chapter and discussed in greater detail in chapter 4.

[1] "KE" after Dagostino's (D'Agostino and Mondadori, 1994) "surgical cut" system for propositional logic on which the general proof-search mechanism is based, plus an "L" for "labelled", after Gabbay's labelled deductive systems (Gabbay, 1994) and another "L" to go with CG's vast collection of "Ls" after (Lambek, 1958).

In addition to these, we still have a component which manages a hash table associating (natural language) lexical entries to CG types and a module to control the features of the algebraic module. The output is delivered by a module which, given a closed tree (i.e. a graph encoding a successful proof search in the target calculus), extracts the relevant linguistic information and returns the LLKE equivalent of a parse tree — ignoring the subtrees of the proof tree which correspond to lemma generation, failed searches etc. The central features of this output module are detailed in chapters 4 and 5.

3.2 Labelled deduction based on tableaux

LLKE is based on a theorem proving technique originally developed for (classical) first-order logic: semantic tableaux (Smullyan, 1968; Fitting, 1990). Tableaux are built according to recursive rules which operate on sets of formulae. These rules specify conditions under which to "expand" the set with new formulae or "mark" the set as *closed*. A closed set of formulae does not undergo further expansion.

Tableaux, like model-elimination and resolution provers, are *refutation proof systems*. This means that, in classical predicate logic, expanding a tableau corresponds to building a counter-model for the negation of a formula one wants to prove — i.e. a formalisation of mathematical reasoning by *reductio ad absurdum*: one tries to show that if a presumably false claim is assumed to be true the assumption leads to a contradiction. Various data structures may be used to encode counter-models. The most popular are trees. A counter model for a formula is then assumed to be a tree structure whose branches are all closed.

The condition for a branch to be considered closed in a standard tableau is that both a formula and its negation occur on it. Since the syntax of the categorial calculi defined in chapter 2 does not allow for negation, we have to appeal to some extrinsic mechanism in order to be able to express contradiction. Automated deduction systems which like LLKE appeal to bookkeeping devices, i.e. systems which use symbols outside the logical language of their calculi, are called *non-uniform theorem provers* (Fitting, 1990). In the non-uniform systems described in (Fitting, 1990), all formulae occurring in a derivation are preceded by *signs*: "T" to sign formulae evaluated as "true" and "F" for formulae evaluated as "false".

To illustrate the method, suppose one wishes to prove $A \Rightarrow A$ in classical logic. One starts by assuming that the formula is "false", prefixing it by F, and tries to find a refutation for $F : A \Rightarrow A$. For this to be the case in classical logic both $T : A$ (the antecedent) and $F : A$ (the consequent) must be the case. The pair of formula $\langle T : A, F : A\rangle$ in the (single) branch of our proof tree $\mathcal{T} = \{F : A \Rightarrow A, T : A, F : A\}$ yields a contradiction, hence $\mathcal{T}$ may be regarded as a counter-model for $F : A \Rightarrow A$.

In classical logic one can interpret T and F respectively as assertion and denial of a proposition. Therefore one is able to translate the external symbol F into the logical syntax as negation, thus obtaining a *uniform notation* and eliminating the need for signed formulae. In LLKE there is no alternative to the use of signed formulae as proof theoretic devices. However, since the notion of "truth" in classical logic is obviously not the same as logical truth in substructural systems[2] these signs should receive different interpretation. To give them a (necessarily imprecise) intuitive interpretation we could say that "T" and "F" will be used in LLKE to indicate whether or not a certain string is *available for combination* in the tableau to produce a new string.

If we had restricted the system to dealing with signed formulae only, we would have ended up with a proof method for an implicational fragment of standard propositional logic enriched with backwards implication and conjunction. However, we have seen that the Lambek calculus does not exhibit any of the structural properties of standard logic, and that different calculi

[2]See discussion on *logics of information flow* (Barwise, Gabbay, and Hartonas, 1995) in chapter 6 to see how the notion of truth plays a much less important role in logics aimed at modelling dynamic phenomena such as the ones presented in chapter 2.

may be obtained allowing different structural rules. Therefore, we also need a mechanism for keeping track of the structure of our proofs. This mechanism is provided by labelling each formula in the derivation with *information tokens*. This labelling technique, already used in different CG systems (see chapter 2), has been motivated proof-theoretically in Gabbay's LDSs (Gabbay, 1994), and semantically in Barwise's Channel Theory (Barwise, Gabbay, and Hartonas, 1995). Before we carry on with our description of LLKE, let's formalise a few concepts related to the data structures employed by the system which have been loosely used above.

3.2.1 Tree structures

LLKE proofs will be represented as *ordered dyadic trees*. An unordered tree $\mathcal{T}$ is a defined by (i) a set of elements which we will refer to as *nodes* (we also refer to nodes as being elements of $\mathcal{T}$) (ii) a function *level* which assign to each node $x \in \mathcal{T}$ a positive integer $level(x)$ and (iii) relation xPy to be read as "x is a *predecessor* of y" or "y is a *successor* of x" obeying the following conditions: there is a unique node i (called *the root* or *origin* of $\mathcal{T}$) such that $level(i) = 1$, all other nodes have a unique predecessor, and for any nodes x, y if xPy then $level(y) = level(x) + 1$. A tree will be represented graphically with the root at the top, so we will sometimes refer to this node as *the topmost node* of the tree. A node l is called a *leaf* or an *end point* if it has no successors. A node with two successors is called a *branching node*. A dyadic tree is a tree structure in which each node has at most 2 successors. A *path* is a denumerable sequence of nodes beginning at the root in which each node is the predecessor of the next, except naturally in the case of the last node of a finite path. If a node x is the last node of a path on which a node y occurs we say that y *dominates* x. In case $x \neq y$ and y dominates x in $\mathcal{T}$ we say that y occurs *above* x in $\mathcal{T}$. A path whose last node is a leaf is called *branch*.

We speak of a *subtree* or $\mathcal{T}'$ referring to a subset of $\mathcal{T}$ obeying conditions (i), (ii) and (iii) above. An ordered tree can be generated by equipping an unordered tree with a function which orders the successors of each node in it. In displaying a tree this ordering will be reflected by the position of the node on the page: the *first* successor will be displayed as the

leftmost node etc. We will sometimes refer to *tree expansion*: by this we mean the adjoining of a node to a leaf of a tree — i.e. y expands tree $\mathcal{T}$ into $\mathcal{T}' = \mathcal{T} \cup \{y\}$ if there is a node x s.t. x is a leaf of $\mathcal{T}$, the predecessor relation is extended so to $R' = R \cup \{\langle x, y\rangle\}$, and $level(y) = level(x) + 1$ in $\mathcal{T}'$.

Trees in which all nodes have a finite number of successors are said to be *finitely generated*. A *finite tree* is a tree which have an finite number of nodes. A finitely generated tree can have an infinite number of points. An important result on trees from a theorem proving perspective is Königs lemma which says that every finitely generated tree with an infinite number of points must have at least one infinite path — see (Fitting, 1990; Smullyan, 1968) for proofs and discussion. We now overview the differences between two non-labelled tableau systems (i.e. systems for standard propositional logic): Smullyan-style tableaux and tableaux which incorporate a mild form of cut (D'Agostino and Mondadori, 1994). After this we return to our remarks on labels and proof structure.

3.2.2 Non-labelled propositional systems: tableaux Vs. "surgical" cut

As we have seen, in categorial grammar one ascribes lexical entries (words) to *types* which describe their function. Types can be primitive – e.g. N (noun) – or compound from primitives through the operators: "/", "\" and "•". Parsing thus (roughly) corresponds to determining whether a set of types joined by "•" yields another. The "residuation" operators "\" and "/" can be seen as forms of implication, while "•" can be regarded as a form of conjunction. Therefore, both tableau and Gentzen rules for these categorial connectives resemble the ones for classical connectives, except for the structural devices. If we ignore structural properties for the moment, we can say that a standard (Smullyan-style) tableau system has two kinds of rules:

- an "α rule" which expands formulae of the form $F : A/B$ $(F : B\backslash A)$ and $T : A \bullet B$ by adjoining $\{T : B, F : A\}$ and $\{T : B, T : A\}$ respectively to the branch where those formulae occur and
- a "β" rule which expands the dual of those formulae — $T : A/B$ etc — by adjoining two subtrees to the branch where these β-formulae occur. For instance, a β expansion for a list $\langle T : NP/S, T : S\rangle$ would generate two branches: $\langle T : NP/S, T : S, T : S\rangle$ and $\langle T : NP/S, F : NP, T : S\rangle$.

The assumption that Smullyan style tableau systems are adequate for automated deduction in propositional logic has been challenged recently (D'Agostino, 1992; D'Agostino and Mondadori, 1994) on the basis that tableaux, as well as cut-free Gentzen systems, exhibit three anomalies: 1) they fail to reflect the principle of bivalence (whose counterpart in Gentzen systems is the cut rule), 2) they are computationally expensive (not even being able to p-simulate truth-tables!), 3) they don't allow for "nesting" of subproofs (lemmas), thus leaving little room for heuristics which could mitigate computational complexity. In order to address these problems, (D'Agostino and Mondadori, 1994) proposes a system where the tableau β (tree branching) rules are replaced by linear rules plus a single branching one: a "surgical" cut. In this system, β-formulae do not immediately branch the derivation tree. Instead they "look up" the branch where they occur for other formulae with which they might combine in order to yield a third formula which then gets adjoined to the branch. We will call such rules "σ rules". For instance, a rule for "/" says: *for any A, B, if both $T : A/B$ and $T : B$ occur in a branch, then expand this branch with $\{T : A\}$*. A σ-expansion for $\langle T : NP/S, T : S\rangle$ would generate a single branch: $\langle T : NP/S, T : S, T : NP\rangle$.

However, rules α and σ alone do not suffice to generate Hintikka sets (Smullyan, 1968) from arbitrary sets of formulae, and therefore a system restricted to these rules would be incomplete. An extra rule needed, for instance, to deal with those β-formulae to which no σ-rule can be successfully applied. In order to complete the system (D'Agostino and Mondadori, 1994) proposes adding a form of cut rule to it. With cut, completely expanded branches are achieved by branching the tree with pairs of the form *T:A* and *F:A* – clearly an implementation of the principle of bivalence. Let's call this rule "θ rule". If we restrict A to subformulae occurring in the derivation, we guarantee termination. Under this restriction an application of a σ rule is sometimes called "surgical cut".

Systems which employ surgical cut can be shown to outperform Smullyan-style tableaux for propositional calculi in general. They are able to simulate standard tableau proofs (and nat-

urally truth tables) in polynomial time while the converse does not hold (Cook and Reckhow, 1979; Haken, 1985; D'Agostino and Mondadori, 1994). This is one of the reasons why we have chosen to build LLKE as a surgical-cut system. There are other reasons derived from specific properties of the labelling regime employed in categorial logics. The latter will be discussed in detail in chapter 4.

3.2.3 The labelling algebra

Labels will act not only as mechanisms for encoding the structure of the proof, from a proof-theoretic perspective, but will also serve as means to propagate *semantic*[3] information through the derivation. A label can be seen as an information token *supporting* the information conveyed by the signalled formula that it labels. Tokens may convey different degrees of *informativeness*. Therefore we will assume that they are ordered by an anti-symmetric, reflexive and transitive relation "$\sqsubseteq$" so that the expression $x \sqsubseteq y$ will assert that "y" is at least as informative as "x". In other words: "*y verifies* (or supports the occurrence at a certain position of) at least as many types as x". We will also assume that this semantic relation, "verifies", whose meaning will be precisely defined below, is closed under deductibility (i.e if a verifies A and $A \vdash B$ then a verifies B).

It is natural to suppose that, as well as syntactic types, information tokens can be combined. We have seen that a type such as S/NP can combine with a type NP to produce a third type S. If we assume that there are semantic tokens x and y verifying respectively S/NP and NP, how would we represent the token that verifies S? In order to answer this question we define a *token composition* operation, $\circ$. One can think of information tokens as being sets (multisets, lists) of types closed under certain logical operations and of the types they support as elements (members) of those sets (multisets, lists). Following our natural language intuitions, we assume that, *a priori*, the number and order (position) of syntactic types appearing on a string matter with regard to its grammaticality. Therefore we interpret information tokens as *lists* of types. Accordingly, a minimal information token verifying S in the example above

[3] In the context of tableau and labelled deduction in general, we will sometimes use the term "semantic" to refer to structural properties of proofs as opposed to *semantics of natural languages*, for instance.

will be given by appending list y to list x, which we notate as $x \circ y$.

As we will see below, the constraints imposed on $\circ$ will ultimately determine which inferences will be licensed for a particular calculus. For instance:

- if we decide to relax our constraints so that the order in which the types occur is made irrelevant, then we may allow permutation on the operands, which corresponds to property $x \circ y \sqsubseteq y \circ x$ with respect to the ordering relation
- or we may decide that in certain cases the number of types occurring on a string is not relevant and allow contraction as a structural property of the calculus. In this case strings such as the one obtained by concatenating $\{S/NP, NP, NP\}$ will also yield an S, in spite of the duplication of an NP. In terms of information ordering we say that $y \circ y \sqsubseteq y$, etc.

Let's formalise the notions introduced above through definition 3.2. We will use an algebraic structure, called *Information frame* (Gabbay, 1994) which encompasses the necessary structural sensitivity.

Definition 3.2 *An Information Frame is a structure $\mathcal{L} = \langle \mathcal{P}, \circ, \mathtt{1}, \sqsubseteq \rangle$, where*

(i) *$\mathcal{P}$ is a non-empty set of information tokens;*

(ii) *$\circ$ is an order-preserving, binary operation on $\mathcal{P}$ which satisfies continuity, i.e., for every directed family $\{z_i\}$, $\bigsqcup\{z_i \circ x\} = \bigsqcup\{z_i\} \circ x$ and $\bigsqcup\{x \circ z_i\} = x \circ \bigsqcup\{z_i\}$; and*

(iii) *$\mathtt{1}$ is an identity element in $\mathcal{P}$.*

Combinations of types in derivations are accounted for in the labelling algebra by the composition operator. Now, we need to define an algebraic counterpart for the decomposition of types joined by the multiplication operator "$\bullet$". When a formula such as $S/NP \bullet NP$ is verified by a token x this is because its components were available for combination, and consequently were verified by other tokens. Now, suppose S/NP was verified by a token, say a. What would be the appropriate token for NP, such that S/NP combined with NP would be

verified by token x? It certainly could not be more informative token than x. Moreover, if the expression $S/NP \bullet NP$ were to stand for the composition of the (informational) meanings of its components, then the label for NP would have to verify, when combined with a, at most as much information as token x. In order to express this, we define the label for NP as being *the greatest information token y such that x is at least as informative as a combined with y.* This token will be represented by $x/\!\!/a$. In general, $x/\!\!/y \stackrel{\text{def}}{=} \bigsqcup\{z \mid y \circ z \sqsubseteq x\}$. An analogous operation, $\backslash\!\backslash$, is defined to cope with cases in which it is necessary to find the appropriate label for the first operand by reversing the order of the tokens. Both operators are forms of algebraic division, and we use the double lines to differentiate them from syntactic CG division. Below are some properties of $/\!\!/$ (with analogous properties holding for $\backslash\!\backslash$):

$$y \circ (x/\!\!/y) \sqsubseteq x \qquad (3.1) \qquad\qquad 1 \sqsubseteq x/\!\!/x \qquad (3.2)$$

$$(x/\!\!/y) \circ z \sqsubseteq (x \circ z)/\!\!/y \qquad (3.3) \qquad\qquad (x/\!\!/y)/\!\!/z \sqsubseteq x/\!\!/(y \circ z) \qquad (3.4)$$

We now define a language of algebraic expressions which mimics our type language of definition 2.1. Given a set of tokens $\mathcal{P} = \{a, b, c, ...\}$ and $\mathcal{V} = \{x, y, z, ...\}$, a set of *label variables*, we define our language of label expressions, $\mathcal{L}^*$, as the closure of $\mathcal{P} \cup \mathcal{V}$ under label operators $\circ, /\!\!/, \backslash\!\backslash$. It is sometimes convenient to distinguish between $\mathcal{P}^*$, the set of variable-free label expressions, and $\mathcal{V}^* = \mathcal{L}^* - \mathcal{P}^*$, the set of label expressions containing *at least one* label variable.

3.3 Derivation trees

Both the (syntactic) type language defined in the previous chapter and the algebraic structure defined above will be employed in LLKE derivations. In fact, each node of a derivation tree will contain a sign, a type expression and a label expression. We call the expressions on each node *Signed Labelled Formulae* (*SLF* for short). Definition 3.3 provides some extra tools for dealing with the components of a derivation.

Definition 3.3 *The set of Signed Labelled Formulae (SLF) is the set of expressions of the form '$S : Type : L$', where $S \in \{T, F\}$, $Type \in \mathcal{C}$ and $L \in \mathcal{L}$. We also define the functions $s : SLF \to \{T, F\}$, $t : SLF \to \mathcal{C}$, and $l : SLF \to \mathcal{L}^*$ to denote the components of a SLF. A SLF X where $s(X) = T$ is sometimes called a* T-formula*; likewise, if $s(X) = F$, then X is called a* F-formula

A derivation, or proof will be a tree structure built according to certain syntactic rules. These rules will be called *expansion* rules, since their application will invariably expand the tree structure. Again, as in the non-labelled version, there are three sorts of expansion rules: those which expand the tree by generating two formulae from a single one occurring previously in the derivation, those which expand the tree by combining two formulae into a third one which is then added to the tree, and the branching rule. The first kind of rule is a labelled version of what is called α-rule in Smullyan-style tableaux (Smullyan, 1968). These rules will be called α-rules here as well. The second and third kinds have no equivalent in standard tableau systems. We will abuse the naming conventions of section 3.2.2 and refer to the second kind as σ-rules, and to the branching rule as θ rule[4].

Table 3.1 summarises the expansion rules to be employed by the system. Notice that a, b are informations tokens, n is a *new* label (i.e. a label not occurring previously in the derivation) and x is a label variable. A deduction bar specifies that if the formula(e) appearing above it occurs (we call these *premises*) in the tree, then the formula(e) below it (we call these *conclusions*) should be added to the tableau. The rules are easily interpreted according to the intuitions ascribed above to signs, formulae and information tokens. A rule like $\alpha_{(i)}$, for instance, says that if $A\backslash B$ is not available for combination and x verifies such information, then this is because there is an A available at some token a but the combination of a and x (notice that the order is relevant) fails to make B available in the proof tree for further combinations. Rule $\sigma_{(i)}$ says that if both types $A\backslash B$ and A are available in a proof then type B can be made available provided that the token labelling it contains all information contained in the tokens which labelled the former types. Rule θ implements a version of the classical principle of bivalence: a token x either supports a type or its denial.

[4] Although this rule is a tableau-branching one, we prefer to call it θ rule, instead of β, in order to avoid confusion with β rules in Smullyan-style tableaux.

α**-rules**	(i)	(ii)	(iii)
(α_1)	$\underline{F : A \backslash B : a}$	$\underline{F : A / B : a}$	$\underline{T : A \bullet B : a}$
(α_2)	$T : A : n$	$T : B : n$	$T : A : n$
(α_3)	$F : B : n \circ a$	$F : A : a \circ n$	$T : B : a \mathbin{/\!\!/} n$
σ**-rules**	(i)	(ii)	(iii)
(σ_1)	$T : A \backslash B : a$	$T : A \backslash B : a$	$T : A / B : a$
$\underline{(\sigma_2)}$	$\underline{T : A : b}$	$\underline{F : B : b \circ a}$	$\underline{F : A : a \circ b}$
(σ_3)	$T : B : b \circ a$	$F : A : b$	$F : B : b$
	(iv)	(v)	(vi)
(σ_1)	$T : A / B : a$	$F : A \bullet B : a$	$F : A \bullet B : a$
$\underline{(\sigma_2)}$	$\underline{T : B : b}$	$\underline{T : A : b}$	$\underline{T : B : b}$
(σ_3)	$T : A : a \circ b$	$F : B : a \mathbin{/\!\!/} b$	$F : A : a \mathbin{\backslash\!\!\backslash} b$
θ**-rule**			
$\overline{(\theta_1)\ \ F : A : x \mid (\theta_2)\ \ T : A : x}$			

Table 3.1: Tableau expansion rules

We assume that the SLF at the topmost node is labelled by the identity element of $\mathcal{L}$. Furthermore, we generalise rule $\alpha_{(i)}$ to cover the first expansion, viz., the decomposition of $T : X \vdash Y : \mathtt{1}$ into $T : X : n$ and $F : X : n \circ \mathtt{1} = n$. We call the subformulae introduced by the first round of exhaustive applications of α rules — see algorithm 3.1 below — to a set of formulae *initial (sub)formulae* — or *initial sub(types)*. Likewise, we extend the scope of $\alpha_{(iii)}$ to cover the concatenation operators (',') which appear on the antecedent of an entailment relation. Thus we define an *initial tableau* for $A_1, ..., A_k \vdash A_0$ as a tree, $\mathcal{T}$, with the following structure:

$$\begin{array}{c} F : A_0 : l_0 \\ T : A_1 : l_1 \\ \vdots \\ T : A_{k-1} : l_{k-1} \\ T : A_k : (...(l_0 \mathbin{/\!\!/} l_1) \mathbin{/\!\!/} ...) \mathbin{/\!\!/} l_{k-1} \end{array} \tag{3.5}$$

Given the expansion rules, the definition of the main data structure to be manipulated by the parsing algorithm is straightforward: a derivation tree, $\mathcal{T}$, is simply a dyadic tree built from a given set of formulae by applying the rules in a certain order. The algorithm's termination

depends on the notions of completion along with (branch and tree) closure. It can be readily seen on table 3.1 that for a finite set of formulae, the number of times α rules can be applied increasing the number of SLF-s (nodes) in $\mathcal{T}$ is finite. Unbounded application of σ and θ, however, might expand the tree indefinitely. In order to assure termination some restrictions must be placed. We shall discuss them below, after we have defined the unrestricted procedures for α and σ expansion of the tableau.

The first step towards building a counter-model for the denial of a formula to be proved is the search for a tree containing *potential* contradictions, meaning pairs of types prefixed by T and F respectively. Whether or not a potentially closed tree is a counter-model for the formula will depend ultimately upon the constraints on the labelling algebra. The notion of *closure* defined below is employed by the tableau expansion algorithms.

Definition 3.4 *(Branch and Tree Closure) A branch (list of formulae) is* closed *with respect to the labelling algebra iff it contains SLFs of the form $T : X : x$ and $F : X : y$ — let's call a pair of such SLFs a* closure pair *— where $x \sqsubseteq y$, in which case the closure pair is said to be* successful*. Likewise, a (sub)tree is closed iff it contains only closed branches.*

Now we are ready to define an algorithm for linear expansion of the derivation tree. By *linearly expanded tableau* we mean a set of formulae to which only α and σ rules have been applied. For efficiency reasons non-branching rules will be exhaustively applied before we move on to employing θ-rules. Furthermore, since α rules do not involve search, it is more convenient to apply them first. We therefore split the linear expansion procedure into algorithm 3.1 and algorithm 3.2 as shown below in pseudocode[5]. Algorithm 3.2 describes the top level of expansion for a branch and may yield, under certain circumstances a complete derivation tree. The output of algorithm 3.1 applied to the initial SLF corresponds to a tableau's initial tree as defined in (3.5).

[5]The pseudocode symbols $\Leftarrow$ and $\triangleright$ denote value attribution and comments respectively. As usual, $\neg$ stands for negation whereas $\wedge$ stands for conjunction as read by, say, a Lisp interpreter. We also use functions whose interpretation should be reasonably straightforward: *closed*($\mathcal{T}$) returns boolean "true" if $\mathcal{T}$ is a closed branch according to definition 3.4, *(α)σ-type(f)* test whether f is of a certain kind according to table 3.1, *head($\mathcal{T}$)* returns the first element of branch $\mathcal{T}$, removing it from $\mathcal{T}$, or simply returns an "empty string" if $\mathcal{T}$ is empty.

Algorithm 3.1 *(Alpha Expansion) Given $\mathcal{T}$, a LLKE tableau structure, and rules α_i,...,α_{iii}, we define the procedure:*

α-expansion($\mathcal{T}$)
1 do formula $\Leftarrow$ head($\mathcal{T}$)
2 while non-empty(formula) $\wedge$ $\neg$closed($\mathcal{T}$)
3 do if α_1-type(formula)
4 then do $\alpha_2 \Leftarrow$ generate-new-label(α_2) $\triangleright$ assign new label to α_2
5 $\alpha_3 \Leftarrow$ combine-labels(α_1,α_2)
$\triangleright$ combine α_1 and α_2 labels into α_3
6 $\mathcal{T} \Leftarrow$ append($\mathcal{T}$,$\langle\alpha_2$,$\alpha_3\rangle$)
$\triangleright$ add subformulae to the original list
7 do formula $\Leftarrow$ head($\mathcal{T}$)
8 return $\mathcal{T}$

For notational convenience we define the set $|A_0, .., An|_\alpha$ as the result of applying algorithm 3.1 to a set of SLFs $\{A_0, .., An\}$, and $|A_0, .., An|_{\alpha\sigma}$ as the result of applying algorithm 3.2 to set $\{A_0, .., An\}$. The complete LLKE algorithm which uses the procedure below, algorithm 3.3, will be presented after we have discussed tableau closure from the information frame perspective.

Algorithm 3.2 *(Algorithm: Linear Expansion) Given $\mathcal{T}$, a LLKE tableau structure, we define linear exhaustive expansion as follows:*

linear-expansion($\mathcal{T}$)
1 do $\mathcal{T} \Leftarrow \alpha$-expansion($\mathcal{T}$)
2 formula $\Leftarrow$ head($\mathcal{T}$)
3 while non-empty(formula) $\wedge$ $\neg$closed($\mathcal{T}$)
4 do set$_{aux} \Leftarrow \emptyset$
5 do if σ_1-type(formula)
6 then do set$_{aux} \Leftarrow$ search($\mathcal{T}$,σ_2) $\triangleright$ set$_{aux}$ is a set of σ_2-slf's
7 else do if σ_2-type(formula)
8 then do set$_{aux} \Leftarrow$ search($\mathcal{T}$,σ_1) $\triangleright$ set$_{aux}$ is a set of σ_1-slf's
9 do if set$_{aux} \neq \emptyset$
10 then do σ_3-set $\Leftarrow$ combine-σ(formula, set$_{aux}$)
$\triangleright$ σ_3-set results of combining *formula* to each element of set$_{aux}$
11 σ_3-expansion $\Leftarrow$ α-expansion(σ_3-set)
12 $\mathcal{T} \Leftarrow$ append($\mathcal{T}$,σ_3-expansion)
13 do formula $\Leftarrow$ head($\mathcal{T}$)
14 return $\mathcal{T}$

We have seen above that the labels are means to propagate information about the formulae through the derivation tree. From a semantic viewpoint, the calculi addressed in this thesis are

obtained by varying the structure assigned to the set of formulae in the derivation[6]. Therefore, in order to verify whether a branch is closed for a calculus one has to verify whether the information frame satisfies the constraints which characterise the calculus. For instance, the standard Lambek calculus L does not permit any sort of structural manipulation of formulae apart from associativity; NL doesn't even allow that; LP allows formulae to change places in a string; LPE allows permutation and expansion; LPC allows permutation and contraction; etc. Definition 3.5 sets the algebraic counterparts of these properties[7].

Definition 3.5 *For all x, y, $z \in \mathcal{P}$, we call an information frame*

(i) associative *if*

$$x \circ (y \circ z) \sqsubseteq (x \circ y) \circ z \tag{3.6}$$

and

$$(x \circ y) \circ z \sqsubseteq x \circ (y \circ z) \tag{3.7}$$

(ii) commutative *if*

$$x \circ y \sqsubseteq y \circ x \tag{3.8}$$

(iii) contractive *if*

$$x \circ x \sqsubseteq x \tag{3.9}$$

and (iv) expansive *if*

$$x \sqsubseteq x \circ x \tag{3.10}$$

However, it is not immediately obvious that the differently constrained information frames of definition 3.5 suffice to account for the structural rules in (2.12). The most obvious approach would be to add structural tableau rules in the same way as structural sequent rules are added to Gentzen systems: $(P) = \frac{T:A:x \circ a \circ b \circ y}{T:A:x \circ b \circ a \circ y}$ for permutation, $(E) = \frac{T:A:x \circ a \circ y}{T:A:x \circ a \circ a \circ y}$ for expansion and $(C) = \frac{T:A:x \circ a \circ a \circ y}{T:A:x \circ a \circ y}$ for contraction (assuming L as the basis). Proposition 3.1 guarantees that no such structural tableau rules are needed [8].

Proposition 3.1 *All successful closure pairs obtained in trees generated by application of table 3.1 rules plus (some combination of) tableau structural rules (P), (E) and (C) can be obtained from trees generated exclusively from table 3.1 via (some appropriate combination of) structural constraints* (3.6)–(3.10).

[6] For instance, resource sensitive logics such as linear logic are frequently characterised in terms of multisets to keep track of the "use" of formulae throughout the derivation.

[7] In practice we will assume that expansive frames are also monotonic in order to preserve the label closure conditions of definition 3.4. Purely expansive frames would require closure to be evaluated with respect to the $\circ$-concatenation of the tokens of all T-formulae.

[8] This fact is responsible for much of the flexibility exhibited by the system, as we will discuss in chapter 4.

Proof. We say that a rule R *enables* (the application of) a rule T if one of the following holds: (i) the conclusion of R is a premise of T (in which case we say that R *immediately enables* T, or (ii) there is a rule S such that the conclusion of R is a premise of S and S enables T. There are two main cases to consider:

Case 1: The conclusion of a structural rule causes closure (i.e. the conclusion is a SLF in a closure pair). In this case it is obvious that the premise of the structural rule also causes closure under the appropriate structural constraint.

Case 2: The conclusion of a structural rule immediately enables an operational rule (table 3.1) which causes tableau closure (after a finite number of steps). Clearly, the only kind of operational rule that can be immediately enabled by the result of a structural rule is σ, particularly $\sigma_{(ii)}$ and $\sigma_{(iii)}$. The premises of the remaining rules are not sensitive to number and order of information tokens as can be readily verified by inspection on table 3.1. Furthermore, if the premise of an α rule were the conclusion of a structural rule, then the conclusions of the α rule would contain the same tokens as its premises (plus a newly introduced one) which obviously can be treated by structural constraints. Now, let's suppose the resulting SLF enables a σ rule. Again we have two subcases:

Subcase 1: The conclusion of the σ rule enabled by a structural rule, let's call this conclusion σ_3, forms a closure pair with some rule in $\mathcal{T}$. Then the formula in $\mathcal{T}$ with which σ_3 closes is of the form $\overline{\sigma}_3 = T : f(\sigma_3) : l(\sigma_3)$. But the result of applying σ_1 (the first premise of the σ-enabled rule) to $\overline{\sigma}_3$ is a formula $T : f(\sigma_2) : l(\sigma_2)$, which forms a closure pair with σ_2. Therefore the structural rule application is redundant in this case. See the path below for a graphical illustration of this fact with respect to rule (E):

$$
\begin{array}{lll}
 & \vdots & \\
1- & T : B : y & \mathit{Assump.} \\
 & \vdots & \\
2- & T : A/B : a \circ x \circ b & \mathit{Assump.} \\
3- & F : A : a \circ x \circ x \circ b \circ y & \mathit{Assump.} \\
\boxed{\begin{array}{lll} 4- & T : A/B : a \circ x \circ x \circ b & 2, (E) \\ 5- & F : B : y & 4, 3, \sigma_{(iii)} \\ & \times & \end{array}} & & \\
6- & T : A : a \circ x \circ b \circ y & 2, 1, \sigma_{(iv)} \\
 & \times &
\end{array}
\tag{3.11}
$$

The subtree enclosed in the box contains the redundant steps to be eliminated. We append the symbol $\times$ to denote branch closure. The vertical dots denote (possibly empty) subtrees and the numbers on the right identify the rule which yielded the SLF on the left. These conventions will be used throughout this thesis.

Subcase 2: Suppose σ_3 doesn't cause immediate closure as in subcase 1 but enables other rule applications. Let's assume σ_3 to match α_1 for some rule α. The tableau will then be extended with α_2 (where $s(\alpha_2) = T$ and $l(\alpha_2) = a$, a being a newly introduced token) and α_3 (where $s(\alpha_3) = F$ and $l(\alpha_3) = l[a]$, $l[a]$ being a token l in which a occurs. Clearly, α_2 cannot be in a closure pair with any SLF occurring above it in $\mathcal{T}$, since its label is new. Neither can α_3, since the token which labels α_2 occurs in $l(\alpha_3)$. Induction shows that no sequence of α rules can cause closure. Finally, if σ_3 matches σ_2 for some SLF in $\mathcal{T}$, then subcases 1 and 2 apply recursively. ■

3.4 Label checking and non-termination

Having established in proposition 3.1 that there is no need to manipulate label structure at the syntactic level (i.e. via expansion rules) we can safely circumscribe closure checking in the labelling algebra module. Checking for label closure will depend on the calculus being used, and consists basically of reducing information token expressions to a *normal form*, via properties (3.1)–(3.4), and then matching tokens and/or variables that might have been introduced by applications of the θ-rule according to the properties or combination of properties (*Definition 3.5*) that characterise the calculus considered. The precise label checking mechanism will be detailed in chapter 4. Example (3.1) below shows how linear expansion works in general:

Example 3.1 *Let's prove that the string* $NP \bullet (NP\backslash S)/NP$ *yields a type* S/NP *in the Lambek calculus. So, the expression we want to find a counter-model for is:*

$1 - F : NP \bullet (NP\backslash S)/NP \vdash_L S/NP$.

Therefore, the following has to be proved:

$2 - T : NP \bullet (NP\backslash S)/NP : m$ *and* $3 - F : S/NP : m$.

We proceed by breaking 2 and 3 down via $\alpha_{(iii)}$, obtaining:

$4-T:NP:a$, $5-T:(NP\backslash S)/NP:m/\!\!/a$, $6-T:NP:b$, *and* $7-F:S:(m\circ b)$.

Now we start applying σ-rules (annotated on the right-hand side of each line):

$8-\quad T:NP\backslash S:(m/\!\!/a)\circ b \quad 5,6\sigma_{(i)}$

$9-\quad T:S:a\circ((m/\!\!/a)\circ b) \quad 4,9\sigma_{(i)}$

We have derived a potential inconsistency between 7 and 9. Turning our attention to the information tokens, we verify closure for L *as follows:*

$$\begin{array}{lll} a\circ((m/\!\!/a)\circ b) & \sqsubseteq & (a\circ(m/\!\!/a))\circ b \quad \textit{by associativity} \\ & \sqsubseteq & m\circ b \quad \textit{by property (3.1)} \end{array}$$

■

It should be noticed that the algorithm in algorithm 3.2 performs "brute force" σ-expansion – i.e. each σ_1 formula is combined with all σ_2's in the tableau –. Most potential closures resulting from such combinations will be immediately ruled out by the label checker. We will discuss this point in chapter 4 along with other features of the algebraic module.

Allowing unrestricted bidirectional application of σ rules – steps 6 and 8 of *linear-expansion* – might lead to non-termination. Consider for example the infinite sequence of σ-applications:

$$\begin{array}{llll} 1- & T & A/B & :x \\ 2- & T & B/C & :y \\ 3- & T & C/A & :z \\ 4- & T & A & :w \\ 5- & T & C & :z\circ w, \quad 3,4,\sigma_{(i)} \\ 6- & T & B & :y\circ(z\circ w) \quad 2,5,\sigma_{(i)} \\ 7- & T & A & :x\circ(y\circ(z\circ w)) \quad 1,6,\sigma_{(i)} \\ 8- & T & C & :z\circ(x\circ(y\circ(z\circ w))) \quad 3,7,\sigma_{(i)} \\ & \vdots & & \end{array} \tag{3.12}$$

This fact seems to have been overlooked in (D'Agostino and Gabbay, 1994). They define a linear expansion procedure similar to algorithm 3.2 which they claim to be able to recognise every closed tree in a finite number of steps, though open completed trees can be infinite.

Derivation (3.12) shows that this is not the case unless extra restrictions are added to rule application. A possible solution would be to allow only σ_1 SLFs to "search" for σ_2 SLFs but not vice-versa (i.e. delete lines 7 and 8 in *linear-expansion*). This, however, would potentially increase the number of times the branching rule would have to be applied, thus increasing the number of variables to be introduced in the labelling expressions. Since we want to minimise the number of θ expansions (hence variables) in the derivation[9], this strategy has been rejected. Another solution would be to set an upper bound to the degree (number of connectives) of the labels admissible for σ_3 formulae based on the degree of the initial $\mathcal{T}$. This is the alternative adopted in LLKE. For this purpose we define degree of types or label expressions as follows:

Definition 3.6 *We define* degree *of types and label expressions (labelexp), $dg: \mathcal{C}\cup\mathcal{L}\rightarrow I\!N$ as follows:*

$$dg(\alpha)=\begin{cases} 0 & \textit{if } \alpha \textit{ is an atomic type or token} \\ dg(\beta)+dg(\gamma)+1 & \textit{if } \alpha \textit{ is of the form } \beta\star\gamma, \textit{ where} \\ & \star\in\{/,\backslash,\bullet,\circ,/\!\!/,\backslash\!\!\backslash\} \end{cases}$$

The restriction on maximum label degree may be implemented in *combine-σ* (step 10 of *linear-expansion*) which must then filter out all σ_3's whose labels have degree greater than the degree of the initial tableau. For all non-contractive frames no formula can have a label degree greater than the degree of the initial tableaux and satisfy the label closure condition, since in non-contractive calculi types cannot be re-used — this will be proved in section 4.3.2. For contractive frames, however, eventual labels of greater degree will be introduced as variables by application of θ rules. Given the restriction on σ rules, it is easy to see that algorithm 3.2 terminates when applied to a finite number of formulae. We will see below that this restriction can be carried through to the general LLKE algorithm without loss of generality.

Another interesting question regarding expansion by non-branching rules is: how far we can get by means of *linear-expansion* alone? The answer to this question requires additions to

[9] The reasons for this will be spelled out in chapters 4 and 5. For the time being let's just say that the more branches one has in a derivation the more costly its manipulation becomes.

the labelling apparatus which won't be made until chapter 4. However, we could anticipate some facts by having a look at the following proofs[10] derived in L without any application of rule θ.

Proposition 3.2 (Reduction Laws) *Let X, Y and Z be types, and $\mathcal{L}$ an information frame. The following properties can be proved via θ-free derivations:*

$$\begin{array}{ll} X/Y \bullet Y & \vdash X \quad \textit{and} \\ Y \bullet Y\backslash X & \vdash X \quad \textit{for any } \mathcal{L}. \end{array} \tag{3.13}$$

$$\begin{array}{ll} X/Y \bullet Y/Z & \vdash X/Z \quad \textit{and} \\ Z\backslash Y \bullet Y\backslash X & \vdash Z\backslash X \quad \mathcal{L} \textit{ associative.} \end{array} \tag{3.14}$$

$$\begin{array}{ll} X & \vdash Y/(X\backslash Y) \quad \textit{and} \\ X & \vdash (Y/X)\backslash Y \quad \textit{for any } \mathcal{L}. \end{array} \tag{3.15}$$

$$\begin{array}{ll} (Z\backslash X)/Y & \vdash Z\backslash(X/Y) \quad \textit{and} \\ Z\backslash(X/Y) & \vdash (Z\backslash X)/Y \quad \mathcal{L} \textit{ associative.} \end{array} \tag{3.16}$$

$$\begin{array}{ll} X/Y & \vdash (X/Z)/(Y/Z) \quad \textit{and} \\ Y\backslash X & \vdash (Z\backslash Y)\backslash(Z\backslash X) \quad \mathcal{L} \textit{ associative.} \end{array} \tag{3.17}$$

Proof. The proofs are obtained by straightforward application of algorithm 3.2. We illustrate the method by proving (3.13) and (3.14):

(3.13) To prove right application we start by assuming that it is verified by the identity token 1. From this we have: $1 - T : X/Y \bullet Y : m$, $2 - F : X : \mathbf{1} \circ m = m$. Then, we apply $\alpha_{(iii)}$ to 1 obtaining $3 - T : X/Y : n$ and $4 - T : Y : m/\!\!/n$. The next step is to combine 3 and 4 via $\sigma_{(iv)}$ getting $5 - T : X : n \circ (m/\!\!/n)$. Now we have a potential closure caused by 5 and 2. If we apply property (3.1) to the label for 5 we find that $n \circ (m/\!\!/n) \sqsubseteq m$, which satisfies the closure condition thus closing the tableau.

[10] See also appendix A.1 for a the full set of reduction law proof as generated by the system.

(3.14) Let's prove left composition. As we did above, we start with: $1 - T : Z\backslash Y \bullet Y\backslash X : m$ and $2 - F : Z\backslash X : \mathbf{1} \circ m$. Applying rule $\alpha_{(iii)}$ to line 1 we obtain the following SLFs: $3 - T : Z\backslash Y : a$ and $4 - T : Y\backslash X : m/\!\!/a$.

Now, we may apply rule $\alpha_{(i)}$ to 2 and expand the tree with:

$5 - T : Z : b$ and $6 - FX : b \circ m$.

Then, combining 3 and 5 via $\sigma_{(i)}$ we obtain: $7 - T : Y : b \circ a$. And finally, lines 4 and 7 can be combined through the same rule yielding $8 - T : X : (b \circ a) \circ (m/\!\!/a)$. The closure condition for 8 and 6 is achieved as follows:

$$\begin{array}{lll} (b \circ a) \circ (m/\!\!/a) & \sqsubseteq b \circ (a \circ (m/\!\!/a)) & \textit{by associativity} \\ & \sqsubseteq b \circ m & \textit{by (3.1) and } \circ \textit{ order-preserving.} \end{array}$$

The remaining proofs can be easily obtained by the same method. ■

Properties (3.13)–(3.17) correspond to Zielonka's axioms for L. If we add identity and inference rules allowing for recursion of the unary type transitions, then we get an axiomatisation of the Lambek calculus. Even though L does not enjoy a finite design — proved in (Zielonka, 1981) — the results above suggest that the calculus finds a natural characterisation in LLKE with associative information frames[11]. In chapter 4 we show the implications of this fact with respect to the complexity of the label checking module and discuss θ-free LLKE proofs in greater detail. We end this section with another example of LLKE derivation, this time one which does use a θ rule:

Example 3.2 *Prove the following:* $(A\backslash A)\backslash B \vdash_{\mathsf{L}} (B\backslash C)\backslash C$.

1−	T	$(A\backslash A)\backslash B$	$: a$	
2−	F	$(B\backslash C)\backslash C$	$: a$	
3−	T	$B\backslash C$	$: b$	$2, \alpha_i$
4−	F	C	$: a \circ b$	*idem*
5−	F	B	$: a$	$3, 4, \sigma_{ii}$
...				

[11] The Division Rule (3.17) can be regarded as L's characteristic theorem, since it is not derivable on weaker calculi such as AB, NL, and F.

Now we could apply a special case of σ_{ii} to 1 and 5, assuming that 5- F B:a is actually 5- F B:1∘a. We choose, however, to use the θ rule as follows:

			↙					↘	
6−	T	$(A\backslash A)$	$: x$	$1, \theta$	9−	F	$(A\backslash A)$	$: x$	$1, \theta$
7−	T	B	$: a \circ x$	$1, 6, \sigma_i$	10−	T	A	$: c$	$9, \alpha_i$
8−	T	C	$: b \circ (a \circ x)$	$3, 7, \sigma_i$	11−	F	A	$: x \circ c$	*idem*
	×					×			

Closure is thus achieved by replacing x with 1 *in 7 and 11 in order to solve both closure constraints, $a \circ x \sqsubseteq a$ and $x \circ c \sqsubseteq c$, simultaneously.*

3.4.1 Label upper bounds

After performing linear expansion, if the tableau is still not closed, one needs to make sure that all of its SLFs have been suitably expanded. This is done by applying the θ rule to subformulae of SLFs occurring in the tree. However, not all subformulae need to be introduced in order to generate all relevant models[12]. Definition 3.7 below limits θ rules to be applied only to certain SLFs. The fact that the restriction on σ rules discussed in the previous section and the restriction imposed below preserve completeness as well as yielding a terminating algorithm will be discussed in the following sections.

Definition 3.7 *We say that an SLF $\phi \in \mathcal{T}$ is* fulfilled *iff:*

(i) if $s(\phi) = T$ and $f(\phi)$ is of the form $A\backslash B$, then there is some $\psi \in \mathcal{T}$ s.t. $f(\psi) = A$ and $s(\psi) = T$ or $f(\psi) = B$ and $s(\psi) = F$, or

(ii) if $s(\phi) = T$ and $f(\phi)$ is of the form A/B, then there is some $\psi \in \mathcal{T}$ s.t. $f(\psi) = A$ and $s(\psi) = F$ or $f(\psi) = B$ and $s(\psi) = T$, or

(iii) if $s(\phi) = F$ and $f(\phi)$ is of the form $A \bullet B$, then there is some $\psi \in \mathcal{T}$ s.t. $s(\psi) = F$ and $f(\psi) = B$ or $s(\psi) = F$ and $f(\psi) = A$.

Provided that in all cases above ψ has not *been introduced by a θ-application to an SLF other than ϕ. We say that a branch is* completed *if it has been linearly expanded and all its formulae of the kinds described in (i), (ii) and (iii) above are fulfilled.* A tableau $\mathcal{T}$ is completed *if all its branches are completed.*

[12] Relevant models here being considered by analogy to Hintikka sets for standard predicate logic (Smullyan, 1968). In section 3.5 we will make these notions more precise.

Having set a limit up to which a tableau can be expanded we are now ready to present the higher-level expansion algorithm (algorithm 3.3). Notice that the function *select-subformula*, on line 6, will search the subtree for a formula which is non-fulfilled and return either of its subformulae, according to definition 3.7.

Algorithm 3.3 *(LLKE-completion) The complete tableau expansion for a LLKE-tree $\mathcal{T}$ is given by the following procedure:*

```
expansion(T)
1  do closure-flag ⇐ no
2  while ¬( completed(T) or closure-flag = yes)
3        do T⇐ linear-expansion(T)
4           if closed(T)
5              then do closure-flag ⇐ yes
6              else do  subf ⇐ select-subformula(T)
7                       if subf ▷ There is at least one non-fulfilled subformula in T
8                         then do subf_T ⇐ assign-label-T(subf)
9                                 subf_F ⇐ assign-label-F(subf)
10                                T_1 ⇐ append(T,{subf_T})
11                                T_2 ⇐ append(T,{subf_F})
12                                if (expansion(T_1) = yes and expansion(T_2) = yes)
13                                  then do closure-flag ⇐ yes
14                                  else do closure-flag ⇐ no
15                      else do closure-flag ⇐ no
16 return closure-flag
```

We close this section with an extension to definition 3.4. The extra clause aims at identifying a class of (sub)trees of minimal depth which depict proofs (or proof search). It is specified as follows:

Definition 3.8 *(Minimal Closure) A closed branch is said to be a* minimally closed branch *if no successful closure pair in it is derived from a successful closure pair of greater degree (by application of a rule in table 3.1 to the SLFs in the pair). A* minimally closed tree *is a tree whose branches are all minimally closed.*

Minimal closure is defined for the sake of computational economy: we want a closed tree to be found as early as possible. Although at this point we won't be using this notion directly, its purpose will become clear when we discuss the labelling mechanism in more detail in chapter 4. For the time being, we just illustrate the definition with derivation (3.18): up to line 2 we have a minimally closed tree for $(A\backslash A) \vdash (A\backslash A)$, though the fully expanded tableau cannot be considered minimally closing.

$$\begin{array}{llll} 1- & T & (A\backslash A) & : a \\ 2- & F & (A\backslash A) & : a \\ 3- & T & A & : b \quad 2, \alpha_i \\ 4- & F & A & : b \circ a \quad idem \\ 5- & T & A & : b \circ a \quad 1, 3, \sigma_i \\ & & \times & \end{array} \tag{3.18}$$

A last remark on proof search before we tackle soundness and completeness: although the search space for signed formulae is finite, the search space for the labels is still infinite. The labels introduced via θ rules are in fact universally quantified variables which must be instantiated during the label checking phase via unification. This represents no problem if we are dealing with theorems, i.e. trees which actually close. However, for completed trees with at least one open branch, the task might not terminate.

In order to deal with this problem — and bound the unification task at the labelling level, as we will see in chapter 4 — we restrict the domain of label (variable) substitutions to the set of tokens occurring in the derivation. This will be done by analogy to the way parameter instantiation is dealt with by liberalised quantification rules for first-order logic tableaux (Smullyan, 1968; Fitting, 1990), and will be managed by the module responsible for checking label closure conditions. If no θ rules are applied, then a *ground rewrite system* (Dershowitz and Jouannaud, 1990) suffices for the task. This, however, is not the case in general. The mechanisms effectively adopted in order to get around the complexities of associative rewriting are described in section 4.3.2.

3.5 Soundness and completeness

A model-theoretic semantics can be defined for LLKE (see definition 3.9 below) based on the interpretation of information tokens as *structured databases* as suggested in (Gabbay, 1994; D'Agostino and Gabbay, 1994) and mentioned above.

Definition 3.9 *A valuation over a given information frame $\mathcal{L} = \langle \mathcal{P}, \circ, \mathbf{1}, \sqsubseteq \rangle$ is a function $v : \mathcal{C} \times \mathcal{P} \to \{T, F\}$ which assigns truth-values to syntactic types, such that for each type A, v is a continuous mapping — i.e. $\sqcup\{v(A,a) | a \in S\} = v(A, \sqcup S)$ and $\sqcap\{v(A,a) | a \in S\} = v(A, \sqcap S)$ for all directed sets S. A relation "$\models$" such that $a \models A$ can be read as "type A is available at database a" can be defined through the following conditions:*

(i) $a \models A$ *iff* $v(A,a) = T$ *for all types in* $\mathcal{C}$

(ii) if $a \models A$ *and* $a \sqsubseteq b$ *then* $b \models A$

(iii) if $a \models a$ *and* $b \models A$ *then* $a \sqcap b \models A$, *for all types* A *and all tokens* x, y.

(iv) if $a \not\models a$ *and* $b \not\models A$ *then* $a \sqcup b \not\models A$, *for all types* A *and all tokens* x, y.

(v) $a \models A\backslash B$ *iff* $b \models A$ *or* $b \circ a \models B$ *for all tokens* b

(vi) $a \models A/B$ *iff* $b \models B$ *or* $a \circ b \models A$ *for all tokens* b

(vii) $a \models A \bullet B$ *iff* $b \models A$ *and* $a /\!\!/ b \models B$ *for some token* b

The semantics above differs for example from the semigroup (Lambek, 1958) and relational interpretations (van Benthem, 1991) presented in chapter 2. This is due to the fact that the primary concern here is not *semantic informativeness* — i.e. how much purely model-theoretic objects (sets and other structures) tell us about about purely syntactic objects (types and proofs) — but *coverage* of a maximum number of logics by "bringing model-theory back into proof-theory", to quote a well-known LDS slogan. In fact, the semantics of definition 3.9 doesn't play any fundamental role in the presentation of the theory, since as we will see below completeness and soundness are proved with respect to sequent presentations. We regard this to be a limitation of the system rather than an advantage. There is more to the semantics of labelling than its apparent simplicity leads one to believe. See for instance (Venema, 1996) on tree models for labelled CG for a discussion of these aspects. On the other hand, It is perhaps possible to base LLKE on a relational semantics and vary the notion of theorem-hood

by adjusting Kripke-style accessibility relations and then obtain direct completeness proofs along the lines of (Andreka and Mikulas, 1994). This, however, will not be attempted in this thesis.

The correspondence between the conditions in definition 3.9 and the labelling regime specified on table 3.1 is evident. Although a model theory so defined doesn't yield informative completeness proofs, it serves to show (see proposition 3.3 below) that the system which incorporates the restrictions on the size of the labels and on the application of θ rules (see definition 3.7) is complete with respect to the expansion rules.

Proposition 3.3 *Every completed open branch has a model.*

Proof. Inductively on the degree of SLFs in an open subtree $\mathcal{T}$ by defining a valuation function over the information frame so that for all atomic types A: (i) $v(A,a) = T$ if $T : A : a$ occurs in $\mathcal{T}$ and $v(A,a) = F$ otherwise. ■

The results on soundness and completeness presented below have been adapted from similar theorems proved in (D'Agostino and Gabbay, 1994) and (Gabbay, 1994). The soundness result stems from the interpretation of the labelling algebra as discussed in section 3.4 and sketched below. Completeness appeals to a notion of *non-critical substitutions* of atomic labels on a derivation in order to show that all rules of the sequent CG presentation can be derived through LLKE rules.

Proposition 3.4 *(Soundness) For all syntactic types X, Y, if there is a closed LLKE tree for $F : X_1, ..., X_n \vdash Y : 1$ (in* NL,...LPCE*) then there is a sequent derivation for $X_1, ..., X_n \vdash Y$ (in the respective calculus)*

Proof. Given an information frame $\mathcal{L} = \langle \mathcal{P}, \circ, 1, \sqsubseteq \rangle$, we define $\mathcal{P}$ to be a set of types closed under the (sequent) entailment relation $\vdash$. We then interpret the label composition operator as follows: $a \circ b \stackrel{\text{def}}{=} \{A \bullet B | A \in a \wedge B \in b\}$. The "identity" token will be interpreted as the set of *all* theorems in the calculus, i.e. $1 = \{A | \vdash A\}$. The partial order on the labelling algebra

defined by relation $\sqsubseteq$ will be regarded as as set inclusion.
Now, if we interpret $T : A : a$ as $A \in a$ and $F : A : a$ as $A \notin a$, we have that a closed tree for $T : X_1 : a_1, ..., T : X_n : a_n, F : Y : a_1 \circ ... \circ a_n$ implies that $Y \in a_1 \circ ... \circ a_n$. Therefore (in a sequent system) we have $X_1 \bullet ... \bullet X_n \vdash Y$ and consequently $X_1, ..., X_n \vdash Y$. This shows that under the assumptions above the LLKE rules are sound with respect to sequent rules for the calculi in the substructural hierarchy. ■

Finally, it is shown that all valid sequents sequents also receive LLKE derivations — proposition 3.5. Following (D'Agostino and Gabbay, 1994), we denote a substitution of non-critical atomic labels in $\mathcal{T}$ by $\mathcal{T}$[x/y] (i.e. $\mathcal{T}$ with all occurrences of y replaced by x). Non-critical labels are those not introduced by expansion rules. The fundamental result concerning non-critical substitutions is given by lemma 3.1

Lemma 3.1 *Non-critical substitutions preserve the structure and closure status of the trees onto which they apply.*

Given this lemma we set up a bit of notation to be used in the main result (and in similar proofs in the next chapter): δ° is taken to denote the $\circ$-concatenation of the labels assigned to each formula in Δ.

Proposition 3.5 *(D'Agostino and Gabbay, 1994) For all Gentzen proofs in the substructural Lambek hierarchy there is a corresponding LLKE proof in the respective algebra $\mathcal{L}$ with the appropriate structural constraints* (2.12)

Proof. The proof is carried out by showing that the LLKE entailment relation is closed under rules (2.8) plus zero or more structural rules (2.12) according to the target calculus. The (Id) axiom is trivial. In proving closure under the other rules, the use of θ rule is crucial. The technique is illustrated here by proving that LLKE is closed under the contraction rule (C) — closure of the system under the remaining rules can be similarly obtained. We at first assume $\Gamma A, A, \Delta \vdash_{llke} B$ and try to show $\Gamma A, \Delta \vdash_{llke} B$. Our hypothesis implies the existence

of a closed tree as follows:

$$\begin{array}{ll} 1- & T : \Gamma : \gamma \\ 2- & T : A : a \\ 3- & T : A : b \\ 4- & T : \Delta : \delta \\ 5- & F : B : \gamma^{\circ} \circ a \circ b \circ \delta^{\circ} \quad \textit{Assump.} \\ & \times \end{array} \tag{3.19}$$

Now by building a tableau for $\Gamma A, \Delta \vdash_{\mathit{llke}} B$ we arrive at a closed tree as follows:

$$\begin{array}{ll} 1- & T : \Gamma : \gamma \\ 2- & T : A : a \\ 4- & T : \Delta : \delta \\ 5- & F : B : \gamma^{\circ} \circ a \circ \delta^{\circ} \quad \textit{Assump.} \end{array} \tag{3.20}$$

$$\begin{array}{ll} 6- & F : B : \gamma^{\circ} \circ a \circ a \circ \delta^{\circ} \quad \text{rule } \theta \\ & \mathcal{T} = (3.19)[a/b] \end{array} \qquad \begin{array}{ll} 7- & T : B : \gamma^{\circ} \circ a \circ a \circ \delta^{\circ} \quad \text{rule } \theta \\ & \times \end{array}$$

The right branch closes because the closure pair on lines 7 and 5 obeys the closure constraint $\gamma^{\circ} \circ a \circ a \circ \delta^{\circ} \sqsubseteq \gamma^{\circ} \circ a \circ \delta^{\circ}$ for LPC, i.e. (3.9). The subtree $\mathcal{T}$ on the left is closed since it is a subset of the tree depicted in (3.19) with all occurrences of b replaced by a. and (3.19) is closed (by hypothesis). The lemma 3.1 guarantees that the substitution is allowed. ■

3.6 Summary

We have presented the general architecture of a tableau-based labelled deductive method for the categorial calculi defined in chapter 2, motivated the presentation of the algebraic bookkeeping devices and the tableau expansion rules from an intuitive point of view, and presented formal definitions for tableau proofs through algorithms which manipulate expansion rules. The semi-decision procedure of (D'Agostino and Gabbay, 1994) has been bounded at the level

of label introduction and a semantics based on the labelling algebra has been defined. Finally, soundness and completeness have been shown to hold with respect to the sequent systems of chapter 2. The label checking strategy and the termination results following from it still remain to be discussed in greater detail . These issues will be addressed in the next chapters.

Chapter 4

Syntactic structure and labelling

In chapter 3 we presented the basic apparatus to deal with *syntactic types* and indicated the main aspects of the other major module which is comprised in our approach to CG parsing: the labelling algebra. This chapter details the bookkeeping strategies employed in the module with emphasis on the ones designed to maintain complexity within acceptable bounds. It also introduces the discussion of how a classical problem of (both combinatorial and sequent-based) categorial systems, spurious ambiguity, affects our tableau-based approach, showing the existence of a trade-off between minimal-effort label checking and unambiguous proofs. Finally, we describe the interaction between the two main modules of LLKE with a focus on the labelling algebra and on the mechanisms defined for its manipulation, and present time complexity results for the system as a whole.

4.1 Variable-free derivations

Branch (tableau) closure checking is performed on two formulae of the same form and opposite signs. Identifying possible closure pairs is computationally easy: it requires no more than symbol identity checks. The bottleneck of closure checking is clearly situated in the task of determining whether the closure conditions specified in definition 3.4 are met. Label expressions connected by the symbol "$\sqsubseteq$" describe closure constraints which along with the

calculus-specific properties of definition 3.5 can be generalised as rewrite reductions to be solved by the label-checking module. Under this view, variable-free labels produce *ground reductions* which yield *ground rewrite systems* (Dershowitz and Jouannaud, 1990). However, careless variable introduction combined with naive rewrite techniques might render the resulting system intractable or even undecidable. In LP, for example, regarding closure constraints as an equational theory modulo associativity and commutativity one would have to perform AC-unification of label expressions which has been proved to be NP-complete (Kapur and Narendran, 1986). In L, solving constraints modulo associativity is NP-hard (Siekmann, 1989; Baader and Siekmann, 1993), and so on.

One possible way of getting around the problem is to reduce the number of variables in the tableau. Among all of LLKE rules (see table 3.1), the only rule to introduce variables in labelling expressions is θ. Definition 3.7 binds label variable introduction to the number of sub-formulae necessary to *fulfill* each formula in the tableau. Now the question is: how far can we get without having to resort to θ rules?

Recall that properties (3.13)–(3.17), which correspond to Zielonka's axioms for L, were proved in chapter 3 by means of strictly θ-free derivations[1]. This fact naturally suggests a class of theorems which can be proved in label-free derivations. Proposition 4.6 shows that in L, if associativity is treated at the level of syntactic types then algorithms 3.1 and 3.2 alone are sufficient for the purpose of generating complete sets of fulfilled types (see definition 3.7) from any input set.

Proposition 4.6 *All closed LLKE-trees for* L *derivable by the application of the set of rules* $\mathcal{R} = \{\alpha_i, \ldots, \alpha_{iii}, \sigma_i, \ldots, \sigma_{vi}, \theta, Assoc\}$ *can be also derived from* $\mathcal{R} - \{\theta\}$.

Proof. The proof relies on the fact that the Gentzen formulation of the calculi, (R/),...,(L•), can be proved in LLKE by means of θ-free derivations. We follow the pattern of proposition 3.5 (recall that we denote a substitution of non-critical atomic labels in $\mathcal{T}$ by $\mathcal{T}[x/y]$ as before. The (Id) axiom is trivial. In order to prove (R/) we suppose $\Delta, A \vdash B$ and try to show

[1] See also appendix A.1 for a complete listing of the LLKE proofs.

$\Delta \vdash B/A$. By hypothesis tree (4.1) is closed. Notice that in (4.1) δ° is taken to denote the $\circ$-concatenation of the labels assigned to each formula in $\Delta \cup \{A\}$. The rationale behind the label for Δ is the following: assuming that the label expression δ° supports the sequent Δ, A (let's denote this by $\delta^\circ \models \Delta, A$) and that there is a token, say x, s.t. $x \models A$, a label expression supporting Δ, taking into account that A occurs to the right of Δ, should contain as much information as $\bigcup\{z | z \circ \delta^\circ \sqsubseteq x\}$, i.e., $x \backslash\!\backslash \delta^\circ$.

$$\begin{array}{llll} 1- & T & \Delta & : x \backslash\!\backslash \delta^\circ \\ 2- & T & A & : x \\ 3- & F & B & : \delta^\circ \\ & & \times & \end{array} \tag{4.1}$$

Now, we try to find a closed tree for the formula in the succedent:

$$\begin{array}{llll} 1- & T & \Delta & : \gamma^\circ \\ 2- & F & B/A & : \gamma^\circ \\ 3- & T & A & : x \qquad 2, \alpha_i \\ 4- & F & B & : \gamma^\circ \circ x \quad idem \end{array} \tag{4.2}$$

Since $\mathcal{T}_{(4.1)} \subseteq \mathcal{T}_{(4.2)}[x \backslash\!\backslash \delta^\circ / \gamma^\circ]$ and $\mathcal{T}_{(4.1)}$ is closed (by hypothesis), so is $\mathcal{T}_{(4.2)}$, by the substitution lemma. The other rules are proved analogously. ■

A straightforward corollary of proposition 4.6 is that all NL-theorems are provable by variable-free derivations. The same result will not hold for LP and stronger calculi, however. The reason for this is that σ rules (particularly instances (ii) and (iii) in table 3.1) could require the labels to be structurally modified before a rule application is licensed.

In example 4.3, a detailed θ-free derivation of an L-theorem is shown:

Example 4.3 *Let's assume the following type–string correspondence:*

$$\underset{\text{John}}{NP},\ \underset{\text{loves}}{(NP\backslash S)/NP},\ \underset{\text{but}}{((S/NP)\backslash(S/NP))/(S/NP)},\ \underset{\text{Mary}}{NP},\ \underset{\text{hates}}{(NP\backslash S)/NP},\ \underset{\text{Bill}}{NP} \vdash_L S \tag{4.3}$$

Bracketing will be controlled exclusively in the labelling algebra. Syntactic types can thus be re-bracketed or have their bracketing simply ignored. After some bracket re-shuffling, we will try to find a counter-model for the following:

$$F: NP \bullet NP\backslash(S/NP) \bullet (S/NP)\backslash((S/NP)/(S/NP)) \bullet NP \bullet NP\backslash(S/NP) \bullet NP \vdash_L S. \tag{4.4}$$

Therefore, the following has to be proved:

2- T: $NP \bullet NP\backslash(S/NP) \bullet S/NP\backslash((S/NP)/(S/NP)) \bullet NP \bullet NP\backslash(S/NP) \bullet NP : m$

and

3- F: $S : m$.

We proceed by breaking 2 down with successive applications of $\alpha_{(iii)}$*:*

4- T: $NP : a$

5- T: $NP\backslash(S/NP) \bullet S/NP\backslash((S/NP)/(S/NP)) \bullet NP \bullet NP\backslash(S/NP) \bullet NP : m/\!/a$

6- T: $NP\backslash S/NP : b$

7- T: $S/NP\backslash((S/NP)/(S/NP)) \bullet NP \bullet NP\backslash(S/NP) \bullet NP : (m/\!/a)/\!/b$

8- T: $S/NP\backslash((S/NP)/(S/NP)) : c$

9- T: $NP \bullet NP\backslash(S/NP) \bullet NP : ((m/\!/a)/\!/b)/\!/c$

10- T: $NP : d$

11- T: $NP\backslash(S/NP) \bullet NP : (((m/\!/a)/\!/b)/\!/c)/\!/d$

12- T: $NP\backslash(S/NP) : e$

13- T: $NP : (((m/\!/a)/\!/b)/\!/c)/\!/d)/\!/e$

Now we start applying σ*-rules (annotated on the right-hand side of each line) to the formulae above:*

14- T: $S/NP : a \circ b$ — *4, 6,* $\sigma_{(i)}$

15- T: $S/NP : d \circ e$ — *10, 12,* $\sigma_{(i)}$

16- T: $(S/NP)/(S/NP) : c \circ (d \circ e))$ — *8, 15,* $\sigma_{(i)}$

17- T: $S/NP : (a \circ b) \circ (c \circ (d \circ e))$ — *14, 16,* $\sigma_{(iv)}$

18- T: $S : ((a \circ b) \circ (c \circ (d \circ e)) \circ (((((m/\!/a)/\!/b)/\!/c)/\!/d)/\!/e))$ — *17, 13,* $\sigma_{(iv)}$

We have derived a potential inconsistency between 18 and 3. By checking the token for 18 we verify that $(a \circ b) \circ c \circ (d \circ e)) \circ (((m/\!/a)/\!/b)/\!/c)/\!/d)/\!/e$

$$\begin{array}{lll} \sqsubseteq & (((a \circ b) \circ c) \circ d) \circ e \circ (((m/\!/a)/\!/b)/\!/c)/\!/d)/\!/e & \textit{by associativity} \\ \sqsubseteq & ((a \circ b) \circ c) \circ d \circ (((m/\!/a)/\!/b)/\!/c)/\!/d) & \textit{by property (3.1)} \\ & \vdots & \\ \sqsubseteq & m & \end{array}$$

So, repeated applications of property (3.1) *and associativity to 18 suffice to satisfy the tableau closure condition, which shows that* (4.4) *is provable in* L.

4.1.1 Branching via θ rule and spurious ambiguity

A common problem in early, sequent-based proof systems for categorial logics was the so-called phenomenon of *spurious ambiguity*. Recall that in chapter 2 we mentioned that one of the attractive features of Lambek systems is their clear syntax-semantics interface — i.e. proofs can be assigned λ-terms which keep track of the "meaning" of the derivations — through the Curry-Howard isomorphism whereby different categorial calculi correspond to different fragments of the lambda calculus (van Benthem, 1991; van Benthem, 1996). From a processing point of view, the ideal situation would be to have one and only one derivation for each semantic term in all cases, i.e. guarantee that the isomorphism holds as it does in the original natural deduction setting (Prawitz, 1965). Unfortunately, this is often not the case of sequent-based proof systems (Hepple, 1990). Derivations (4.5) and (4.6) (Hendriks, 1993) show two distinct syntactic derivations which have the same semantics.

$$\dfrac{\dfrac{\dfrac{C:x\vdash C:x \qquad C:u\vdash C:u}{C/C:u', C:x\vdash C:(u'x)}(\mathrm{L}/) \qquad C:z\vdash C:z}{C/C:z', C/C:u', C:x\vdash C:(z'(u'x))}(\mathrm{L}/) \tag{4.5}$$

$$\dfrac{\dfrac{C:x\vdash C:x \qquad \dfrac{C:u\vdash C:u \qquad C:z\vdash C:z}{C/C:z', C:u\vdash C:(z'u)}(\mathrm{L}/)}{C/C:z', C/C:u', C:x\vdash C:(z'(u'x))}(\mathrm{L}/) \tag{4.6}$$

In Gentzen–Lambek systems, this is due to the degree of freedom enjoyed by the prover with respect to choosing which rule to apply at a given point, and in the choice of the *active* type on each deductive step. The derivations above illustrate non-determinism of the latter kind. Proof normalisation (König, 1989; Hepple, 1990; Hendriks, 1993) is a technique which has been developed to deal with spurious ambiguity in substructural proof systems. Also, proof nets are known not to suffer from the problem (Roorda, 1991).

Perhaps of a more practical concern than the loss of the one-to-one map between proofs and their semantics is the amount of extra, computationally expensive work which must be done in some cases. As Eisner (Eisner, 1996) points out, the *composition* (3.14) rule — both in

Lambek systems and Combinatory Categorial Grammar (CCG) (Steedman, 1990) — causes even simple sentences to have many ambiguous parses. In CCG, associative "chains" of the forms:

$$A/B, B/C, C \tag{4.7}$$

and

$$A/B/C, C/D, D/E\backslash F/G, G/H \tag{4.8}$$

give rise to spurious ambiguity. For example, in order to derive the two readings in examples (e.19) and (e.20) one has to go through 252 different combinations (Eisner, 1996). In CCG, spurious ambiguity reflects the generality of the application rules which allow for essentially equivalent function applications to be repeatedly performed.

(e.19) (the galoot in the corner) that I said Mary pretends to like

(e.20) (the galoot) in the corner that I said Mary pretends to like

Similarly, it is clear that in LLKE if we eliminate θ rules in favour of syntactic associativity (as in proposition 4.6), σ-rule non-determinism gives rise to a large number of alternative derivations. The reader may check this by deriving NP,(NP\S)/NP,NP⊢S, re-bracketing, and proving the sequent NP,NP\(S/NP),NP⊢S. The same, however, does not occur in the original system, since associativity there is confined to the labelling algebra[2]. and therefore has no effect on the number of syntactic derivations for any given reading. Eliminating θ poses a trade-off between label unification and exhaustive σ tableau expansions: on the one hand we do not want to unleash the full power of unification, and on the other hand we want to avoid the pitfalls of combinatorial explosion. Notice, for instance, that although all label expressions in example (4.3) are variable-free, the potential efficiency gains in terms of label-checking are overshadowed by the presence of a very large number of extra σ rule instances (omitted in the example for the sake of space). Obviously, some of these extra instances lead nowhere.

[2]Incidentally, chains of type (4.7) which yield multiple derivations in CCG cause no problem to LLKE, even if associativity is permitted at the level of syntactic types.

Others, however, give rise to more alternative derivations for (4.3) than one would like to have.

The other side of this coin is that while the label algebra encodes the logical structure of the derivation, there's no obvious reason why one should assume that a closed tableau must always provide enough information to build a semantic interpretation of the sentence. In the next section we formalise and prove a positive answer to this question before tackling the label checking mechanisms. We leave the task of providing a more detailed account of spurious ambiguity in Lambek systems (as opposed to CCG) to chapter 5, where other proof systems will be discussed and compared with LLKE.

4.2 Recovering the syntactic structure of a type

LLKE derivation trees can be seen from a logical point of view as depicting a systematic search for counter-models of an assumption which one wishes to refute. However, from a CG parsing perspective one expects the display of the proof to say something about the syntax of the item being parsed, in the sense discussed in (Leslie, 1990). The topology of our proof trees does not seem to provide this kind of information.

The syntactic types whose structure we want to recover will be either basic types (NP, S etc) or types built out of basic types with operators '/' and '\'. This restriction will be more rigorously specified in BNF — see (4.20) for *SlTypes* – in section 4.3.2. However, for the time being we will just assume that we are dealing with entailment of the kind defined below, unless stated otherwise:

$$X_a \vdash X_s, \qquad \text{where type } X_s \text{ contains no '}\bullet\text{' operators.} \tag{4.9}$$

In order to be able to draw the syntactic structure of a type as a tree or to assign a semantic interpretation to a type, typically a sentence type, one needs to be able to recover from the tableau all information contributed by each conjoined type on the antecedent (the

initial T-formula). The question then arises: where in the proof-tree should we look for such information? Since we are dealing with a labelled deductive system, it seems natural that the best places to start are the spots where labels are checked, i.e. the closing pairs.

Two similar strategies could be employed within LLKE to construct semantic (lambda) terms for syntactic types as in (Morrill, 1994), for instance. One could assign lambda terms to each lexical item and then either (a) specify the semantic outcome of each α and σ rule dynamically, or (b) assign lambda terms with each initial label (i.e. labels introduced by initial formulae) and derive the semantic interpretation from the a closing pair of a closed branch via (2.15)–(2.17). However, from the tableau construction rules and algorithms, it is not immediately apparent that all tableaux should enjoy the property of containing label constraints from which one will always be able to derive a relevant syntactic tree (with respect to the target type). This property certainly does not hold for all subtrees: not all satisfiable label constraints in all subtrees say something semantically useful about the relationship between lexical items in the sentence being parsed.

For instance, consider that the closing pair on the right-hand branch in example 3.2, page 54, lines 10 and 11, carries very little structural information about the type being proved, viz. $(B\backslash C)\backslash C$. Now, compare the T-formula of this pair with the T-formula on line 18 of example 4.3. In the latter, the label expression contains exactly one sub-token for each initial sub-formula in the tableau, thus "recovering" their semantic contribution. Before we proceed, let's define formally what we mean by *recoverability* as follows:

Definition 4.10 *(Recoverability) Given a closed tableau for* $A'_1, ..., A'_m \vdash A'_0$*:*

$$\mathcal{T} = \left\{ \begin{array}{c} \left.\begin{array}{c} A_0 \\ A_{0_1} \\ \vdots \\ A_{0_k} \end{array}\right\} = |A_0|_{\alpha\sigma} \\ \triangle \\ A_{n>m} \\ \times \end{array} \right. \tag{4.10}$$

where $f(A_0) = A'_0$ *is the initial F-formula, and* $|A_0|_{\alpha\sigma} = \{A_0, A_{0_1} ... A_{0_k}\}$ *the result of applying*

algorithm 3.1 to $\mathcal{T}' = A_0$*, we say that SLF* A_n recovers the syntactic structure of *type* A'_0 *iff the following conditions hold:*

$$s(A_n) = T \tag{4.11}$$

$$l(A_n) \sqsubseteq l(A_{0_i}), \text{ where } A_{0_i} \in |A_0|_{\alpha\sigma} \text{ and } s(A_{0_i}) = F \tag{4.12}$$

Now, in the light of definition 4.10, our original question can be restated as the question of whether we are always able to find a closing match for the type on the left-hand side of the entailment relation (or a type derived from it via α rule) in all closed tableaux. In order to give this a positive answer we first show the following:

Lemma 4.2 *Given a minimally closed tableau* $\mathcal{T}$ *for* $A'_1, ..., A'_m \vdash A'_0$*, subtrees* $\mathcal{T}_0, ..., \mathcal{T}_n \subseteq \mathcal{T}$*, and an initial F-formula* A'_0*, there is at least one finite subtree* $\mathcal{T}i - |A'_0|_{\alpha\sigma}$ *which contains exclusively T-formulae.*

Proof. Let $\mathcal{T}$ be a closed tableau for $A'_1, ..., A'_m \vdash A'_0$, $\mathcal{T}_0 = \{A_0, ..., A_m\}$ be an initial tableau — see (3.5) for the definition of *initial tableau* — and distinguish a $\mathcal{T}'_0 \subset \mathcal{T}_0$ containing the initial T-formulae of $\mathcal{T}_0$. The idea is to extend $\mathcal{T}'_0$ via rules in table 3.1 so that the resulting subtree is minimally closed and contains no F-formulae. Inspection of table 3.1 shows that the rules which extend $\mathcal{T}_0$ by adding F-formulae to it are: $\alpha_{(i)}$, $\alpha_{(ii)}$, $\sigma_{(ii)}$, $\sigma_{(iii)}$ and θ. We divide the proof into two main cases: θ-free extension and branching extension.

Case 1, θ*-free extension:* By definition, all applications of α rules are exhausted in $\mathcal{T}_0$ and all instances of α and σ rules are exhausted in $|A'_0|_{\alpha\sigma}$. Now, suppose we can apply $\sigma_{(ii)}$ to a pair of SLFs in $\mathcal{T}_0 \cup |A'_0|_{\alpha\sigma}$, say X and Y, where X matches $\sigma_{(ii)_1}$ and Y matches $\sigma_{(ii)_2}$, and obtain a minimally closed branch. It's easy to see that $X \in \mathcal{T}'_0$ and $Y \in |A'_0|_\alpha \subseteq |A'_0|_{\alpha\sigma}$. Therefore, Y must have been introduced by an application of $\alpha_{(i)}$ to a SLF in $\mathcal{T}_0$, say Z. But, by definition of $\sigma_{(ii)}$ and $\alpha_{(i)}$ we have: $s(X) = T$, $s(Z) = F$, $f(Z) = f(X)$ and $l(X) = l(Z)$, thus contradicting our assumption that the branch is minimally closed. Analogous reasoning applies to $\sigma_{(iii)}$.

Case 2, branching extension: Extending a tableau $\mathcal{T}$i via θ rule we obtain two branches: $\mathcal{T}i' = \mathcal{T}i \cup \{X\}$ and $\mathcal{T}i'' = \mathcal{T}i \cup \{Y\}$, where $s(X) = T$, $s(Y) = F$, $f(X) = f(Y)$, and

$l(X) = l(Y) = n$. Ignoring the subtree dominated by Y, we try to extend $\mathcal{T}i'$ by applying σ rules. If X matches formulae σ_1 or σ_2 in rule $\sigma_{(i)}$ or in rule $\sigma_{(iv)}$, then clearly a T-formula results. If X matches σ_1 in either $\sigma_{(ii)}$ or $\sigma_{(iii)}$, then in order for a F-formula to result there must be a Z in $\mathcal{T}i$ s.t. $s(Z) = F$ and $l(Z) = n \circ m$ ($\sigma_{(ii)}$) or $l(Z) = m \circ n$ ($\sigma_{(iii)}$) for some label m occurring in $\mathcal{T}i$. Since n is a newly introduced token, no F-formula results from X. Inductively, the descendants of X fall under either case 1 or case 2. ■

Based on this result, we can now guarantee that a SLF in the tableau will always provide one with enough information to recover the information contributed by each lexical item (i.e. each initial formula) to the structure parsed.

Proposition 4.7 *For any closed tableau* $\mathcal{T}$ *for* $\Delta \vdash A_0$*, there is at least one SLF* A'_i *which recovers the structure of* A_0*.*

Proof. Two main cases: (1) A_0 is an atomic type and (2) $dg(A_0) > 0$. In case (1), by lemma 4.2 there is a branch containing only T-formulae hence in this branch A_0 is one of the formulae in the closure pair. By the tableau closure conditions (definition 3.4), the other element of the closure pair is an A_i s.t. $s(A_i) = T$ and $l(A_i) \sqsubseteq l(A_0)$. Therefore A_i recovers the structure of A_0. In case (2), if the tree closes as a result of expanding A_0, i.e. the closure pair is in $|A_0|_{\alpha\sigma}$, then recoverability is trivial. Otherwise, again lemma 4.2 and definition 3.4 guarantee that a suitable closing formula is found. ■

A final remark on recoverability should be made. It is easy to see that, in non-contractive calculi, whenever the type on the right-hand side of the entailment is atomic, its closing formula is totally (and only) labelled with structurally relevant tokens. On the other hand, if that type is not atomic then the labels eventually introduced in its α-expansion must be removed from the labelexp in the closing T-formula in order to recover the precise syntactic structure. The reader can verify the latter by proving $NP \bullet (NP\backslash S)/NP \vdash S/NP$ (see appendix section A.3 for the LLKE proof). A more complex proof tree can be seen in example 4.4.

Example 4.4 *A derivation tree for* (4.3) *with no re-bracketing allowed at the syntactical level*

is shown below. Notice that nodes not directly related to the closing pairs have been removed. The complete derivation tree can be seen in appendix A. The first θ-expansion step produce the following open tableau:

$\mathcal{T}_0 =$

1.	$\underline{F : S : l_0}$	
2.	$T : NP : l_3$	
3.	$T : ((NP\backslash S)/NP) : l_4$	
4.	$T : (((S/NP)\backslash(S/NP))/(S/NP)) : l_5$	
5.	$T : NP : l_6$	
6.	$T : ((NP\backslash S)/NP) : l_7$	
7.	$T : NP : (((((l_0 /\!\!/ l_3) /\!\!/ l_4) /\!\!/ l_5) /\!\!/ l_6) /\!\!/ l_7)$	

8.	$F : (S/NP) : v_8$	$4, \theta$	13.	$T : (S/NP) : v_8$	$4, \theta$
9.	$T : NP : l_9$		14.	$T : ((S/NP)\backslash(S/NP)) : (l_5 \circ v_8)$	$4, 13, \sigma_{iv}$
10.	$\underline{F : S : (v_8 \circ l_9)}$	$8, \alpha_{ii}$		⋮ ⋮	
11.	$T : (NP\backslash S) : (l_7 \circ l_9)$	$6, 9, \sigma_{iv}$			
12.	$\underline{T : S : (l_6 \circ (l_7 \circ l_9))}$	$5, 11, \sigma_i$			
	×				
	$(10, 12)$				

The right branch however still contains an unfulfilled formula (line 14), which allows us to continue the proof by applying the θ rule to the SLF on line 14.

	⋮				
13.	$T : (S/NP) : v_8$	$4, \theta$			
14.	$T : ((S/NP)\backslash(S/NP)) : (l_5 \circ v_8)$	$4, 13, \sigma_{iv}$			
15.	$F : (S/NP) : v_{10}$	$14, \theta$	20.	$T : (S/NP) : v_{10}$	$14, \theta$
16.	$T : NP : l_{11}$		21.	$T : (S/NP) : (v_{10} \circ (l_5 \circ v_8))$	$14, 20, \sigma_i$
17.	$\underline{F : S : (v_{10} \circ l_{11})}$	$16, \alpha_{ii}$	22.	$\underline{T : S : ((v_{10} \circ (l_5 \circ v_8)) \circ$	
18.	$T : NP\backslash S : (l_4 \circ l_{11})$	$3, 16, \sigma_{iv}$		$(((((l_0 /\!\!/ l_3) /\!\!/ l_4) /\!\!/ l_5) /\!\!/ l_6) /\!\!/ l_7))$	$21, 7, \sigma_{iv}$
19.	$\underline{T : S : (l_3 \circ (l_4 \circ l_{11}))}$	$16, 2, \sigma_i$		×	
	×			$(1, 22)$	
	$(17, 19)$				

The closure pairs of the tableau above are underlined and the pairs of numbers at the bottom indicate the lines where the closing SLFs are. The resulting constraints are satisfied with the substitution mapping $\varsigma = \{v_8 \mapsto (l_6 \circ l_7), v_{10} \mapsto (l_3 \circ l_4)\}$. *and a few of applications of* (3.6)–(3.7) *on the set of constraints:*

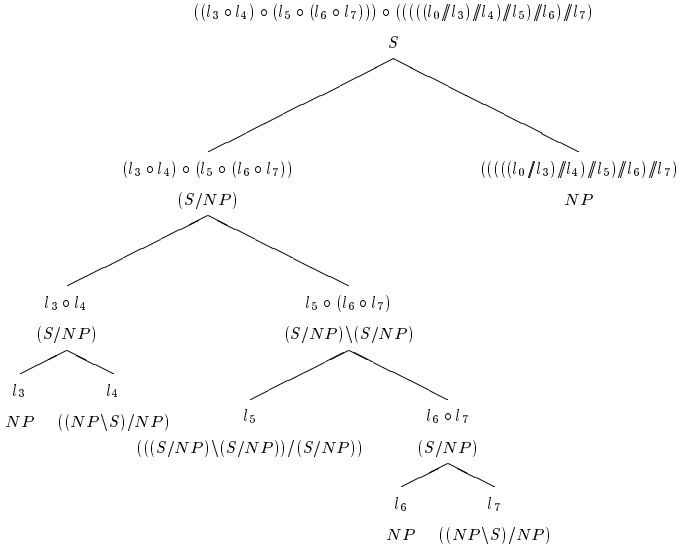
$$((v_{10} \circ (l_5 \circ v_8)) \circ (((((l_0 /\!\!/ l_3) /\!\!/ l_4) /\!\!/ l_5) /\!\!/ l_6) /\!\!/ l_7)) \quad \sqsubseteq \quad l_0 \tag{4.13}$$

$$((l_3 \circ l_4) \circ l_{11}) \quad \sqsubseteq \quad (v_{10} \circ l_{11}) \tag{4.14}$$

$$(l_6 \circ (l_7 \circ l_9)) \quad \sqsubseteq \quad (v_8 \circ l_9) \tag{4.15}$$

Extracting structural information from the relevant SLF in example 4.4 now amounts to instantiating and analysing its label on the left-hand side of (4.13). The label expression, preserving the original bracketing, yields the graph shown in (e.21).

(e.21)



The recoverability results and notions introduced in this section are relevant to the extent that they provide an effective way of extracting lexical information from a proof tree. This

guarantees that even though rule θ has no intuitive linguistic or semantic counterpart[3] its application does not result in loss or irreparable scattering of type-semantical information in a LLKE proof.

4.3 A closer look at label checking

The discussion in section 4.1.1 suggests that it is not necessarily a good idea to do away with branching (hence variable introduction) altogether. We mentioned that solving closure constraints in a tableau can be regarded as solving systems of equations modulo some set of *equalities* (Baader and Siekmann, 1993) – e.g. associativity in L, associativity and commutativity in LP etc. Left unchecked, however, such systems tend to become computationally intractable (see (Siekmann, 1989) for a survey and complexity results). Fortunately, there are particular facts about the class of logics with which we are dealing which make them not quite as hard as one would expect. We explore some of these facts in section 4.3.2. First, we borrow a few tools from unification theory in order to define the problem more precisely.

4.3.1 Word problems and unification

Let's start by recasting our label expressions in the framework of unification theory (Baader and Siekmann, 1993) and term rewriting (Kirchner, 1994). The first step is to redefine the label algebra of definition 3.2 as a language

$$T(\Omega, V) \tag{4.16}$$

where V is a countable set of variables and $\Omega = \{\circ, /\!/, \backslash\!\backslash c_0, c_1, ..., c_n\}$ is a *signature* — the first part of which is a set of fixed binary functions whereas the second is a finite set of 0-ary functions, i.e. constants c_i.

[3]Notice that the α rules can be seen as forms of lambda abstraction and σ as function application.

One way to approach label checking in $T(\Omega, V)$ is to regard certain properties of the algebra — e.g. (3.1)–(3.4) plus (3.6)–(3.8) for LP — as a set of equations, and closure constraints as *word problems* to be solved, as in (Knuth and Bendix, 1983). This involves basically the following steps: (a) an ordering relation is defined on the labelexps, also called *words*, (b) the terms of the word problem to be solved are progressively reduced until they get reduced to the same word (in which case the closure constraint is satisfied) or fail to converge. A partial order can be defined by assigning weights to each constant, variable and function symbols which make up the words. Provided that all words are well-ordered by the ordering relation so defined, one can orientate the equations as *reductions* so that the left-hand side will always be rewritten as a "smaller" word — i.e. the right-hand side labelexp.

Although the scheme above suffices in calculi such as NL and L, we soon run into trouble as we target more powerful calculi. For instance, the commutative property of LP cannot be oriented into a terminating rewrite rule, as noted in (Peterson and Stickel, 1981). An alternative to this is to treat the problematic properties separately so as to get around non-termination. This is the solution adopted in our system for label checking (to be presented in the section below) as well as in unification theory in general. In unification theory the strategy most commonly adopted is to regard subsets of the properties in definition 3.5 as sets of identities E, s.t. $E \subseteq T(\Omega, V) \times T(\Omega, V)$, and then define an *equational theory* $=_E$ so that it yields the (*E-free*) quotient algebra $T(\Omega, V)/=_E$. It is then with respect to this algebra that equational constraints are to be solved. For example, reinterpreting (3.6) and (3.7) as equalities making up a set A, we get $T(\Omega, V)/=_A$, the A(ssociativity)-free algebra *induced* by A, which characterises label checking in calculus L. When $E = \emptyset$ the equational theory characterises NL.

Our closure constraints, which have the form $X \sqsubseteq Y$, are regarded in this framework as *(in)equations* to be solved in theory $T(\Omega, V)/=_E$. In cases where constraints contain variables, these equations become *unification problems* (Baader and Siekmann, 1993). The main ingredients of the recipe to solve unification problems are *substitution mappings* defined as $\varsigma : V \to T(\Omega, V)$ such that $\{x \in V \mid x\varsigma \neq x\}$ is finite. In LLKE, the constraints to solve take the form of systems:

$$\Sigma = \{X_1 \sqsubseteq Y_1, ..., X_n \sqsubseteq Y_n\} \tag{4.17}$$

We call a substitution ς a *unifier* or *solution* of the system *modulo* E if the following E-equalities hold: $X_1\varsigma =_E Y_1\varsigma, ..., X_n\varsigma =_E Y_n\varsigma$. Thus, label checking in NL can be described as an $\emptyset$-unification problem, in L as A-unification, in LP as AC-unification and so on. Furthermore, the type of unification to be performed on the labelling algebra can be classified as *E-unification with constants* (Siekmann, 1989), since the terms may contain free constant symbols in addition to the symbols in the theory's signature[4]. A system of the form (4.17) where $\varsigma = \emptyset$ and X_i are variables occurring nowhere else in Σ is said to be in *solved form*.

Putting a system in solved form amounts to finding the most general unifier for the system. For efficiency reasons we will always be interested in finding a *minimal complete set of unifiers* for a system Σ in a theory E, denoted $\mu U_E(\Sigma)$. A set of unifiers is said to be complete if its substitutions suffice to generate all unifiers of the system by instantiation. A complete set of unifiers is minimal if no substitutions in it can be obtained by instantiation of any other substitution in the set.

Algorithms for general E-unification have a wide range of applications in theorem proving, logic and functional programming etc and so have been extensively studied over the last decades. A summary of complexity results for the most relevant E-theories is reported and discussed in (Kapur and Narendran, 1986; Siekmann, 1989; Baader and Siekmann, 1993). Table 4.1, which is based on (Siekmann, 1989), shows the complexity of the decision problem for some unification theories with constants relating the types of unification to the categorial calculi whose label-checking they characterise.

Categorial calculus	Unification type	Cardinality of $\mu U_E(\Sigma)$	Complexity
NL	$\emptyset$	≤ 1	linear
L	A(ssociative)	∞	NP-hard
NLP	C(ommutative)	> 1	NP-complete
NLPC	Idem		
LP	AC	> 1	NP-complete
LPC	Idem		

Table 4.1: Summary of complexity results for generalised label checking

[4] E-unification with constants is an intermediary class between *elementary E-unification*, which deals with constant-free terms, and *general E-unification*, which deals with function symbols of arbitrary arity.

Notice that, although closure checking for the non-associative Lambek calculus is computationally tractable on a generalised unification setting, tractability is soon lost when structural rules are added to the calculus. However, in categorial parsing specifically, the picture does not need to be as dramatic as these complexity results may lead us to believe. In the following section we will explore domain specific facts which will improve this prospect.

4.3.2 Alternative label-checking strategies

Although unification theory provides an adequate framework for describing our label-checking problem, standard rewrite techniques in their full generality are much too powerful and costly for the task if the domain is restricted to CG parsing. The best way to go about keeping complexity under control then might be to explore facts specific to the categorial domain. In the following sections we describe two kinds of techniques used to that effect: pre-processing of label constraints and bounded unification. The former is inspired by van Benthem's results on *count invariance* (van Benthem, 1986) which has been used in practice to prune the search space in sequent-based systems (Moortgat, 1988). The latter is a new strategy introduced in (Luz, 1997) and can be combined with the former to make up the whole label-checking module.

We start with defining our target types by further constraining the syntax specified by definition 2.1. The syntactic types effectively used in the calculi covered by LLKE will be characterised in terms of the following BNFs:

$$Type ::= \langle BasType\rangle \mid \langle SlType\rangle \bullet \langle SlType\rangle \quad (4.18)$$

$$BasType ::= S \mid NP \mid N \mid ... \quad (4.19)$$

$$SlType ::= \langle BasType\rangle \mid \langle SlType\rangle / \langle SlType\rangle \mid \langle SlType\rangle \backslash \langle SlType\rangle \quad (4.20)$$

Moreover, the expressions dealt with by the parser can be presented as *clauses* of the form:

$$FX_a \vdash X_s : 1, \qquad \text{where } X_s \text{ is a SlType.} \quad (4.21)$$

This restriction, a refinement of (4.9), guarantees that no $/\!\!/$ or $\backslash\!\!\backslash$ appears on the right hand side of satisfiable label constraints in θ-free derivations, as can be readily verified by inspection on table 3.1. We will refer to the restriction imposed on the calculi by attempting to prove only clauses of the form (4.21) as *syntactic restriction* (SR). The calculus obtained by replacing the product operators in L types defined according to (4.18) is known in the literature as the *product-free Lambek calculus* (Cohen, 1967). We will sometimes refer to the other calculi in the hierarchy on table (2.1) when they obey SR as *product-free Lambek calculi*. Most theorem proving techniques developed for CG parsing have been implemented in product-free calculi (Hepple, 1990). One should also note that restriction (4.9) which has sufficed so far in assuring recoverability of syntactic structure in LLKE doesn't immediately yield product-free systems. The reason for this is simple: product-connected types can still occur on the right-hand side of the entailment symbol at intermediary steps of a (Gentzen-style) derivation even if X_s is product-free.

4.3.3 Pre-processing of label constraints

The devices to be introduced in this section benefit from restriction (4.21) in order to process label constraints prior to matching and unification at relatively low computational cost. In chapter 3, we generalised the notion of *degree* to cover label expressions in addition to syntactic types with definition 3.6, page 52. Now similarly, based on property (3.1) and its right-residual counterpart, we introduce the notion of *degree of cancellation* of labelexps.

Definition 4.11 *Degree of Cancellation, dc: $\mathcal{L} \rightarrow I\!N$ is defined as follows:*

$$dc(\alpha) = \begin{cases} 0 & \text{if } \alpha \text{ is an atomic type.} \\ dc(\beta) + dc(\gamma) - 1 & \text{if } \alpha \text{ is of the form } \beta \star \gamma, \text{ where} \\ & \star \in \{/\!\!/, \backslash\!\!\backslash\} \\ dc(\beta) + dc(\gamma) + 1 & \text{if } \alpha \text{ is of the form } \beta \circ \gamma \\ dc(\beta) & \text{if } \alpha \text{ is of the form } \beta \star \mathbf{1} \text{ or } \mathbf{1} \star \beta, \text{ and} \\ & \star \in \{/\!\!/, \backslash\!\!\backslash, \circ\} \end{cases}$$

Now, among other things we want to formalise our claim, stated in section 3.4, that restricting σ rules to generate SLFs whose labels have degree no greater than a certain upper-bound does not restrict the class of theorems that can be proved via linear expansion. In order to do this we first prove the following:

Lemma 4.3 (Cancellation test) *The following restrictions hold for any satisfiable label constraint $X \sqsubseteq Y$, where X and Y are ground terms:* (i) *$dc(X) = dc(Y)$, for all non-expansive, non-contractive calculi;* (ii) *$dc(X) \leq dc(Y)$, for all non-contractive calculi.*

Proof. By restriction (4.21) and rules on table 3.1 no operator other than "$\circ$" can occur in Y. Therefore, property (3.1), associativity — i.e. (3.7) (3.6) – and commutativity (3.8) are the only ones which can be effectively applied in calculi lacking structural rules (E) and (C)[5]. Expansive calculi also admit (3.10). Induction then shows that properties (3.1), (3.6), (3.7) and (3.8) preserve *(i)*, while (3.10) preserves restriction *(ii)*. ■

This fact has been used in our implementation of LLKE to decide most label closure tests straightforwardly: in many cases it suffices to test the label formula with respect to degree restrictions instead of applying potentially more wasteful rewrites. We mentioned that our *degree of cancellation* lemma is related to the *count invariance* property (van Benthem, 1986). In order to prove a count invariance theorem for the sequent version of L (and LP), van Benthem defines a *count function* which compares two types, returning zero if the two are identical primitive types, one if they are different basic types, and incrementing or decrementing a counter depending on whether the types are multiplications or divisions respectively. Count invariance then says that for (the Gentzen formulations of) L, LP and their non-associative counterparts, all counts of primitive types in the sequent with respect to the antecedent formula must equal the corresponding counts with respect to the consequent. Testing for count invariance has been used in generate-and-test implementations of sequents for categorial (Moortgat, 1988) grammars as a way to evaluate the search space of some nodes before actually exploring it. The test could also be used in LLKE as follows:

[5] See (2.12).

Definition 4.12 (Count):

$$\begin{cases} ct(X,X) & = 1 & \text{if } X \text{ is an atomic token.} \\ ct(X,Y) & = 0 & \text{if } X,\ Y \text{ are atomic tokens and } X \neq Y. \\ ct(X,Y/\!\!/Z) & = ct(X,Y) - ct(X,Z) & \\ ct(X,Z\backslash\!\!\backslash Y) & = ct(X,Y) - ct(X,Z) & \\ ct(X,Y \circ Z) & = ct(X,Y) + ct(X,Z) & \end{cases}$$

Proposition 4.8 (Count invariance) *For all labelexps X, Y of non-contractive frames, and all atomic sub-tokens $X_1, ..., X_n$, of X, if $X \sqsubseteq Y$ then $ct(X_i, X) \leq ct(X_i, Y)$, $i = 1, ..., n$.*

Count invariance tests are more effective than cancellation test in foreseeing unsatisfiability of constraints in the label search space. In fact, Pentus (Pentus, 1994a) suggests a proof method for L based almost exclusively on invariance properties. However, count invariance algorithms are less efficient than cancellation checking. Notice that the latter can be performed in linear time on the degree of the input formula by simply "flattening out" the labelexp and subtracting the number of "$\circ$" from the number of "$\backslash\!\!\backslash$" and "$/\!\!/$" found in the resulting string (identity elements being ignored). Count invariance, on the other hand, requires each distinct atomic token to be tested against X and Y each down to its atomic tokens. Therefore, even if $ct(X_i, Y)$ can be calculated in linear time, the overall complexity of count invariance is $O(n^2)$ on the number of atomic tokens in the constraint[6]. Since labelled tableau systems typically require a large number of closure tests to be performed, the analysis above suggests that in LLKE one profits more from testing degree of cancellation than count invariance — which obviously doesn't preclude one from using count invariance to check the initial entailment.

We now establish a limit to the size of labelexps thus binding an otherwise infinite search space — see derivation (3.12) for an example. Proposition 4.9 states that we do not miss out any theorem of non-contractive calculi by setting the sum of the degrees of antecedent and consequent of the formula that we want to prove as the upper bound for the size of the labelexps introduced via linear expansion.

[6]This estimate is based on the assumption (which we haven't verified) that count invariance can be performed in $O(n)$. If a naive "divide and conquer" strategy is adopted, then the the checking algorithm becomes even less efficient, i.e. $ct = O(n^2 \lg n)$.

Proposition 4.9 *For all non-contractive calculi, if the linear expansion of an initial formula $FX_a \vdash X_s : 1$, where X_s is a SlType (see (4.18)), results in a closed tableau $\mathcal{T}$ with closure pair $\langle \alpha, \beta \rangle$, where $s(\alpha) = T$ and $s(\beta) = F$, then $max(dg(l(\alpha)), dg(l(\beta)))$ does not exceed $max(dg(X_a), dg(X_s))$.*

Proof. First, show that there is no ψ in $\mathcal{T}$ s.t. $max(dg(X_a), dg(X_s)) < dg(\psi)$. This is done by induction on α and σ rules (see fig 3.1), noticing that:

$$dg(f(\alpha_2)) = dg(f(\alpha_3)) < dg(f(\alpha_1)), \text{ for } \alpha_i, ..., \alpha_{iii}. \tag{4.22}$$

and likewise:

$$dg(f(\sigma_3)) < dg(f(\sigma_1)), \text{ for } \sigma_i, ..., \sigma_{vi}. \tag{4.23}$$

Now, looking at rules α_i, ..., α_{iii}, σ_i, ..., σ_{vi} we see that the degree of the labels to be introduced never exceeds the degree of the formula(e) on which any of these rules is applied. Furthermore, α_i, ..., α_{iii} are the only rules to introduce new information tokens. The above plus lemma 4.3 complete the proof. ■

A final remark on label bounds: although proposition 4.9 does not hold for contractive calculi, it is convenient to set an upper bound for the size of labelexps (i.e. a limit to the application of σ rules) in these calculi anyway, letting variables account for contraction (via θ rule).

4.3.4 Cancellation constraints

In the previous section we described the pre-processing of label constraints by testing count invariance and label-degree upper bounds. Now we move on to present the complete matching and unification strategies for NL, ..., LPCE. The basic idea is to treat structural properties of information frames in two phases: (a) handling of associativity by encoding labelexps into special data structures, and (b) progressive reduction and checking (matching and unification) of these structures according to the properties allowed: permutation, contraction and expansion.

In phase (a), labelexps are converted into what we call `canconsts`. Each `canconst` is composed of pairs of substructures called `+struct` and `-struct`, each of which is built as lists of stacks so as to keep track of the relative positions of multiplicative tokens — those connected by "$\circ$" — with respect to division substrings (or subtrees) within labelexps.

The way by which the resulting structures are interpreted by the label checking algorithm depends on the target calculus: they are treated as lists for NL and L, and as *multisets* (bags) for commutative, expansive and/or contractive calculi). Similarly, the `canconst` construction algorithm varies according to whether the target calculus is associative or non-associative. The former just requires flattening the labelexp and moving a pointer from left to right over the string, looking for atomic tokens; once one is found the pointer is shifted rightwards until it hits a connective or the end of the string. The token is then pushed either onto a stack in `+struct` or in `-struct` depending on the relation between previous and next connectives. Let's assume for the sake of the argument that there exists a "null" connective $*$ in addition to $\circ$ and $/\!\!/$, and that $*$ denotes the beginning or the end of a labelexp. Given an information token, say t, a "next" connective n (the first one to the right of t) and a "previous" connective p (the one immediately to the left of t), there are four main cases:

- if $n = \circ$ and ($p = *$ or $p = \circ$) then we push t into a stack in `+struct`.
- if $n = *$ and $p = \circ$ then we push t into a stack in `+struct`.
- if $n = /\!\!/$ or ($p = /\!\!/$ and $n = *$) then we push t into a stack in `-struct`.
- if $p = /\!\!/$ and $n = \circ$ then we push t into a stack in `-struct`, start new stacks in `-struct` and `+struct` and apply the cases above to the labelexp to the right of n.

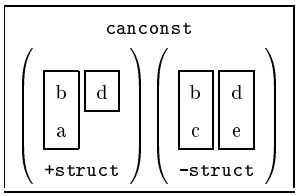
Similar cases can be derived for $\backslash\!\backslash$. The cancellation structures will partition labelexp into stacks of labels to be cancelled. For non-associative calculi the only difference is that the algorithm searches through a tree rather than a string. Both construction algorithms can be performed in linear time.

Flattening of words is a technique often employed in term rewriting systems for efficiency reasons. In (Kirchner, 1994, chapter 4) the technique is used to transform structured terms

into pairs composed of function symbols and multiset of constants and/or variables which are then taken as the inputs of the AC-unification algorithm. A somewhat more elaborated representation of flattened terms compatible with the Knuth–Bendix procedure is presented in (Christian, 1989). The technique we present in this chapter to reduce labelexps to `canconsts` takes advantage of the cancellation properties of the labelling and therefore may be seen as an instance of flattening as described in (Kirchner, 1994).

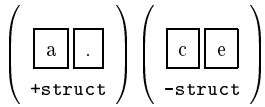
In (4.24) we see an example of labelexp and its corresponding `canconst` in an associative logic. The labelexp is treated as a string: order-relevant information is preserved but the original tree structure is lost. Notice that the number of stacks in `+structs` and `-structs` is determined by the number of division connectives in the labelexp so as to preserve order-relevant information.

$$(a \circ (b \circ (c /\!\!/ b)) \circ ((d \circ e) /\!\!/ d)) \Rightarrow$$



(4.24)

Once `canconsts` have been generated, the phase (b) involving matching and (possibly) unification starts. The structures resulting from labelexps on the right and left-hand side of the constraints are progressively reduced until they either match or fail to, at a point where no further reductions are possible. `canconst` reduction in L consists simply of popping elements off the i^{th} stack in `+struct` if they match (or unify with) elements of the i^{th} stack in `-struct`, for all stacks in `+struct`. For example: (4.24) gets reduced to (4.25).



(4.25)

A `+struct` will invariably contain the same number of lists (or sets, or multi-sets, depending

on the calculus) as its `-struct` counterpart. We refer to a `+struct` is the *complement* of a `-struct` (and vice-versa) if they have been generated from the same labelexp. Analogously, we refer to the i^{th} element of a `+struct` as the complement of the i^{th} element of its complement, and vice-versa. Where label variables are present, they get instantiated with as many elements of the complement as necessary for the matching to succeed on the next position. Global instantiation lists are built and kept for one branch, say the left branch, and used in variable instantiation when right-branch constraints are checked. The basic label checking mechanism (for NL and L) is given in algorithm 4.4. Dealing with non-associative calculi differs mainly in the `canconst` building procedure. Instead of reading atomic terms linearly off a string, we read them off a tree structure before pushing them into the appropriate `+struct` or `-struct`. After the `canconsts` have been built, label checking can be performed by algorithm 4.4 exactly as in L.

Algorithm 4.4 *(Label Checking) Given labelexps X and Y, where X is the label of a T-formula and Y labels an F-formula, and the* cancellation constraints*, pairs $\langle$`+struct`$_l$, `-struct`$_l\rangle$ and $\langle$`+struct`$_r$, `-struct`$_r\rangle$, for X and Y respectively, we define:*

label-check(X, Y)
1 do $<$ `+struct`$_l$, `-struct`$_l$ $>$ $\Leftarrow$ build-canconst(X)
2 $<$ `+struct`$_r$, `-struct`$_r$ $>$ $\Leftarrow$ build-canconst(Y)
3 while `+struct`$_l$ $\triangleright$ reduce each of the sublists
4 do list$_p$ $\Leftarrow$ pop(`+struct`$_l$)
5 list$_m$ $\Leftarrow$ pop(`-struct`$_l$)
6 redlist$_l$ $\Leftarrow$ append(redlist,reduce(list$_p$,list$_m$))
$\triangleright$ reduce$_2$ sets a global list of bindings as a side-effect
7 while `+struct`$_r$ $\triangleright$ do the same for the canconst of the labelxp on the right-hand side
8 do list$_p$ $\Leftarrow$ pop(`+struct`$_r$)
9 list$_m$ $\Leftarrow$ pop(`-struct`$_r$)
10 redlist$_r$ $\Leftarrow$ append(redlist,reduce(list$_p$,list$_m$))
11 do result $\Leftarrow$ match(redlist$_l$,redlist$_m$)
12 return result

Structural rules other than associativity are dealt with at reduction time, therefore requiring modifications in algorithm 4.4. Commutative logics are treated by simply allowing elements extracted from `+struct` to search through `-struct` for their matches and/or unifiers. Contractive frames allow deletion of repeated occurrences if necessary, on the resulting `canconsts`. Finally, expansive label checking uses marking of tokens instead of deletion; tokens which have been marked as used in a previous step can be reused as many times as necessary.

The use of `canconsts` also seems to provide a fairly general and straightforward way of treating structural modalities: in a system based on L enriched with a commutative modality, for example, tokens introduced by types marked by commutative operators would be free to move within `+structs` and `-structs` while the remaining tokens would obey the constraints described above.

4.4 Termination and Computational Complexity

There are two interdependent modules in LLKE whose termination ought to be guaranteed in order to assure termination for the system as a whole. These are: the tableau construction module — which expands the set of syntactic types — and the label-checking module — which is in charge of deciding whether a given set of closure pairs meet the requirements for a given proof tree to be considered closed according to the target calculus. In this chapter we have verified the fact — first mentioned in chapter 3, derivation (3.12) — that termination in both modules depends on the upper bound one sets to label introduction.

In the syntactic module, definition 3.7, page 55, in conjunction with the bounding of admissible labelexps within the set of words whose size does not exceed a certain degree, as expressed in proposition 4.9, 81,suffices to guarantee that after a finite number of steps all applicable rules of table 3.1 have been applied so as to produce a proof-tree in which either all formulae are fulfilled or all branches are closed.

Termination in label-checking of `canconsts` is easily verified by noting that: (a) the `canconst` building function is not essentially recursive (although we have presented if in a recursive form for the sake of clarity), (b) each iteration in the label-checking algorithm 4.4 reduces the input constraints. These facts hold true for labelling algebras characterised by various structural properties. In what follows we analyse other aspects of the interaction between the two main

modules.

4.4.1 Dynamic caching of variable bindings

A straightforward, however both naive and costly approach to label checking would be to perform full tableau expansion up to the point where all formulae in it are fulfilled and only then start the search for closing pairs, solving label constraints as a unification problem described by an equational system such as (4.17). This approach is costly because it would not only require all syntactic expansions to be performed regardless of whether a branch closes before all formulae in it are fulfilled or not, but also because typically many different sets of closure pairs would have to be checked until eventually either a satisfiable one would be discovered or all possibilities exhausted. The strategy is naive because both tableau expansion and label-checking can be performed concurrently without any loss of generality.

In a word, in order to guarantee that the sound incremental variable instantiation and label checking are performed all one has to do is to expand *first* all the left-hand branches introduced by applications of θ rules, i.e. the ones which introduce F-formulae. Given a set of closure constraints to be fed to algorithm 4.4 one wishes them to be ordered so that the constraints with the least number of uninstantiated variables get processed first. This is meant to ensure that the set of substitutions (bindings) returned by the algorithm at each step is the most general. For instance, if (4.13) in example 4.4 gets tested before (4.14) variable v_{10} might end up being assigned the value $((((l_3 \circ l_4) \circ l_5) \circ l_6) \circ l_7)$ thereby causing the tableau not to close as intended.

Let $v(l)$ be the set of variables occurring in a label or label closure constraint l and define a relation $\preccurlyeq$ on the set of closure constraints as follows: for all closure constraints c, d

$$c \preccurlyeq d \text{ if } \begin{cases} var(c) \subseteq var(d) \\ \text{or} \\ var(c) \cap var(d) = \emptyset \text{ and } |var(c)| < |var(d)| \end{cases} \tag{4.26}$$

It's easy to see that $\preccurlyeq$ defines a partial order on the set of label constraints and that if label constraints are solved in the order imposed by $\preccurlyeq$, then variable instantiation will go from the most general to the least general binding. Now, given the ordering of formulae yielded by the θ rule, if we perform a depth-first search on a LLKE-tree for closure constraints, the list resulting from this search is partially ordered by $\preccurlyeq$:

Proposition 4.10 *Let $dfirst(\mathcal{T}) = \langle c_1, ..., c_n \rangle$ be a list of closure constraints resulting from a depth-first search on $\mathcal{T}$. For all $c_i, c_{j>i} \in dfirst(\mathcal{T})$, either $c_i \preccurlyeq c_l$ or $c_i \| c_j$.*

Proof. Table 3.1 shows that θ applications always start subtrees with initial F-formulae on the left-hand branch. In derivations obeying the recoverability constraints stated in section 4.2, the F-formula in closing pairs on left-hand subtrees will always contain a label which is either a newly introduced variable or a newly introduced label variable conjoined with newly introduced label constants (see the proofs of lemma 4.2 and proposition 4.7). ■

The practical significance of proposition 4.10 is that it guarantees that one can always check label closure on LLKE-trees as soon as one encounters a closure pair. The ability to do so is fundamental not only as far as it allows a tableau to be closed as soon as a minimal set of satisfiable label constraints is found for a valid sequent but also as far as the complexity of the entire system is concerned. This is so even in cases where an open tableau results.

4.4.2 Time complexity

Let m be the number of SLFs in a tableau $\mathcal{T}$. We will assume the size of the input to the syntactic module to be given by the sum of the degrees of each type in $\mathcal{T}$:

$$n = \sum_{m \in \mathcal{T}} dg(f(m)) \tag{4.27}$$

With the input measured as above, and ignoring label-checking for the time being (i.e. deciding label-checking in time $O(1)$ by assuming that no closure constraint is satisfiable), it can be

shown that the algorithm for θ-free branch expansion is $O(n^2)$ (D'Agostino and Mondadori, 1994). We call LLKE derivations where label-checking is ignored *label-free LLKE expansions*. Moreover, a completed tree — i.e. one which all types are fulfilled (definition 3.7) — can be obtained from a given set of SLFs in polynomial time:

Proposition 4.11 *Any label-free LLKE expansion with a fixed number of θ-applications, say k, can be performed in polynomial time.*

Proof. First notice that all linear rules in table 3.1 (i.e. α_i,...,α_{iii} and σ_i,...,σ_{vi}) obey the subformula property and that the number of distinct subformulae of the initial tableau is at most n. Therefore, algorithm 3.2 can be performed in time $O(n^2)$. Now, a tree with n distinct subformulae where the θ rule has been applied at most k times (a fixed constant) on each branch will exhibit at most $2^k - 1$ branching points in total, or a maximum of $2^{k+1} - 1$ subtrees, that is n^{2^k-1} possible arrangements of θ-applications. Since k is a constant and linear expansion of each subtree is $O(n^2)$, a polynomial upper bound of $O(n^{2^k+1})$ is achieved. ■

If the entire system is considered, label complexity has to be added on top of this complexity result. There are essentially three procedures involved in label-checking, as seen above: (a) cancellation test, (b) `canconst` generation, and (c) `canconst` check. These are performed not on all SLFs in $\mathcal{T}$ but only on closure pairs. Since we set a limit to the size of labelexps to be a function of the size of the input types, we can safely assume the input to the label checking algorithms to be of size n as above.

We have seen that items (a) and (b) of the labelling module are linear on the size of the input. Item (3), `canconst` check, involves reducing the `+structs` against `-structs` on both sides of the closure constraint and then matching the results. Assuming each `+struct` and `-struct` to contain $n/2$ atomic tokens, reduction in total will be $O(n^2/2)$. Given the restrictions discussed above, even in the worst (and somewhat unrealistic) case of every new formula introduced in $\mathcal{T}$ forming a closure pair with every other formula in $\mathcal{T}$ the overall complexity will still be in $O(n^{2^{k+1}+6})$, for a fixed number of nested θ-applications k.

4.5 Summary and Discussion

Labelling in substructural logics is expensive in general. However, as we argue in (Luz, 1996a), due to its particular characteristics CG parsing appears to present an application of labelling which is feasible and can be efficient. This chapter has presented and discussed strategies to tame the target categorial calculi into natural, computationally tractable fragments. Interestingly enough, the complexity results obtained through such techniques are reminiscent of those for normally achieved for context-free and mildly context-sensitive grammars ($O(n^3)$ and $O(n^9)$, respectivelly).

On the parsing front, we have showed how to recover the syntactic structure of a type being parsed from an LLKE proof-tree so as to comply (at least partially) with the requirements on *display of proofs* discussed in (Leslie, 1990). We have also introduced the issue of spurious ambiguity, mainly from the perspective of (Eisner, 1996)[7], into the question of how to treat label constraints showing that any efforts to minimise variable introduction in the labelling algebra by eliminating the branching rule will necessarily have the unfortunate effect of giving rise to spurious LLKE-proofs.

[7] Spurious ambiguity from a Gentzen sequent perspective as in (Hepple, 1990) will be treated in chapter 5

Chapter 5

Redundancy in Labelled and non-Labelled CG Deduction

This chapter presents, discusses and compares different proof strategies aimed at improving efficiency of some CG deduction systems. In addition to LLKE, the presentation covers proof normalisation, parts of natural deduction and proof-nets. It focus on proof structure and therefore starts with an analysis of sequent calculi.

It is widely recognised that cut-free sequent systems tend to be highly redundant which causes inefficiency in implemented systems. This problem, which appears disguised in sequent calculi as multiple proofs for equivalent sequents (Hepple, 1990), gets inherited even by cut-free systems in which non-determinism is reduced such as standard tableaux (Boolos, 1984). In the case of automated deduction in substructural logics in general, and categorial grammars in particular, extra bookkeeping mechanisms are often needed which tend to accentuate complexity problems. The techniques to cope with these extra bookkeeping tasks in strictly cut-free contexts are well represented in proof-nets (Roorda, 1991), labelled systems (Moortgat, 1992) and combinations of both (Morrill, 1995b). In this chapter we will discuss some of these systems (and the assumptions behind them) in the light of what has been presented in previous chapters and make the case for the use of a tableau system with controlled cut such as LLKE in automated CG deduction.

5.1 CG sequents revisited

Gentzen's sequent calculus is normally regarded as the archetypal proof system in proof-theory. There are numerous reasons for this. First of all, having been introduced as the result of a criticism of classical logic and its hidden assumptions, the calculus permits a fine-grained study of proof structure by distinguishing between *operational* and *structural* rules, as seen in chapter 3 and discussed in (Girard, 1987; Došen, 1992) among others. Perhaps more importantly from the perspective of this thesis is the fact that the calculus facilitates the detailed study of algorithmic aspects of deduction — recall the relation between the *Hauptsatz* (Gentzen, 1969) (i.e. cut-elimination[1]) and the determinism of computation[2].

In fact, many proof systems originated from cut-elimination both for classical logic (Fitting, 1990) and "resource" logics (Girard, Lafont, and Regnier, 1995). Cut elimination, however, causes logical distortions in the deductive apparatus[3] which may have negative consequences on efficiency. However, since sequent calculi are *natural* proof-theoretic devices (in the sense explained above) and given that they enjoy decidability provided that the cut rule can be eliminated, it seems also natural to take these calculi to be the starting points of automated CG deduction. In fact, the first theorem provers for L (Moortgat, 1988; König, 1989; Hepple, 1990) relied heavily on principles derived directly from Gentzen's rules. The Prolog matching engine and database search mechanisms provided to these pioneer systems an economical — from a notational point of view — and straightforward way of encoding the deductive apparatus. Moreover, proof search in sequent systems involves choice, i.e. there are points at which the prover is given two or more alternatives as to *what rule* to apply and *which type in the sequent* to apply a rule to — the latter being called the *active type*. Again, it turns out that *backtracking* in logic programs is nicely suited to the purpose of accounting for this form of non-determinism through a generate-and-test setting.

The problem however arises that the extensive search regime enforced by the backtracking engine on sequent proofs not only produces all "relevant derivations" but also a (potentially very

[1] Which also corresponds to *normalisation* in λ-calculus (Girard, 1995).
[2] Girard: "A sequent calculus without cut-elimination is like a car without engine" (Girard, 1995).
[3] (D'Agostino and Mondadori, 1994) argues that if cut is eliminated there is no rule in the system expressing the *principle of bivalence*.

large) number of other proof trees which have essentially the same semantics, thus rendering the whole system highly inefficient, even for fragments of the calculi such as the product-free fragment. The notion of *relevant derivation* here is assumed to be characterised with respect to the Curry-Howard correspondence[4] stated in chapter 2. In addition to redundant proofs, the non-determinism of sequent formulations tend to give rise under a backtracking regime to a considerable number of partial proofs in the problem space. Smullyan-style tableaux, on the other hand, can be seen as deduction systems in which this inherent non-determinism is eliminated from the rules altogether. In a CG context, however, tableaux are not sensitive enough to capture proof structure, requiring therefore labelling or some other sort of external bookkeeping device to do the job. In what follows we discuss strategies for getting around redundancy in sequent calculi and labelling in Smullyan-style tableaux, relating computational features of both to LLKE.

5.2 The problem of proof redundancy revisited

Proof redundancy, which in CG sequent calculi gives rise to what has been named (in a CCG context) *the spurious ambiguity problem* (Pareschi and Steedman, 1987), was first addressed in a Gentzen proof-theoretic sequent setting in (Hepple and Morrill, 1989) and (König, 1989). Subsequent attempts to tackle the problem include Moortgat's (Moortgat, 1990b), Hepple's (Hepple, 1990) and Hendriks' (Hendriks, 1993). Of those, Hepple's is probably the most rigorous and comprehensive.

5.2.1 Proof normalisation by derivation constraints

The basic step towards normalisation in (König, 1989) is the partitioning of the set of sequent proofs into equivalence classes. Once these are defined, one proof of each equivalence class is picked out to represent the *normal proof*. In order to define these classes, sequent proofs are mapped onto *syntax trees* built in accordance to the following (Curry-Howard)

[4] I.e proofs yielding the same lambda-terms (or lambda terms whose normal forms coincide, via Church-Rosser property).

correspondence: $(L/)$ and $(L\backslash)$ correspond to function application whereas $(R/)$ and $(R\backslash)$ correspond to function abstraction. The rules stated in (5.1) show the intended correspondence in terms of "annotated types". This is essentially the system (2.18), already seen in chapter 2, except that application of a function t_f to argument expression t_a is denoted here by $t_f[t_a]$ and abstraction of a term, say (b/a), over another term t is represented by $`(b/a)'(t)$. This somewhat less perspicuous notation is meant to encode information (i) about the "root" category itself and (ii) about the order of the arguments. Once such information has been encoded one is able to show that it is possible to construct a syntax tree for every proof tree and then reverse the process deriving a unique proof tree from every syntax tree, respectively the "syntax-tree construction" and "proof reconstruction" algorithms in (König, 1989). Since structurally equivalent proofs are mapped into the same syntax trees, proof reconstruction guarantees that all proofs of an equivalence class get mapped onto a single proof.

$$
\begin{array}{lll}
(R/) & \dfrac{\Delta, A \vdash B : t}{\Delta \vdash B/A : `(b/a)'(t)} & (\mathrm{L}/) \quad \dfrac{\Gamma \vdash C : t_a \quad \Psi, A : t_f[t_a], \Phi \vdash B : t}{\Psi, A/C : t_f, \Gamma, \Phi \vdash B : t} \\
(R\backslash) & \dfrac{A, \Delta \vdash B : t}{\Delta \vdash A\backslash B : `(a\backslash b)'(t)} & (L\backslash) \quad \dfrac{\Gamma \vdash C : t_a \quad \Psi, A : t_f[t_a], \Phi \vdash B : t}{\Psi, \Gamma, C\backslash A :, \Phi \vdash B : t}
\end{array}
\tag{5.1}
$$

Syntax-tree construction and proof reconstruction *per se* do not play any role in the parsing algorithm. They are just the proof-theoretical devices used in a constructive proof of the existence of equivalence classes across sequent derivations from which a normal proof can be selected. The parsing algorithm works essentially by imposing restrictions — derived naturally from the proof reconstruction algorithm — on the application of sequent rules. König calls these restrictions *nesting constraints*: (i) preference on the choice of an active type — i.e. the complex type to be decomposed in a top-down application of a sequent rule in (5.1) — is always given to a succeedent type; (ii) when a non-atomic active type occurs in the antecedent its subtypes must be immediately decomposed in the next steps of the derivation; (iii) a functor type in the antecedent cannot be chosen as an active type unless its head is identical to the type in the sequent.

The original system is translated into a "natural deduction"[5] system in (König, 1991). This is possibly motivated by the "unsafeness" and high complexity of the extended sequent system (5.1). In fact, it is shown in (Hepple, 1990) that not only does the combination of syntax tree and proof reconstruction techniques fail to produce normal proofs in some cases[6] but also the parsing method derived from the nesting constraints is still not restrictive enough to prevent the occurrence of redundant derivations. The new system is shown in (5.2), in sequent notation. It still uses partial trees for semantic reconstruction, though their role is significantly downplayed. Normal proofs are achieved by means of two main constraints. One is an adaptation to L of Prawitz' normal form (Prawitz, 1965), a cut-elimination theorem in disguise. The other is a restriction on the interaction between the axiom scheme, (AX), and the elimination rules, (/E) and (\-E): non-atomic types occupying argument positions in elimination rules are not allowed to instantiate axiom schemes.

$$\frac{}{\mathrm{A} \vdash \mathrm{A}}\,(\mathrm{AX}) \qquad \frac{\Delta, A \vdash B}{\Delta \vdash B/A}\,(/\mathrm{I}) \qquad \frac{\Gamma \vdash A \quad \Psi \vdash B/A}{\Psi, \Gamma \vdash B}\,(/\mathrm{E}) \qquad \frac{A, \Delta \vdash B}{\Delta \vdash A\backslash B}\,(\backslash\mathrm{I}) \qquad \frac{\Gamma \vdash B \quad \Psi \vdash B\backslash A}{\Gamma, \Psi \vdash A}\,(\backslash\mathrm{E}) \tag{5.2}$$

The scheme is general enough to accommodate different parsing strategies. Bottom-up, top-down, shift-reduce and chart parsing methods are discussed in connection with the basic natural deduction presentation of L. The complexity results for these in product-free calculus of (5.2) are still quite discouraging: the chart-parser is $O(n!)$. However, if the parser

[5] At this point some terminological clarification is necessary. It should be noticed that, although the system described above is said to be a system of *natural deduction* in (König, 1989), it is in fact a Gentzen-style sequent system rather than the method known by that name in the theorem-proving literature (Fitting, 1990). In the former, the term "natural deduction" refers to deductive methods where the operators of a logic are treated explicitly by the deduction rules, as opposed to Hilbert-style systems where deduction is performed by closing a set of axioms under inference rules. LLKE, standard tableaux, proof nets and even resolution, mistakenly identified in the paper as an example of a Hilbert system, are all "natural deduction" according to this point of view. In the latter the use of the term is generally restricted to systems similar to the one described in (Prawitz, 1965).

[6] That is, although proof reconstruction assigns *unique* readings to structurally identical syntax trees, not *all* syntax trees yield "reconstructed" proofs. This is the case of the derivation for $S\backslash NP/NP \vdash S\backslash NP/NP$, for example.

is constrained so as to allow no more than two-fold phrase extraction a polynomial time upper-bound can be achieved. The system appears to have developed towards formulations which diverge from the original philosophy of Lambek calculi, incorporating feature-structure unification techniques similar to those used in unification categorial grammar (Calder, Klein, and Zeevat, 1988), its descendants (König, 1995) borrowing considerably from other lexical formalisms such as HPSG.

The techniques used by König to deal with proof representation in the construction of equivalence classes exhibit similarities with the labelling discipline adopted in LLKE. The motivations behind each bookkeeping system, however, are different. While König's goal is to achieve unambiguous proofs by relating the structure of *proof trees* (i.e. those whose nodes are sequents) and *syntax trees* (i.e. those whose nodes are either lexical types or place-holders for arguments of lexical types), LLKE aims at decoupling the rules governing the behaviour of function application and the "semantics" (in the restricted algebraic sense defined in chapter 3) of these rules so as to enable maximal generality in the characterisation of different calculi. Not surprisingly, there are parallels between the syntax-tree construction algorithm and our recoverability result (Proposition 4.7). However, LLKE seems to stand on more solid logical principles than the parsing mechanisms developed in (König, 1989). It is pointed out in (Hepple, 1990, pp 187–189) that the parsing method fails to reflect the asymmetry between the simplest proof of $(NP\backslash S)/NP \vdash (NP\backslash S)/NP$ (by instantiation of the axiom scheme) and its proof by full decomposition ("unfolding") of the types, blocked in König's system. Now, compare this fact with the LLKE derivation in (e.22).

(e.22)
$$\begin{array}{c} T : (NP\backslash S)/NP : a \\ F : (NP\backslash S)/NP : a \\ T : NP : b \\ F : NP\backslash S : a \circ b \\ T : NP\backslash S : a \circ b \\ T : S : c \\ F : NP : c \circ (a \circ b) \\ T : NP : c \circ (a \circ b) \end{array}$$

Although the full derivation (e.22) does not characterise a minimally closed tree (definition 3.8), the exhaustive application of LLKE's α and σ rules permit full type decomposition to be performed.

5.2.2 Normalisation via partial execution

The techniques proposed in (Moortgat, 1990b) to deal with spurious ambiguity are akin to those described above. The main difference is that, in the former, type derivations which might yield redundancy are, so to speak, "pre-compiled" into special deduction rules. Thus, a type such as the top formula of (e.22) in this method must be fully unfolded by repeated applications of (/L) and/or (L\) before the proper theorem proving task starts. A possible way to do this is by postulating the existence of sequences of types around the formula to be unfolded, constraining it to play the role of active type in a (L\) or (/L) deduction step. The appropriate steps are performed until the type has been totally analysed, and then the leaves matching the axiom scheme are pruned away along with intermediary nodes to generate what Moortgat calls *derived rules* — hence the allusion to pre-compilation. Likewise, cases where higher order types must be unfolded may require (R/) and (R\) to be used in the partial deduction phase. In fact, the pre-compilation steps amount to defining a partial ordering on the application of the original sequent rules.

A common criticism to this method is that proofs performed by the system which results from the derived rules fail to give a uniform and meaningful account of the logical structure of the types involved (Hepple, 1990), therefore contradicting the main motivation for using a Gentzen-style system in the first place. In addition, the methods developed in (König, 1989) and (Moortgat, 1990b) require considerable, extra-logical bookkeeping which doesn't seem totally justified in some cases[7]. Although no mention is made of how to perform efficient deduction in augmented substructural calculi and their linguistically properties in either approach, none of them seems to encompass the meta-theoretic ingredients needed to tackle this question. A more principled meta-theoretical approach is given in (Hepple, 1990).

[7] E.g. (Hepple, 1990) argues that Moortgat's pre-compilation phase can be dismissed.

5.2.3 Constructive and reductive normal forms

As in (König, 1989), in (Hepple, 1990) we find the development of two different systems of normal forms: one based upon properties defined inductively on the structure of proofs, and one based on a system of reductions, technically a rewrite system which operates on *derivation patterns*[8]. The former, which Hepple calls *constructive method*, underlies the parsing algorithm whereas the latter provides the apparatus necessary to prove the method's completeness and is called *proof reduction*.

Although directly related, König's and Hepple's approaches appear to have followed opposite directions on roughly the same path in order to attain proof normalisation: while König derived her parsing method from a meta-theoretical observation — the observation that proofs can be grouped into equivalence classes from which one is able to "reconstruct" normal proofs, Hepple starts off with a proof-theoretical notion grounded on linguistic principles of CG (Flynn, 1983), "headedness", to arrive at meta-theoretical properties by means of a system of rewrite rules.

Since the ultimate aim of proof normalisation is to eliminate semantically (in terms of the Curry-Howard correspondence) redundant proofs, the notion of a "proof head" was introduced in (Hepple, 1990) as the formal construct which seeks to express in purely syntactic — i.e. derivational, or *constructive* — terms the relation between the lambda semantics of the succeedent of a (product-free) L-entailment relation (the "meaning" of the proof) and types in the antecedent, down to the axiom leaves. On the semantic side the head of a proof is defined as the type in the antecedent which is labelled by the lambda term having widest scope over the lambda term in the succeedent. Since this condition is not verified for all proofs, some proofs are doomed to be "headless". It also turns out to be the case that for all headed proofs the antecedent of the proof's main branch axiom is always a subtype of the proof's head. It is interesting to compare this fact with our recoverability results (lemma 4.2 and proposition 4.7) to see LLKE deduction as an algorithm performing a sort of normalisation in a system where the cut rule has not been completely eliminated.

[8] We use the terms *proof/derivation pattern* to refer to subtrees consisting of 1 or more successive applications of sequent rules in a sequent proof.

Syntactically, Hepple defines an algorithm which maps cut-free proofs into subtypes of types occurring in them. The algorithm keeps track of the position of the head in the antecedent sequent by taking the axiom scheme as the base case (the head of an axiom instance is the entailment's antecedent, thus its position is 1) and recursively assigning a rank to subproofs ending with each of the sequent rules: (R/), (L/), (R\) and (L\). These essentially count the number of types which get added in next to the head of a subproof in the antecedent as subproofs are combined. Let's call subproofs above the deduction bar *child subproofs* and the ones below the deduction bar *mother (sub)proof*. For (R/) if the head count, say m, of the child subproof equals the number of types in the sequent on the left of the active type's subtype plus 1, then the head count for the whole proof, say n, is zero; otherwise $n = m$. For (R\) if $m = 0$ then $n = 0$, otherwise $n = m - 1$. For (L/), if the count m of the right-hand side subproof is greater than the number of types on the left of the active type's subtype in the same subproof plus one, then the count n for the whole proof is m plus the number of types in the antecedent of the left-hand child subproof; otherwise $n = m$. Analogously, in a proof ending in (L\), if m is greater than the number of types to the left of the active subtype on the right-hand child subproof, then n is m plus the number of types in the antecedent of the left-hand child subproof; otherwise $n = m$. Example (e.23) shows the algorithm in action to determine the head of a proof (compare the result with the labels for each type). The numbers on the right-hand side show the head count for the mother subproof (n) and the count for the relevant child (m).

$$\text{(e.23)}\quad \dfrac{\dfrac{C : h \vdash C : h \qquad \dfrac{B : f \vdash B : f \quad A : ghf \vdash A : ghf}{B : f, B\backslash A : gh \vdash A : ghf}\,(\backslash\text{L})}{B : f, B\backslash A/C : g, C : h \vdash A : ghf}\,(/\text{L}) \qquad m = 1, n = 2}{B : f, B\backslash A/C : g \vdash A/C : \lambda h.ghf}\,(/\text{R}) \qquad \begin{array}{l} m = 2, n = 2 \\ m = 2, n = 2 \end{array}$$

Normal proofs are defined constructively with respect to two main properties: headedness, as described above, and the occurrence of right inferences on the main child subproof. The base case in the definition of a a *constructive normal form* (CNF) is again the axiom scheme: it has no children and obviously does not contain any occurrence of right inferences. A proof ending in (L/) or (L\) whose left-hand child is in CNF and whose right-hand child is headed

by a subtype of the rule's active type exhibit no right inference on the main branch. All rules containing no right inferences on the main child are in CNF. Finally, all (R/) and (R\) subproofs whose children are in CNF are also in CNF.

This definition of CNF is implemented as a cut-free sequent through the following proof-theoretic refinements: (i) the original entailment relation is decomposed into relations $\vdash_1$ and $\vdash_2$ and (ii) the type which is required to be the head of any proof of a sequent is enclosed in a special concatenation operator, "+", a new rule is added which handles transitions between the two types of derivability relation. The resulting system of (Hepple, 1990) is shown in (5.3), below.

$$\begin{array}{lll} (2/1)\dfrac{\Delta + A + \Gamma \vdash_2 B}{\Delta, A, \Gamma \vdash_1 B} & (\text{L}/)\dfrac{\Delta \vdash_1 B \quad \Gamma + A + \Phi \vdash_2 C}{\Gamma + A/B + \Delta, \Phi \vdash_2 C} & (\text{R}/)\dfrac{\Gamma, B \vdash_1 A}{\Gamma \vdash_1 A/B} \\ (\text{Ax})\dfrac{}{e + A + e \vdash_2 A} & (\text{L}\backslash)\dfrac{\Delta \vdash_1 B \quad \Gamma + A + \Phi \vdash_2 C}{\Gamma, \Delta + B\backslash A + +\Phi \vdash_2 C} & (\text{R}\backslash)\dfrac{A, \Gamma \vdash_1 B}{\Gamma \vdash_1 A\backslash B} \end{array} \qquad (5.3)$$

This proof normalisation system is expected to meet the following requirements: (i) every proof should have a normal form and (ii) a proof in normal form is equivalent to the proof which it normalises. In order to do this (Hepple, 1990) introduces a second system in which properties (i) and (ii) are more easily proved and then shows that this second system is equivalent to the constructive one.

Firstly, a set of eighteen reductions (rewrite rules) is defined in which the redexes are proof patterns not in normal form. Then the set is shown to exhibit the property of *strong normalisation* — i.e. given a proof pattern X, either X is irreducible or it can be reduced in a finite number of steps to a pattern Y which doesn't match the redex of any rewrite rules. Standard term rewriting techniques are used in this proof (Dershowitz and Jouannaud, 1990). An arithmetic interpretation which maps proof patterns (redexes and contracta) into non-negative integers (*scores*) is provided. It is then shown that, under that arithmetic interpretation, for each rule the score of the redex is always greater than the score of the contractum. Since no negative scores exist, every sequence of reductions must necessarily be

finite. Strong normalisation corresponds to requirement (i).

It is proved that the reduction system meets requirement (ii) by showing that it obeys *strong confluence* — which is demonstrably equivalent to exhibiting the Church–Rosser property (Kirchner, 1994) — in addition to exhibiting strong normalisation. The Church–Rosser property says that if a term P reduces to terms A and B in a finite number of steps, then terms A and B can both be reduced to a term T. Now, it is easy to see that this property together with strong normalisation imply the existence of a unique normal form for any proof: if A and B are both in normal form, being therefore irreducible, then we must have that $A \equiv B$. Simple case analysis and induction on the structure of proofs suffice to show that the system of reductions yields exactly the same normal proofs as the constructive system.

5.2.4 LLKE labelled formulae and normal forms

We have seen that spurious ambiguity is a problem that arises in CCG due to the generality of the forward and backward composition rules (Eisner, 1996) and in Gentzen presentations of Lambek calculi due to the level of non-determinism found in standard sequent rules. Now, since non-determinism in LLKE gets reduced to the choice of which subformula of unfulfilled types to use in augmenting the tableau, it is difficult to situate proof normalisation in the context of the system. Firstly, spurious ambiguity is a practical concern only as far as it has a negative impact on a system's performance. In fact, this is the case for most combinatorial systems which work on a search regime of extensive enumeration of subproofs, as is the case of most rewrite-based parsing techniques for CCG and non-normalised sequent methods. In a logical framework this fact can be seen as the manifestation of the adverse side-effects of cut-elimination (Boolos, 1984), the price one generally has to pay for decidability. Complete elimination of cut deprives the proof theory of an explicit statement of the classical principle of *bivalence*, whereby either an assertion or its denial holds in a theory. In algorithmic terms, cut-elimination amounts to blocking subproof reuse. In other words, the use of lemmas is forbidden in cut-free calculi. LLKE reinstates the principle of bivalence through the θ rule[9].

[9]If a proof-tree is regarded as a model for a formula, then the θ rule can be seen as generating two submodels: one in which the assertion of the θ formula holds and other in which its denial is the case.

Even if the analytic restriction is obeyed and θ rules have to be applied to subtypes occurring in the proof tree, a certain degree of reasoning by lemmas is still allowed. In addition, tableaux in general and LLKE in particular can be seen semantically as model elimination systems. This fact combined with the presence of analytic cut guarantees that generating all relevant LLKE-models is not as costly as generating all relevant proof patterns in sequent systems — recall that the complexity results in section 4.4.2 are obtained under the assumption that the tableau is fully expanded. Therefore, efficiency does not appear to be the relevant criterion when assessing the consequences of proof redundancy in LLKE and similar systems.

We have already seen in section 4.1.1 that by allowing type-level re-bracketing we introduce in LLKE some spurious ambiguity of the kind discussed in (Eisner, 1996). This observation, which led to our rejection of the strategy of dealing with associativity at the syntactic level, in chapter 4, whereby the θ rule could be made redundant, reveals the contrast described above. Rules $\sigma_{(i)}$–$\sigma_{(iv)}$ may be regarded in practice as forms of function application. If they are allowed to operate across bracket boundaries at the level of syntactic types, then the purely combinatorial search regime of algorithm 3.2 allied to the associativity property of the labelling algebra ends up producing redundant derivations.

In section 5.2.1, we pointed out the resemblances between the (partial-tree) labelling technique of (König, 1989) and techniques for recovery of syntactic structure in LLKE. In fact, it appears that the labelling algebra of the latter, without type re-bracketing, encodes all redundant derivations which would have been generated if the tableau rules allowed full branching of σ_1-type formulae, as in conventional tableau systems, or type re-bracketing, as in the θ-free version of LLKE. If this is the case, then LLKE can be regarded as a normal-form system. This point is probably worth further analysis. However, since we have decided to focus LLKE on syntactic types and the effect of proof redundancy in the system's performance is quite distinct from the impact of spurious ambiguity on sequent based approaches, we leave this kind of comparison to future research.

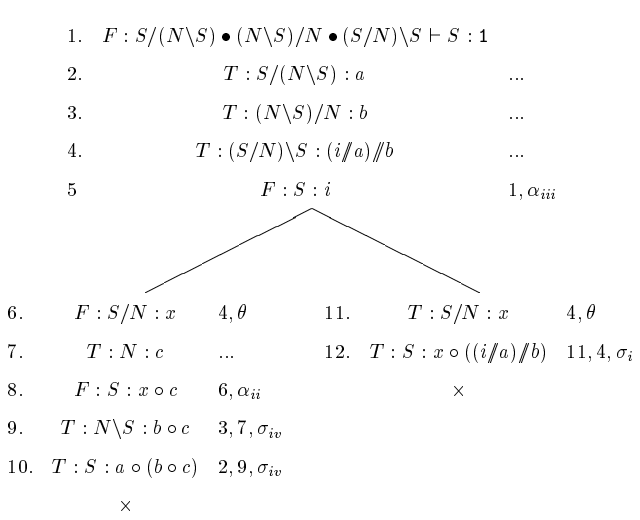
The processing of an example from (Hendriks, 1993) by LLKE is shown below in order to illustrate the non-occurrence of redundant derivations in the system and anticipate the starting points for further research in this area. Under a standard sequent calculus, the

parsing of the sentence of example (e.24) receives six derivations which are equivalent to only two distinct semantic readings[10].

(e.24) Someone loves everyone.

A minimally closing LLKE proof-tree for (e.24) is with the following lexeme–type correspondence: "someone" = $S/(N\backslash S)$, "loves" = $(N\backslash S)/N$, "everyone" = $(S/N)\backslash S$, is shown in e.25.

(e.25)



The proof-tree above closes with substitution $\varsigma = \{x \mapsto (a \circ b)\}$. Now, let's assume the assignment of lambda terms to each lexical entry in the antecedent — line 1 of example (e.25) — to be as shown in (e.26).

(e.26) $\{\text{"}someone\text{"} := \lambda P \exists x P(x),\ \text{"}loves\text{"} := loves,\ \text{"}everyone\text{"} := \lambda P \forall y P(y)\}$

The closure condition in the branch that recovers the structure of the sentence "Someone loves everyone" derives from lines 5 and 7. It is: $x \circ ((i/\!\!/a)/\!\!/b) \sqsubseteq i$. It can be easily verified that, even though (e.25) does not depict a fully-expanded tableau (since the type in line 3 is not fulfilled on the right-hand branch), no other satisfiable closure conditions would be derived if we had applied the θ to the unfulfilled formula. Applying substitution mapping ς to the closure constraint we obtain the solutions shown in (e.27) and (e.28). Notice that the properties of the labelling algebra (see chapter 3) which license each step are written on the right of each line.

(e.27)

$$
\begin{array}{lcll}
x \circ ((i/\!\!/a)/\!\!/b) & \sqsubseteq & (a \circ b) \circ ((i/\!\!/a)/\!\!/b) & \varsigma \\
 & \sqsubseteq & (a \circ (b \circ ((i/\!\!/a)/\!\!/b) & (3.6) \\
 & \sqsubseteq & (a \circ (i/\!\!/a) & (3.1) \\
 & \sqsubseteq & i & (3.1)
\end{array}
$$

(e.28)

$$
\begin{array}{lcll}
\ldots & \sqsubseteq & (a \circ b) \circ ((i/\!\!/a)/\!\!/b) & \varsigma \\
 & \sqsubseteq & (a \circ b) \circ (i/\!\!/(a \circ b)) & (3.4) \\
 & \sqsubseteq & i & (3.1)
\end{array}
$$

Under a type–semantics assignment similar to (e.26) (Hendriks, 1993) arrives at two different readings for (e.24): $\exists x \forall y.loves(y)(x)$ and $\forall y \exists x.loves(y)(x)$. In LLKE, if one interprets the σ rules as function application and α rules as lambda abstraction one sees that the alternative rewritings of the closure conditions, (e.27) and (e.28), suggest different application and abstraction schemes. These are granted by the two possible rebracketings labels expressions in (e.27) and (e.28) after variable instantiation. Another example along the same lines is given in appendix A, section A.4. There, however, the label expressions of derivation (e.49) are satisfied under a substitution $\varsigma = \{x \mapsto b \circ ((i/\!\!/a)/\!\!/b)\}$ in which the label variable *already* encodes one of the applications — namely the application of expression $((i/\!\!/a)/\!\!/b)$ to token b — which therefore rules out a second (and ambiguous) reading.

A full account of compositional semantics within LLKE would require an additional labelling scheme to be built on top of the existing system. This has not been attempted here. It is not clear how the other calculi of the substructural CG hierarchy of table 2.1 would behave under assignment of lambda terms. The issue of labelled deduction in categorial calculi as a whole poses complicated questions as to the proper foundations of the logical operations involved (Venema, 1996; MacCaull, 1997). Devising an adequate semantics to account for the the operational aspects of LLKE derivations appears to be a worthwhile task for further research in this context.

[10] The "tagged" sequent system of (Hendriks, 1993), which is demonstrably equivalent to Hepple's constructive calculus, succeeds in eliminating the spurious derivations.

5.2.5 Further remarks on proof normalisation

Proof normalisation was motivated by the need to develop efficient parsing algorithms for a logical grammar whose most perspicuous formal presentation is perhaps done by means of Gentzen sequents. Sequent calculi, however useful in the theoretical analysis of deduction, are not well-suited for automatic deduction, as shown, for instance by recent research in linear logic and proof nets (Girard, Lafont, and Regnier, 1995). From the point of view of analysing properties of proofs regarded as objects, the research on proof normalisation presents interesting and original contributions. Also, from a semantic perspective, the techniques developed in the works cited above provide evidence that the Curry-Howard isomorphism, a central feature of Lambek calculi with respect to (natural language) semantic interpretation, neither yields nor is a result of redundancy. Rather, redundancy is a characteristic of the nondeterministic presentation of the logic which can be curbed either by mapping proofs into equivalence classes, i.e. by performing proof normalisation, or by making nondeterminism – "parallelism" in the linear-logic parlance – explicit by means of graphs.

However, although it is the case that for a sequent calculus to be useful in parsing one must make sure that all and only normal derivations are produced, in practice it appears that there is no absolute need to "normalise" sequent proofs simply because there is no absolute need to *use* sequents. Many proof methods intrinsically more efficient than sequent calculi are well known in the theorem proving literature (Fitting, 1990), including resolution, tableaux, natural deduction and the variant of Bibel's *connection method* (Bibel, 1981) known as proof-nets (Girard, 1987). With Gabbay's work in labelled deductive systems (Gabbay, 1994), most of these systems can be extended to handle resource sensitivity. CG parsing seems to have followed this trend in substructural theorem provers in general, except where a choice was made to abandon the principles of the original Calculi in favour of more implementable variants [11].

[11] Another exception to the widespread use of labelling techniques is found in (Moortgat, 1994b) where it is show how the search regime devised in (Hepple, 1990) can be enforced by translation into an augmented language which includes modalities. Through the use of modalities an axiomatisation can be given which meets Hepple's requirements of non-redundancy and safeness. However, effective automated deduction procedures for the axiomatisation are not developed in the paper. A generate-and-test approach similar to the one adopted in (Moortgat, 1988) for an axiomatisation along the lines of (Zielonka, 1981) tends to suffer from the same efficiency problems.

A last comment, again with respect to the theoretical aspects of proof normalisation. The techniques used in (Hepple, 1990) and (Hendriks, 1993) provide a high level of abstraction over proofs[12]. It is however difficult to see how those techniques could "scale up" to cover other logics. An even more abstract framework seems to be required to fulfill this purpose. Perhaps coincidentally, *categorical proof-theory* (Lambek and Scott, 1988), which originated from a totally independent line of research, *category theory* (MacLane, 1971), may offer the required apparatus. Some work along these lines has been done in (Lambek, 1988) and (MacLane, 1982), but in general this field of investigation remains largely unexplored.

5.3 Branching in model-elimination systems and labelling

Semantic tableaux[13] may be regarded as model-theoretic developments of Gentzen sequent systems (Fitting, 1990). The idea which underlies the system is simple: the formula to be proved is negated and then a systematic search for counter-models for the negated formula is performed. The method is provably sound and complete with respect to standard (Boolean) interpretation for classical logic: in any finished systematic tableau, every open branch is simultaneously (first order) satisfiable (Smullyan, 1968). As in LLKE, tableaux get extended by means of rules according to the subformula principle. Smullyan distinguishes two main kinds of rules (for the propositional case), which he calls respectively α and β. Tableau rules for an implicational connective "/" are shown in (5.4) — the same signing conventions adopted in table 3.1 are adopted here. Analogous rules can be defined for the classical versions of "\" and "•".

$$\text{Rule } \alpha\text{:}\quad \frac{\alpha_1 \quad F : A/B}{\begin{array}{ll}\alpha_2 & T : B\\ \alpha_3 & F : A\end{array}} \qquad \text{Rule } \beta\text{:}\quad \frac{\beta_1 \quad T : A/B}{\beta_2 \quad F : B \;\mid\; \beta_3 \quad T : A} \tag{5.4}$$

[12] Recall for instance that one of the normal form systems presented in (Hepple, 1990) consists basically of rewriting on derivation trees.

[13] We sometimes refer to a standard (Smullyan-style) tableau simply as "tableaux" as opposed to non-standard tableaux such as LLKE.

The rules in (5.4) can be seen from a syntactic point of view as recipes for building Hintikka (or downward saturated) sets out of a initial set of classical propositional formulae, if interpreted algorithmically as follows: given a set $\mathcal{T}$ of formulae, if $\alpha_1 \in \mathcal{T}$ then add α_2 *and* α_3 to $\mathcal{T}$; if $\beta_1 \in \mathcal{T}$ then add β_2 *or* β_3 to $\mathcal{T}$. Although totally adequate for theorem proving in classical logic, these rules fail to provide the necessary level of granularity to deal with substructural calculi. The reason for this is that tableaux are simply sequent systems of which one has removed the sequents that provided the contextual surroundings for the active formulae. To see this, just read "upside-down" rules (R/′) and (L/′) in (5.5), removing the Greek letters and the derivability relation, "⊢".

$$\underset{(\mathrm{R}/')}{\frac{\Delta, T : A \vdash F : B}{F : B/A}} \qquad \underset{(\mathrm{L}/')}{\frac{\Gamma \vdash F : B \quad \Psi, T : A, \Phi \vdash F : C}{\Psi, T : A/B, \Gamma, \Phi \vdash F : C}} \tag{5.5}$$

The loss of contextual information in tableaux with respect to sequent systems is not an issue when tableaux are applied to classical logic. Classical deduction is monotonic and not sensitive to order or "quantity" of premises (in the sense that premises may be used as many times as one wants or not used at all). Lambek calculi, as we have seen, are non-monotonic and exhibit varied degrees of resource sensitivity. Therefore, to use tableaux in categorial deduction one has to be able to keep track of which types got used, how many times, and where in the derivation. Once again one could call Gabbay's labelling techniques (Gabbay, 1994; D'Agostino and Gabbay, 1994) into play. Let $\mathcal{L}^* = \langle \mathcal{P}, \circ, \mathbf{1}, \sqsubseteq \rangle$ be a labelling algebra defined as in definition 3.2. We can now redefine the rules in (5.4) as (5.6), where $\{a, b\} \subset \mathcal{P}$, b does not label any type occurring above it in the tableau and x is a label variable ranging over elements of $\mathcal{L}^*$.

$$\text{Rule } \alpha\text{:} \quad \frac{\alpha_1 \quad F : A/B : a}{\begin{array}{ll}\alpha_2 & T : B : b \\ \alpha_3 & F : A : a \circ b\end{array}} \qquad \text{Rule } \beta\text{:} \quad \frac{\beta_1 \qquad T : A/B : a}{\beta_2 \quad F : B : x \;\mid\; \beta_3 \quad T : A : a \circ x} \tag{5.6}$$

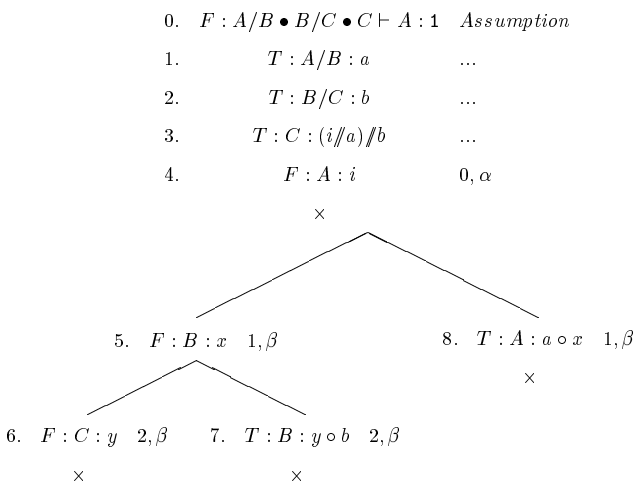
Rule α is the same as the corresponding LLKE rule. Rule β, which has no correspondent in LLKE, can be easily interpreted in model-theoretic terms: given that a type A/B is interpreted as having truth-value "T" in a database "a" (possible-world, multiset etc), then in order for it to be valid in model $\mathcal{L}^*$ there must be a database "x" such that either the subtype B does not hold in "x", or subtype "A" must hold in database "a" augmented with "x". The other operators of L can receive analogous treatment and the constraints on the labelling algebra which enable us to characterise different categorial calculi can remain the same as the ones defined in chapter 3.

5.3.1 Unification bottlenecks in automatic labelled deduction

If by means of straightforward type decomposition built on top of a labelling discipline standard tableaux can be adapted to handle exactly the same class of calculi handled by LLKE, then why do we bother to introduce rules which perform search on the tableau (the σ rules) in addition to a cut rule, which sequent meta-theory goes into so much trouble to prove redundant?

The answer is simply that the extra bookkeeping abilities gained via labelling don't come for free. In fact, most of the computation in labelled systems tend to be located in the label-checking modules — see appendix B for a typical execution profile of LLKE: notice that the cost of handling of linear (σ and α) and θ expansion is almost negligible if compared with the cost of closure checking. In systems where many branching rules are allowed such as standard tableaux, a large number of variables tend to be introduced, which often complicates the checking of label constraints — Not to mention the fact, discussed in (D'Agostino and Mondadori, 1994) that tableaux for propositional logic are inherently less efficient than systems which incorporate some sort of with analytic cut with respect to p-simulation (Cook and Reckhow, 1979). Even worse, arbitrary variable introduction forces one to solve all label expressions at the end of a full tableau expansion, which in some cases pushes the label decision problem into intractability (see table 4.1) or even undecidability. In other words, it is not possible in standard tableaux to keep a cache of intermediary variable bindings as described in section 4.4.1. Example (e.29) shows this.

(e.29)



Notice that a depth-first reading of the derivation tree does not impose an ordering on its set of closure constraints. The tableau closes with the substitution shown in (e.30).

(e.30) $\varsigma = \{y \mapsto ((i/\!\!/a)/\!\!/b), x \mapsto (b \circ ((i/\!\!/a)/\!\!/b))\}$

However, since $\langle (y \circ b), x\rangle \succcurlyeq \langle (((i/\!\!/a)/\!\!/b), y\rangle$ (recall partial ordering on closure pairs defined section 4.4.1, (4.26)) — each pair is composed respectively by the labelexps in lines 7, 5, 6 and 3 — label unification cannot be performed "on the fly", as the depth-first expansion of the proof tree is done[14]. Rather, unification has to wait until all expressions are made available, at step 8. Although we cannot rule out the existence of a non-trivial search regime which imposed the desired ordering on the closure constraints so as to allow label checking to be performed dynamically, we find it very unlikely.

Tableaux were originally designed to take advantage of certain aspects of classical logic which allow sequent deduction to be simplified. They do not provide the most adequate base for

[14]Note that a depth-first search on the proof tree will find the following: $\langle\langle (y \circ b), x\rangle, \langle (((i/\!\!/a)/\!\!/b), y\rangle, \langle (a \circ x), i\rangle\rangle$. The closure constraint derived from lines 7 and 5 cannot be added before the one in lines 6 and 3 because when variable x of line 5 is read there isn't enough information to compute its closing match (i.e. line 7 isn't yet available).

labelled deduction. Similarly, there seems to be little point in labelling sequent calculi for purposes of allowing generalised substructural deduction, since the rules already encompass elements which facilitate a very abstract level of resource control through separate *structural rules* — though labelling by lambda expressions has been fruitfully used in the analysis of *meaning*. There is, however, a system based on type decomposition similar to that performed in tableaux which has received a great deal of attention in CG theorem proving: proof nets. We discuss this kind of system in the following section.

5.3.2 Proof nets and higher-order linear logic programming

Originally developed for linear logic (Girard, 1987), proof nets are meant to provide a more faithful representation of the "parallelism" of computation, which according to Girard and others has been unduly hidden by the sequential structure of Gentzen-style deduction. In order to allow the mentioned parallelism to be encoded and at the same time control resource (formula) usage with respect to quantity and position, certain conditions on the connection of terminal nodes in the resulting graph — similar to those used in Bibel's "connection method" of theorem proving (Bibel, 1981) — are imposed.

To see (briefly) how proof nets work for the implicational fragment of linear logic, assume our set of operators to be $\mathcal{C} = \{\otimes, \multimap, \perp\}$ — where $\otimes$ stands for multiplicative conjunction, $\multimap$ for linear implication and $\perp$ is a form of negation — and let a denumerable set of atomic formulae $\{A, B, ...\}$ closed under $\mathcal{C}$ be the set of formulae of the fragment. In sequent terms, the operational rules describing $\otimes$ and $\multimap$ are the same as their counterparts in system (2.8): (R$\bullet$), (L$\bullet$), (R/) and (L/) respectively. The operational rules for $\perp$ are as stated in (5.7). Notice that the latter, which govern the polarity (negative or positive) of a formula as it moves from one side of the turnstyle to the other, describe explicitly the reasoning underlying the signed version of the sequent rules in (5.5) and hence tableau expansion rules such as (5.4).

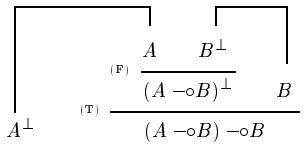
$$\frac{\Gamma \vdash A, \Delta}{\Gamma, A^{\perp} \vdash \Delta}\,(\mathrm{L}\perp) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A^{\perp}, \Delta}\,(\mathrm{R}\perp) \tag{5.7}$$

In addition to these operational rules, the implicational fragment of linear logic needs right and left versions of the permutation rule, (P), of (2.12). Proof-net construction in (Girard, 1987) is performed in two steps: (i) negation rules are used to move all formulae to one side of the turnstyle, where they are put in a *negation normal form* and (ii) rules (F) and (T) of (5.8) are used to "unfold" all formulae down to their atomic subformulae.

$$\text{(F)}\ \frac{A \quad B^{\perp}}{(A \multimap B)^{\perp}} \qquad\qquad \text{(T)}\ \frac{A^{\perp} \quad B}{A \multimap B} \tag{5.8}$$

After a complete proof net is built, the next phase which consists of linking up the atomic formulae at the graph's leaves has to be performed. Additionally, in order to represent valid derivations proof nets are required to obey graph conditions which reflect the substructural characteristics of the logic[15] and a "long trip condition", necessary to ensure that the right partitioning is achieved — i.e. that only atomic formulae meant to be in the same sequent subproofs as initial sequent axioms get connected. Example (e.31) shows a simple proof net for $A \vdash (A \multimap B) \multimap B$.

(e.31)



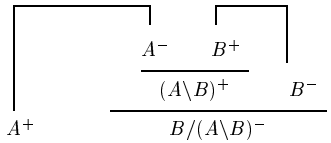
Roorda shows how to adapt proof nets to handle categorial calculi in (Roorda, 1991). The proof-net construction rules, shown in (5.9), are defined according to the same rationale behind the rules above. However, since the syntax of categorial logics excludes negation operators, a mechanism to express the full propositional truth-functionally must be introduced. We could use signs "T" and "F" as done so far. Instead we prefer to adhere to the proof-net tradition and use superscripts "+" and "−" to the same effect. Notice that all rules in (5.9) cause new subtrees to be adjoined the derivation tree.

[15] E.g. Links connecting the graph's leaves cannot cross, each leaf node must be connected to at most one other leaf of opposite polarity, etc.

$$\frac{B^{+} \quad A^{-}}{B/A^{+}} \qquad \frac{A^{-} \quad B^{+}}{A\backslash B^{+}} \qquad \frac{A^{+} \quad B^{-}}{B/A^{-}} \qquad \frac{B^{-} \quad A^{+}}{A\backslash B^{-}} \tag{5.9}$$

An example of application of the rules above is given in (e.32), for the derivation of the lifting theorem: $A \vdash B/(A\backslash B)$. Morrill (Morrill, 1995a) remarks that if the only restriction on proof-net connections were that their links must not cross, then it would be possible to prove invalid entailments such as the non-theorem: $B/(A\backslash B) \vdash A$ ("lowering").

(e.32)



In (Roorda, 1991; Hendriks, 1993), derivation of non-theorems of the form of the aforementioned "lowering rule", among others, is prevented by conditions encoded on the lambda terms which label each node. It is pointed out in (Hendriks, 1993) that the long trip condition alone does not stop spurious ambiguity from plaguing proof net systems, from which Hendriks seems to infer the need for these systems to undergo some sort of normalisation similar to that developed for sequent calculi.

We do not agree, however, that structural ambiguity in proof nets should be placed at the same level as spurious ambiguity in sequent calculi — at least as far as CG parsing is concerned — unless it impairs efficiency. This does not seem to take place in the systems described so far. A perhaps more relevant concern is raised in (Morrill, 1995a) with respect to the method's coverage. Morrill claims that although the proof-net theoremhood conditions suffice to characterise calculi such as L and LP, they are still not general enough. Calculi such as NL or "hybrid" systems like the ones described in (Hepple, 1995; Moortgat and Oehrle, 1993) find no obvious characterisation in terms of graph topology conditions alone. Therefore, an extra level of structural control is called for. Once again, labelling techniques seem to provide a convenient answer.

Morrill's proof nets

In (Morrill, 1995a) and (Morrill, 1995b), two systems of labelled deduction are introduced. Whether or not they should be called proof-net systems is debatable. First of all, in spite of the fact that they keep the basic skeleton of the unfolding rules (5.9) the topological conditions which distinguish among proof nets those which represent genuine proofs play a significantly less important role. These conditions serve only to recover the order-relevant information lost by expanding α-type formulae (in Smullyan's terminology) into two separate branches, instead of simply appending their subformulae to the branch being expanded. In view of this, the breed of "proof nets" described in Morrill's papers, along with those developed in (Moortgat, 1992; Moortgat, 1990a), might as well be regarded as a labelled variety of standard tableaux as described above.

It should be remarked, however, that the parallel drawn above refers mainly to the labelling rules for each method. We shall open a parenthesis here to discuss this. At the propositional (type) level, the redundancy exhibited by Smullyan tableaux, which in standard propositional logic is the very reason why tableaux cannot p-simulate (Cook and Reckhow, 1979) truth tables, gets eliminated if every formula causes a new pair of branches to be adjoined to the proof tree as in (5.9). In tableau systems, a list of formulae introduced by α-rules must be expanded as many times as the number of subtrees occurring below it which ends up causing certain proof trees to have a number of non-terminal nodes in $O(k!)$, where k is the number of distinct occurrences of propositional letters (atomic types) — see (Haken, 1985) for examples of tautologies whose proofs have exponential upper bounds and (D'Agostino, 1992; D'Agostino and Mondadori, 1994) for a comprehensive discussion of redundancy in tableau systems.

Fortunately, both proof nets and LLKE proofs have much lower upper bounds with respect to the size of their proof trees. The fact that standard tableaux exhibit an anomalous degree of topological redundancy while proof nets don't can be explained in terms of data structures. Consider a tableau proof tree such as (e.29). If we read each subtree as a set, as suggested in (Smullyan, 1968), then the data structure representing this tree looks like this (from left to right):

$$\begin{aligned}(\text{e.33}) \qquad \mathcal{T} = \langle\ & \{T : A/B : a, T : B/C : b, T : C : (i/\!\!/a)/\!\!/b, F : A : i, F : B : x, F : C : y\},\\ & \{T : A/B : a, T : B/C : b, T : C : (i/\!\!/a)/\!\!/b, F : A : i, F : B : x, T : B : y \circ b\},\\ & \{T : A/B : a, T : B/C : b, T : C : (i/\!\!/a)/\!\!/b, F : A : i, T : A : a \circ x\}\rangle\end{aligned}$$

Notice that the intersection of sets in example (e.33) yields a fairly large set. The common elements will have to be expanded into each subtree at each expansion step. Therefore, the more branching we have, the more redundancy is introduced. In proof nets, since all rules introduce branching, the data is "parallelised". A fully expanded proof net for the assumption of example (e.29), under the same data structure, will have at most 1 element in the intersection of all sets of formulae. In LLKE, topological redundancy is reduced (if not totally eliminated) by the fact that the system has a single branching rule which can be applied only after application to the other rules have been exhausted (D'Agostino and Mondadori, 1994).

Returning to Morrill's proof nets, the first system — discussed in (Morrill, 1995b) — uses a labelling algebra which can be easily interpreted as the complete lattice $\mathcal{L}^*$ used in section 5.3 to do the bookkeeping in standard tableaux[16]. Given $\mathcal{L}^*$, the system's expansion rules can be stated as in (5.10), where b is a new label token and x a new label variable ranging over information tokens.

$$\begin{array}{ll} 1.\ \dfrac{B^+ : (a \circ x) \quad A^- : x}{B/A^+ : a} & 2.\ \dfrac{A^- : x \quad B^+ : (x \circ a)}{A\backslash B^+ : a} \\[2ex] 3.\ \dfrac{A^+ : b \quad B^- : (a \circ b)}{B/A^- : a} & 4.\ \dfrac{B^- : (b \circ a) \quad A^+ : b}{A\backslash B^- : a} \end{array} \tag{5.10}$$

Following Smullyan's conventions (see rules (5.4), section 5.3), we shall refer to the types below the deduction bar in 1 and 2 as *β-types* and to the ones in rules 3 and 4 as α-types. For instance, notice that under the labelling regime defined in (5.10) our proof of lifting, (e.32), yields the following constraints: $a \sqsubseteq x$ (for the pair of leaves A^+ and A^- connected by the leftmost link) and $x \circ b \sqsubseteq a \circ b$ (pair B^+, B^-). These constraints are trivially satisfiable under

[16] The symbol "+" is used in (Morrill, 1995a) to denote the composition operator "$\circ$".

the substitution $\varsigma = \{x \mapsto a\}$.

The leaf connection condition still has to be enforced in this labelled version of (5.9). However, since nearly all structural control is transfered on to the labelling algebra, that constraint in practice amounts to checking for closure pairs, as in LLKE or standard tableaux. Straightforward inspection of labelling (5.10) and negation rules (5.7) — the latter having all types $X^{\perp}$ replaced by X^{-} and all types X replaced by X^{+} — shows that in a typical product-free categorial proof for $X_1, ..., X_m \vdash Y$, the number of label variables to be introduced is $O(n)$, where $n = \sum_{i=1}^{m} dg(X_i)$ is the sum of the degrees of the formulae in the antecedent: assume each X_i to have the form $(...(X_{i_1}/X_{i_2})/.../X_{i_r})$ — i.e. a β-type with left-associative bracketing — and reason inductively noticing that the unfoldings for each X_{i_j} introduce a new variable plus a new β-type with the same structure as X_i and so on. This represents a large number of variables to be instantiated in a potentially large number of semigroup (groupoid etc) equations to be solved *only* at the end of the unfolding phase. The reason for this is the same as the reason why variable instantiation cannot be performed dynamically in the tableaux discussed in section 5.3.1: the lack of an unfolding algorithm capable of ordering label constraints appropriately as soon as they are introduced. The label-checking problem in this kind of method tends therefore to reside in the intractable domain of A-unification (AC-unification etc), as described above.

The second system of (Morrill, 1995a) addresses this problem by adding a mechanism for compilation of label constraints into higher-order logic programming clauses (Hodas and Miller, 1994). The objective here is not to reduce the number of variables or to perform dynamic constraint solving but to re-organise them into *clauses* so that unification gets restricted to one-way matching which is then implemented via SLD resolution techniques (Siekmann, 1989). The starting point is a semantic analysis based on the labelling algebra, similar to the semantics presented in chapter 3. Conditions (5.11) define a validity relation directly derived from the groupoid interpretation of L — see (2.11) and (2.10) in chapter 2.

$$\begin{aligned} a \models A/B \quad &\text{iff} \quad \forall x(x \models A \Rightarrow a \circ x \models B) \\ a \models A\backslash B \quad &\text{iff} \quad \forall x(x \models A \Rightarrow x \circ a \models B) \end{aligned} \tag{5.11}$$

Now, if we review rules (5.10) in the light of the type interpretation above we conclude that the new tokens ("b") introduced by rules (5.10.3) and (5.10.4) are in fact Skolem constants while the label variables ("x") in (5.10.1) and (5.10.2) correspond to first-order metavariables. It turns out that labelled proof-net rules can be reordered so as to reflect the implications of (5.11) thus yielding expressions in (higher-order) logic programming clausal form. Since the resulting implication is sensitive to the occurrence of tokens, it must be characterised as linear implication, as shown in (5.12).

$$\begin{array}{ll} \text{1.} \ \dfrac{B^+ : (a \circ x) \quad \circ\!\!- \quad A^- : x}{B/A^+ : a} & \text{2.} \ \dfrac{B^+ : (x \circ a) \quad \circ\!\!- \quad A^- : x}{A\backslash B^+ : a} \\ \text{3.} \ \dfrac{B^- : (a \circ b) \quad \circ\!\!- \quad A^+ : b}{B/A^- : a} & \text{4.} \ \dfrac{B^- : (b \circ a) \quad \circ\!\!- \quad A^+ : b}{A\backslash B^- : a} \end{array} \tag{5.12}$$

Now, instead of checking for planar connections or "long trip conditions" one only has to unfold the type completely, concatenate the implications at the terminal nodes of the resulting graph, convert the resulting expression into "uncurried" form — i.e. convert its subexpressions of the form $(...(X^+ \circ\!\!- Y_1^-) \circ\!\!- ...) \circ\!\!- Y_n^-$ into clauses $X^+ \circ\!\!- Y_1^- \otimes ... \otimes Y_n^-$ — and solve it by assuming the leftmost implications to be a *goal* and the remaining subexpressions to be an *agenda*, in a logic programming database. Now, let an agenda be a $\otimes$-concatenation of goals, a goal either an atomic type or a clause of the form $X^+ \circ\!\!- Y_1^- \otimes ... \otimes Y_n^-$, and the program database Δ a multiset of clauses[17]. The linear logic programming theorem proving may then be defined by axiom $A \vdash A$ plus the following sequent rules:

$$\begin{array}{ll} {\scriptstyle (\text{Rs})} \dfrac{\Gamma \vdash B_1 \otimes ... \otimes Bn \otimes C_1 \otimes ... \otimes C_m}{\Gamma, A \circ\!\!- B_1 \otimes ... \otimes B_n \vdash A \otimes C_1 \otimes ... \otimes C_m} & {\scriptstyle (\text{Dt})} \dfrac{\Gamma, B \vdash A \quad \Delta \vdash C_1 \otimes ... \otimes C_m}{\Gamma, \Delta \vdash (A \circ\!\!- B) \otimes C_1 \otimes ... \otimes C_m} \end{array} \tag{5.13}$$

In (5.13), rule (Rs) represents a step of SLD-resolution and (Dt) a version of the deduction

[17] The definition of *goal* here is what characterises the system as higher-order logic programming, as opposed to classical logic programming where goals can only be atomic formulae or predicates. The identification of databases with a multisets is added to comply with this feature.

theorem which permits proof by search of subgoals. Example (e.34) shows a logic programming proof net for lifting.

$$\text{(e.34)}\qquad \dfrac{A^+ : a \qquad \dfrac{B^- : (a \circ b) \;\; \circ\!\!- \;\; \dfrac{B^- : (x \circ b) \;\; \circ\!\!- \;\; A^+ : x}{(A\backslash B)^+ : b}}{B/(A\backslash B)^- : a}$$

Example (e.35) depicts a linear logic programming verification of the theoremhood conditions of (e.34), translated from the notation used in (Morrill, 1995a) back into the sequent notation for the sake of clarity.

$$\text{(e.35)}\qquad \dfrac{\dfrac{\dfrac{}{a \vdash a}\,(\text{Rs})}{a, ((x \circ b) \circ\!\!- x)[x \mapsto a] \vdash a \circ b} \qquad \emptyset \vdash \emptyset}{a, \emptyset \vdash (a \circ b) \circ\!\!- ((x \circ b) \circ\!\!- x)}\,(\text{Dt})$$

The crucial step in (e.35) is the application of of substitution $\{x \mapsto a\}$ to the label of the topmost B^-, which guarantees that the axiom scheme gets the correct instantiation. Morrill-style proof nets use the polarity signing system of standard proof nets only to control which rule to apply at each level of unfolding. There is no need whatsoever to connect nodes of opposite polarity or to perform graph search of any sort. The system has been adapted in (Morrill, 1995b) to work with relational frames (van Benthem, 1991) rather than groupoid models. This appears to have no significant influence on the overall characteristics of the method.

We have seen above that a problem of redundancy arises when two distinct treatments of substructural deduction, proof nets and labelled deductive systems, are carelessly combined: in addition to having to connect nodes of opposite polarities according to the topological constraints that characterise each calculus, one still has to solve systems of label equations which tend to reside in intractable unification classes. Morrill's parsing methods have the merit of correcting this distortion by providing a uniform method of compilation of label expressions into higher-order logic programming clauses. However, by limiting theoremhood

conditions to label clauses one loses what is arguably the most attractive feature of proof nets, i.e. the possibility of using algorithms derived from graph theory in theorem proving. The system that results may without loss of generality be characterised as a modified form of tableaux. We have seen that Morrill's proposal improves on efficiency with respect to Roorda/Moortgat's proof nets (Moortgat, 1990a; Roorda, 1991) in that it limits the unification task to cases where one term is always ground. This certainly facilitates implementation but does not eliminate the need to perform full type decomposition (unfolding) before the next stage, label checking, is initiated, thus restricting the system's computational possibilities to a serial processing model.

5.4 Further issues: Scalability, modularity, heuristics

Most works on automated CG deduction mention generality (coverage over a wide range of substructural calculi) and efficiency as the main requirements to be met by CG theorem provers. In view of the issues discussed in this chapter these requirements can be stated in a more specific way. First of all, high generality is of very little use if the overall complexity of the system is hopelessly intractable. A more reasonable approach seems to be to start with a less expensive system — recall, for instance, that the first attempts to tackle efficiency issues in Lambek calculi (Hepple, 1990; König, 1989) focused on product-free subsets — and build on top of it extensions compatible with the efficiency-boosting techniques developed for the initial calculus. We call a CG system's ability to incorporate new characteristics which increase the expressivity of (or the range of language phenomena addressed by) its calculus *scalability*. We could probably distinguish two different approaches to scalability:

1. One in which the expressivity of the system is increased via strictly logical features within a well-defined class of logics for which Gentzen presentations exist and associated algebraic or model-theoretic semantics can be defined (i.e. those ranging from NL to LPCE as shown in table 2.1), and

2. a more implementation-oriented formulation in which types of a non-monadic nature — for instance, those approaches incorporating feature-structure systems — are added to

a base Lambek calculus.

The latter is best represented by systems along the lines of König's *LexGram* (König, 1995), which has its roots in an attempt to solve proof-theoretic shortcomings of L's sequent apparatus (König, 1991). The former encompasses more ambitious systematisations of early work on the Lambek hierarchy of structural calculi, such as structural modalities (Hepple, 1990; Morrill et al., 1990; Versmissen, 1994) and bidirectional division operators, guided by "minimalist" principles of linear order, constituency and dependency. These include the lattice of dependency calculi in (Kurtonina and Moortgat, 1995), Morrill's *type logical grammar* (Morrill, 1994) and hybrid categorial logics of (Hepple, 1995). The strategies developed for LLKE are aimed at automating deduction in systems of this (former) kind.

After the initial work in proof normalisation, little has been done on techniques to improve sequent deduction which preserve scalability. Rather, sequent deduction seems to have been limited to meta-theoretical work while effective theorem proving devices have been sought elsewhere — witness this the research on proof nets (Roorda, 1991; Moortgat, 1992; Morrill, 1995a). We have seen, however, that standard proof nets alone are insufficient to cover all aspects of substructural deduction required by CG. Labelling has been used to enhance generality (Moortgat, 1992) but it introduces redundancy along with efficiency problems of its own, failing therefore to meet our scalability criteria. The situation improves in (Morrill, 1995a; Morrill, 1995b) with the association of the labelling regime with a strategy for label-checking in a linear logic programming framework. Although the method is shown to be relatively efficient for NL and L, no indication is given of how the system would behave under more powerful calculi. The algebraic component obviously supports extensions, as it does in LLKE. Further research seems to be needed in the logic programming side, however. In LLKE, uniform strategies are employed across different calculi, which we believe provide a better answer to the scalability requirement.

Also related to this generality requirement, we can distinguish a method's ability to deal with different aspects of deduction (e.g. logical rules vs. algebraic constraints) more or less independently, in *modules*, in such a way that developments in one *module* can be easily incorporated to the system as a whole. We call this *modularity*. For instance, it is hard to see

how proof normalisation systems could be modularised. On the other hand, Morrill's proof nets present good modularity because independent improvements on the logic programming module tend to affect the system's performance immediately. LLKE also exhibits a high degree of modularity. In addition, the more uniform balance between the work load for each module achieved in LLKE, along with the fact that computations performed on separate branches of LLKE can be treated dynamically by the labelling module, makes the algorithm amenable to parallelisation — which as we have seen doesn't occur in labelled proof nets, since their search regime effectively collapses into a tableau-like search regime. A parallel version of algorithm 3.3 could, for instance, expand left and right branches simultaneously, with label closure tests on the left branch being prioritised (since as shown in section 4.4.1 a depth first search on a LLKE tree yields an ordering of label expressions from the most to the least general, with respect to variable instantiation). This appears to be a promising direction for future research.

Finally, a central issue in automated deduction which seems to have been almost neglected by most research in CG[18]: a system's ability to incorporate domain-dependent *heuristics*. The situation seems even more paradoxical if one considers that what would be called *heuristics* in standard theorem provers is, in CG parsing, nothing but linguistic knowledge such as word-order, dependency, etc[19]. Theorem provers with analytic cut provide plenty of room for heuristics (D'Agostino and Mondadori, 1994). LLKE's σ rules consist essentially of types *searching* on a string for other specific types with which they can "combine". It's easy to see that order-relevant knowledge, for instance, can be easily added to the linear expansion algorithm. Similar freedom is enjoyed by parsing systems based on natural deduction (Prawitz, 1965) such as the ones in (König, 1991; König, 1995). In the latter, that flexibility is exploited by re-arranging the logical algorithm in different natural-language parsing schemes such as bottom-up, top-down, shift-reduce, and chart parsing. LLKE enables the same sort of tabulation techniques to be used in a framework which preserves the *logicalist* approach to linguistic description and parsing envisaged in (Morrill, 1994; Hepple, 1995; Kurtonina and

[18] With the possible exception of (König, 1991) which later developed into a practical tool for grammar specification.

[19] One could speculate whether it is evidence against the "parsing as deduction" paradigm that a substantial part of the phenomenon which it seeks to model should be encoded as heuristics. We believe however that this is not a problem with the paradigm but a matter to be settled by (proof/language) engineering techniques.

Moortgat, 1995). Again, the work presented here is merely a first step in this direction.

A last point worth mentioning concerns the display of proofs. Most approaches discussed in the last two chapters implicitly assume that theorem proving in CG should reflect the linguistic structure of the types being parsed. Proof-normalisation can be regarded as an effort to reduce the output of a sequent system to derivations which contribute "useful" information, pruning away the ones which display redundant structures. In proof net systems lambda encoding is employed in keeping track of the relevant semantic information, the proof graph itself being of little importance. Likewise, the topology of LLKE trees is quite uninformative in this respect, though the recoverability results of chapter 4 show that the appropriate information can be extracted from it.

5.5 Summary

In this chapter we have reviewed strategies for improving efficiency in automated CG deduction, starting with proof normalisation in sequent systems, covering natural deduction and arriving at proof nets.

From the perspective of sequent calculi the analysis developed in this chapter focused on proof structure rather than semantic labelling. Although no strategy for labelling derivation steps with lambda expressions has been developed here for LLKE, a functional interpretation of its algebraic label expressions yield results which tend to agree with those obtained in proof normalisation. It would be interesting to equip LLKE with a lambda semantics mechanism to investigate such convergence is verified.

In the second half of the chapter, the labelled proof net framework is recast in terms of labelling in a standard tableau system. A few shortcomings of branching systems with respect to variable introduction and label-checking are pointed out. Finally, parallels between the strategies presented in chapters 3 and 4 of this thesis and the ones used by the systems mentioned above are drawn and a set of requirements for CG automated deduction systems is sketched.

Chapter 6

Further issues: Polymorphism and Information Flow

In this chapter we address two extensions of the framework presented so far: quantification — or the handling of polymorphic types in CG — and the more speculative matter of theorem proving in *information networks* of (Barwise, Gabbay, and Hartonas, 1994; Barwise, Gabbay, and Hartonas, 1995).

The first of these extensions concerns practical issues in computational linguistics and parsing. Some kind of treatment of polymorphism seems to be necessary if a system implementing a lexicalised formalism is to provide the adequate level of generality for grammar specification. The tableau rules to be defined below address this problem by borrowing a few tools from *abstract quantification theory* (Smullyan, 1969). This is to be regarded as a first step towards a more comprehensive treatment within analytic deduction rather than finished work.

In the remaining of the chapter, we shift the focus towards automated reasoning in artificial intelligence. We discuss Barwise and coworkers' formalisation of a non-classical logic which we believe can be supported by analytic tableau deductive mechanisms. We point out connections with issues in planning and agent-oriented theories. The basic idea is to discuss the computational properties of theorem proving in a restricted *information network* setting

within the LLKE system. In order to do this, we briefly outline the language of types proposed in (Barwise, Gabbay, and Hartonas, 1995) to classify *sites* and information channels in their *logic of information flow*, and then describe CG parsing as an operationalisation of a fragment of this language. Finally, we show how a canonical model of information-oriented logics can be directly characterised in LLKE.

6.1 Polymorphic types

We mentioned in chapter 2 that one way to deal with lexical items which have different though analogous functions — see for instance the prepositional phrases of section 2.3.5 which can play either adnominal or adverbial roles, and coordinator types which can coordinate different constituents — is to generalise over a limited number of functions which linguistic analysis indicates as being the ones such lexical items are likely to play — e.g. (Morrill, 1994, chapter 6). This approach works by augmenting the language of CG with connectives to express the *collapsing* of a finite number of elements into a single type. A more general approach is adopted in (Emms, 1994) with the introduction of quantifiers into the logical language. The former approach requires addition of new deduction rules and more detailed, complex specifications of lexical types by the user in order to keep the logic within propositional limits while dealing with the generalisations. The latter, on the other hand allows for a greater degree of underspecification but seems to provide much more generality than grammar specification requires, easily leading to incompleteness on the logical side.

The approach described below can be seen as a moderate alternative to full quantification. It is aimed at automation, in keeping with the spirit of the previous chapters of this thesis. Polymorphic types will be assumed to be always under the scope of a universal quantifier which will be allowed to range over a well-defined set of values: the variable-free types of the language in definition 2.1. We present an extension of LLKE — which we call LLKE^v — to deal with the augmented logics. The resulting system, unlike the original LLKE of chapter 3, does not provide an effective method for deciding theoremhood in all cases, since variables will range over a denumerably infinite set of types — recall *lifting* (2.5) and *division* (2.6)

(2.7) to have an idea of where infinity shows up in Lambek calculi — which may yield infinite (open) branches. However, the procedure is *mathematically well defined* (Fitting, 1990) and can be constrained into termination in (we expect) reasonable ways — see, for instance, the Scönfinkel-Bernays class of quantified formulae which has been used in (Johnson, 1991). From a linguistic viewpoint, working with unconstrained domains of quantification may also lead to overgeneration — i.e. a polymorphic type assuming anomalous functions due to inappropriate variable instantiation — a problem which is avoided in Morrill's propositional approach. In the following sections we discuss some linguistic examples of polymorphism presented in (Morrill, 1994) and their treatment in LLKE^v.

6.1.1 Foreword on quantification

Let's start by enriching our language of syntactic types with variables and a (restricted) form of quantification. In order to do so we extend the set of well-formed types, $\mathcal{C}$ (definition 2.1), to a set $\mathcal{C}^v$ as shown in definition 6.13. Unlike first-order languages, the syntax defined below does not provide explicitly for universal or existential quantification. Rather, all variables appearing in a product-free type will be assumed to be under the scope of a universal quantifier. A form of existential quantification will take place where the sign of the SLF in which a type variable occurs is "F" — meaning "it is not the case that for all types" etc. Given the restriction imposed by (4.21), such existentially quantified types must appear only on the right-hand side of a sequent. Therefore both universal statements — those where a universal quantifier has scope over the whole expression — and existential quantification are unlikely to play any major role in grammar specification.

However, the language of definition 6.13 is expressive enough to allow some general *properties of grammars* to be enunciated — e.g. that a given string (or a string of a given structure in case it contains polymorphic types) yields a type of a certain structure, where the "certain structure" is left underspecified through polymorphism — see examples (e.36)-(e.38). The status of such systems is not clear in polymorphic CGs. Although we will not pursue this direction of research here, we will discuss the status of universal statements in the *metatheory* of CG in section 6.1.2.

Definition 6.13 *Let $\mathcal{P} = \{A, B, C, N, NP, AP, PP, ...\}$, as in definition 2.1, and define a set $\mathcal{P}^v = \mathcal{P} \cup \{x, y, z, ...\}$ (with or without subscripts), our augmented set of* basic types. *The extended type language $\mathcal{C}^v$ is the closure of $\mathcal{P}^v$ under operators $\mathcal{O}p = \{/, \backslash, \bullet\}$.*

Our next step is to define a more concise way of talking about substitutions, this time not of label variables as in chapter 4 but of variables ranging over syntactic types. Let φ^x_A represent the result of substituting A for x in φ according to the inductive definition 6.14. We shall assume here that the (universally quantified) variables of $\mathcal{C}^v$ range over the set of variable-free types $\mathcal{C}$. Furthermore, we assume that quantifier scope does not extend across product operator ("$\bullet$") boundaries. For instance, a polymorphic type "$(x/x)\backslash x \bullet x/x$" may be equivalently written as "$(y/y)\backslash y \bullet x/x$". Clause (iv) of definition 6.14 is meant to deal with variable replacement under these scope assumptions. The idea behind the substitution operation in proof search is simple. In proving entailments between sequences of types (i.e. sequents) one assumes the type in the antecedent to be a T-type, the type in the succeedent to be an F-type and tries to find appropriate substitutions so that the tree built from these assumptions closes under the label closure conditions defined in chapter 3, showing therefore the unsatisfiability of the assumption. Informally, the model we have in mind is one in which type variables get mapped into variable-free types which then are interpreted against our *information tokens* of definition 3.2.

Definition 6.14 *A substitution φ^x_A results in the following:*

(i) $\varphi^x_A = \varphi$ *if* φ *is an atomic constant,* $\varphi^x_A = A$ *if* φ *is an atomic variable.*

(ii) $[\varphi/\phi]^x_A = \varphi^x_A/\phi^x_A$

(iii) $[\varphi\backslash\phi]^x_A = \varphi^x_A\backslash\phi^x_A$

(iv) $[\varphi \bullet \phi]^x_A = \varphi^x_A \bullet \phi$ *if* x *occurs in* φ, $\varphi \bullet \phi^x_A$ *otherwise.*

If we assume that in grammatical specification no universally quantified statements are made, as far as the semantics of quantification is concerned we shall be mostly interested in *satisfiability* (with respect to a universe of discourse anticipated to be the the set of well-formed, variable-free types) rather than *validity* (in the sense of *first-order validity*). Once the replacements are made, LLKE-validity with respect to the information frames $\mathcal{L}$ can be tested through algorithm 3.3, for the variable-free case. The semantic basis of these notions is stated in the following section.

It should be remarked, however, that the assumption above — viz. that quantification should be circumscribed to product-free types in grammar specification — is not an empirical fact but rather a working hypothesis which follows the practice adopted in some Lambek systems. There are categorial approaches to grammar specification defined in terms of principles (statements about *all* possible configurations) as well as particular instantiation of types and inference rules (Moens et al., 1989; Calder, Klein, and Zeevat, 1988). By circumscribing the scope of our implicit quantifiers as above (and by keeping them implicit instead of expressing them in the language of types) we choose to stipulate that grammatical principles are implicitly determined by properties of the entailment relation for each system. These are derived, as we have seen, from certain structural features of deduction in general, and often get imported into the grammar specification language in the form of modalities (Morrill, 1994) and hybrid type constructors (Moortgat and Oehrle, 1993; Hepple, 1995). At the level of the tableau expansion rules, however, it should be a straightforward exercise to alter the deductive machinery to cope with the specification of universal principles.

6.1.2 Valuations and universal statements

In standard first-order logic one normally distinguishes between *constants* and *parameters* when defining an *interpretation function*, or *valuation*, for the formulae. Constants are the syntactic counterparts of the semantic elements in a *universe of discourse*, $\mathcal{U}$. First-order variables range over such elements. Parameters are symbols used to instantiate variables with hypothetical individuals (which might turn out to have correspondents in the universe of discourse such that the formula in which they occur is satisfied with respect to that universe under a given interpretation). This distinction is probably meant to reflect the kind of reasoning performed by a mathematician who in the course of a proof, having already shown that a certain property P holds for some individuals x, says "let a be such an x" and carries on with $P(a)$ as part of his assumptions, to be confirmed or rejected later. This newly

introduced symbol, committed to predicate P, is what is called a *parameter* in (Smullyan, 1968).

A valuation is initially defined on the set of parameter-free, *closed formulae* — i.e. those containing no free variables. A universally quantified formula is *true* under a valuation iff *all* instances of its predicates in $\mathcal{U}$ are true under a *Boolean* (propositional) valuation. Similarly, an existentially quantified formula is true under a valuation iff *at least one* element of the universe of discourse makes the sentence true under a Boolean valuation. One could also define an *interpretation function* by specifying n-place relations in $\mathcal{U}$ into which the language's n-place relations would be mapped. This interpretation function may then be extended in the following way: all parameters occurring in a sentence are replaced by constants in $\mathcal{U}$ and the resulting (closed, parameter-free) sentence is evaluated as usual. An interpretation function so defined would be equivalent to a valuation for quantified formulae (Fitting, 1990). If a set of sentences with parameters is such that its parameters can be renamed in such a way that there is a substitution (of constants for parameters) which makes each sentence in the resulting set "true" (under a given interpretation, in a given universe), then we say that this set of formulae is *simultaneously satisfiable*.

Our approach to quantification as a way to deal with polymorphism in CGs requires only a fairly simplified treatment. First of all, the the type syntax does not involve the use of distinct predicates as in first-order logic. All atomic types may be assumed, so to speak, to be arguments of a hidden unary predicate "occurs" which indicates the occurrence of a string in positions and number to be specified through logical operators according to the structural properties of the target calculus.

As we mentioned before, a polymorphic type is to be interpreted as being under the scope of a universal quantifier, and universally quantified variables do not seem to play any relevant role in grammar specification. Universal quantification does, however, play a relevant role in the *metatheory* of CG. We shall have a brief excursion into this issue before proceeding with grammatical polymorphism proper. Let's, for the time being, forget the conventions on variables and quantification in polymorphic types introduced above and consider the hypothesis of yet again augmenting our language of types $\mathcal{C}^v$ with a syntactic counterpart for

"$\vdash$", say "$\Rightarrow$" (e.g. via application of a deduction theorem), and the usual quantifiers "$\exists$" and "$\forall$". Examples of *universal statements* in this augmented language are shown in (e.36)–(e.38). These rules, or "inference schemes" appear frequently in the CG literature — see for instance the introductory chapters in (Moortgat, 1988; van Benthem, 1991) and chapter 2 of this thesis. The universal quantifiers, however, are taken for granted in most textbooks and left out, hence the presumed interpretation of these rules as "templates" which get instantiated in the deductive process.

(e.36) $\forall xy[x/y \bullet y \Rightarrow x]$ (application)

(e.37) $\forall xy[x \Rightarrow (y/x)\backslash y]$ (lifting)

(e.38) $\forall xyz[x/y \Rightarrow (x/z)/(y/z)]$ (division, main functor)

In fact, it is now clear that our LLKE-proof of proposition 3.2 is strictly speaking a proof of *instances* of the so called "reduction laws" rather than a proof of the laws themselves. The reason why the proof can be generalised to *all* instances is that we are allowed to reason with *parameters*, in the sense explained above. LLKE proofs are proofs by contradiction: the negation of the formula to be proved is assumed to be the case and a contradiction is sought which arises from this assumption. Therefore, in proving (e.36) for example we assume that there are values of x and y for which $(x/y \bullet y \Rightarrow x)$ is false. Then we say: "let X and Y be such values" and continue with the proof, as shown in chapter 3.

It should be remarked that most categorial calculi *already* provide for a restricted form of polymorphism within propositional logic. To see this consider the *lifting* and *division* rules of chapter 2 — respectively (2.5) and (2.6), (2.7). The former can be seen as generalisations over expressions of the form $...y/X\backslash y...$ and the latter over types of the form $(X/z_1/.../z_n)/(Y/z_1/.../z_n)$, where $n \geq 0$, and similarly for "$\backslash$". In addition to this inherent generalising ability, we want to provide our calculi with what (Moortgat, 1988) calls "basic type polymorphism". This is the type of polymorphism exhibited by the word "and", for instance, which can coordinate constituents of different form, such as sentences, verb phrases, and noun phrases.

At this point we close the parenthesis on universal statements and take for granted the kind of generalisation discussed in the last paragraph to return to our restricted syntax of universally quantified types confined within product boundaries. We start by specifying the notion of *satisfiability* within our labelled deductive system through definition 6.15. It should be remarked that a level of indirection is introduced with this definition of satisfiability: recall that LLKE is shown to be sound and complete in chapter 3 *with respect to a Gentzen presentation* rather than a semigroup (groupoid, relational etc) semantics. Likewise, quantifiers in LLKE^v will not be evaluated directly with respect to an algebraic structure — our information frames, which are in fact mere bookkeeping devices — but with respect to a universe of discourse composed by the types in $\mathcal{C}$.

Definition 6.15 *Let $\mathcal{L}$ be the labelling algebra (information frame) of definition 3.2, and let a be an information token in $\mathcal{L}$. We say that for a type $\varphi \in \mathcal{C}^v$:*

(i) $a \models \varphi$ iff for each x occurring in φ and each type $A \in \mathcal{C}$ $a \models \varphi^x_A$

Since we have stipulated that no wide-scope universal statements are allowed in the logic, this notion of satisfiability will suffice for our purposes. Given a sequent containing variables on the left-hand side of the turnstyle, we will be interested in finding types which instantiate these variables in such a way that the entailment holds in the given model (or *database*, to use the LDS parlance). This differs from the approach adopted in (Emms, 1990), for instance. We won't discuss this approach in detail here, but it appears that (Emms, 1990) aimed at expressing rather more general properties than the ones to which its quantificational logic is applied in the paper. These further motivations are more clearly spelled out in (Emms, 1994), where a full treatment of quantification is presented along with a string semantics. The linguistic drive behind the treatment described in the former paper, however, can be dealt with if the types are assumed to be quantified as above. The notion of satisfiability for quantified formulae in (Emms, 1994) is tied directly to the (semigroup) semantics against which the satisfiability of a variable-free type is interpreted. Emms' semantic formulation of polymorphism has the advantage of being more uniform than the one in definition 6.15. Our approach introduces an extra level of analysis (from polymorphic types to quantifier-free types and from quantifier-free types to algebraic models) which amounts to a somewhat less

elegant formalisation. From a practical point of view, however, this loss of elegance pays off by enabling us to define a straightforward proof search mechanism based on standard tableau and resolution techniques. We address these techniques in the next section.

6.1.3 Tableau rules for polymorphic types

Polymorphic types will be *signed* and labelled according to the rules defined in chapter 3. Tableau expansion rules α, σ and θ in table 3.1 will be extended to cover polymorphic types in the obvious way: type variables are treated as atomic types. We distinguish between two groups of signed polymorphic types: F-signed polymorphic types and T-signed polymorphic types. We refer to the former as *universal* types, or γ-types, and to the latter as *existential types*, or δ-types. By restriction (4.21) and our assumptions about existential quantification, δ-types only occur in a LLKE proof if the corresponding polymorphic type occurs on the right-hand side of a sequent. Although we see little use for this kind of sequent in linguistic description (see discussion above), rules for γ-types will be given below. The same rules may be used if one decides to extend the syntax to allow explicit universal and existential quantification as in (Emms, 1994). Schemes (6.1) and (6.2) show how to extend a LLKE tree from a node where a polymorphic type occurs. Type X is assumed to be a type in $\mathcal{C}^v$ containing *at least one* type variable x.

$$\frac{(\gamma_1)\quad F : X : a}{(\gamma_2)\quad F : X^x_A : a} \qquad \text{where } A \text{ is any type in } \mathcal{C} \tag{6.1}$$

$$\frac{(\delta_1)\quad T : X : a}{(\delta_2)\quad T : X^x_A : a} \qquad \begin{array}{l}\text{where } A \in \mathcal{C} \text{ has } \textit{not} \text{ been derived from}\\ \text{types added in previous steps that use rule } \delta\end{array} \tag{6.2}$$

The proviso on (6.2) corresponds to the *liberalisation* of the "D rule" in Smullyan-style tableaux (Fitting, 1990) whereby the constraint on A being a *new* parameter in the derivation is relaxed for computational purposes. Recall that, strictly speaking, existential rules should introduce new parameters so as to avoid conflict with parameters committed to predicates

under the scope of quantifiers to which a γ or δ rule has been previously applied. A clear explanation of this kind of conflict in first-order logic is given in (Smullyan, 1968, pp54–55). Smullyan's explanation can be paraphrased as follows. Suppose that two predicates "P" and "Q" apply to distinct existentially quantified variables in the course of a proof. It is perfectly legitimate to take a parameter, say "a", and assume "$P(a)$". However, once "a" has been committed to representing an individual with property "P", we are no longer allowed to predicate "Q" of "a", for then we would be assuming that there is an individual "a" with both properties "P" and "Q", which is stronger than what is asserted. Since no distinct predicates are used in quantified Lambek calculi it would appear that one could eliminate this proviso, collapsing γ and δ rules into the same general substitution scheme. We have chosen not to do so in the interest of soundness. The distinction between the two rules explains, for instance, why polymorphic types must be universally rather than existentially quantified. Consider the two different instantiations of the coordinator "and" (type $(x\backslash x)/x$) in example (e.39) and the ones in example (e.40). If we assume polymorphic types to be existential and observe the proviso of rule (6.2), then (e.39) will be derivable but (e.40) will be ruled out (because once the first "NP" gets introduced as a δ-parameter the introduction of a second "NP" to instantiate the other "and" will be blocked).

(e.39) John *and* Mary went out *and* Paul stayed in.

(e.40) John *and* Mary stayed in while Bill *and* Paul went out.

In picking out the types to be introduced via γ and δ rules we will give preference to (i) subtypes occurring previously on the derivation[1] and to (ii) types resulting from the closure of the former under $\mathcal{O}p$, in ascending order of degrees. These choices have a heuristic character. Although rule (6.1) will in fact allow *any* parameter to be chosen, it's easy to see that choosing subtypes which occur previously in the derivation tends to produce shorter proofs. However, strictly speaking there's no way to guarantee the algorithm's termination in cases where the initial formula is a non-theorem. Some limit, on the maximum degree of type instances for example, must be imposed in order for the procedure to be of practical use.

[1] In the δ case, those which have not been introduced by δ rules nor have been been derived from types so introduced by means of α or σ rules.

With the rules for universal and existential quantification a generalised summary of all LLKE^v rules is shown in table 6.1. The optimal order of rule application for each branch is: first α, then σ and finally γ (and δ) rules. Examples of LLKE^v derivations — (e.41), (e.43) and (e.42) — are shown in the next section.

α scheme	σ scheme	θ scheme	γ scheme	δ scheme
$\dfrac{\alpha_1}{\alpha_2}$ α_3	$\dfrac{\sigma_1}{\sigma_2}$ σ_3	$\dfrac{}{\theta_1 \mid \theta_2}$	$\dfrac{\gamma_1}{\gamma_2}$	$\dfrac{\delta_1}{\delta_2}$

Table 6.1: Generalised LLKE Rules

The completeness result of chapter 3 with respect to the sequent presentation of NL–LPCE can be extended to accommodate the new rules if standard parameter instantiation (Emms, 1994) is assumed to be the Gentzen counterpart of γ and σ since the labelling for the instantiated types remains the same.

6.1.4 Propositional Vs. predicational polymorphism: discussion

Polymorphism can be carried out in different ways: we mentioned the variety of *structural* polymorphism exhibited by L, the extensions to the propositional apparatus used in (Morrill, 1994), and presented a more general implementation of polymorphic behaviour through the use of variables[2]. Polymorphic types have received some attention since the early days of CG. Lambek's pioneering paper (Lambek, 1958) suggests that types implementing negation and coordination should undergo generalisation, resulting in the forms "x/x" and "$(x\backslash x)/x$" respectively. As remarked in (Moortgat, 1988), negation can be structurally generalised if its lexical counterpart is assigned type "S/S": this original form obviously works when the argument is a sentence, and the form yielded by main-functor division, "$(S/NP)/(S/NP)$", works for verb-phrase negation in LP. However, types with structure "$(x\backslash x)/x$" cannot be

[2] We should remark that the *predicational* system presented above does not correspond to the predicational system of (Morrill, 1994). Morrill's quantification scheme aims at providing a unification-based feature system for CG, building it on top of the propositional system, which already provides for polymorphism through additional connectives — "$\wedge$" for functor polymorphism and "$\vee$" for argument polymorphism — as well as intrinsic properties of Lambek calculi (see section 6.1.2).

generalised by means of type raising or division. The predicational approach seems to handle the coordination case satisfactorily. Consider example (e.41), a LLKE^v derivation for the sentence in (e.39) where the coordinators receive different instantiations.

(e.41)

0−	$F : N \bullet (x\backslash x)/x \bullet N \bullet N\backslash S \bullet (x\backslash x)/x \bullet S \vdash S : 1$	Assump.	
1−	$TN \bullet (x\backslash x)/x \bullet N \bullet N\backslash S \bullet (x\backslash x)/x \bullet S : i$		
3−	$F : S : i$	$0, \alpha$	
3−	$T : N : a$		
4−	$T : (x\backslash x)/x : b$		
5−	$T : N : c$		
6−	$T : N\backslash S : d$		
7−	$(x\backslash x)/x : e$		
8−	$T : S : ((((i /\!\!/ a) /\!\!/ b) /\!\!/ c) /\!\!/ d) /\!\!/ e$	$1, \alpha$	
9−	$T : (N\backslash N)/N : b$	$4, \gamma$	
10−	$T : N\backslash N : b \circ c$	$9, 5, \sigma$	
11−	$T : N : a \circ (b \circ c)$	$3, 10, \sigma$	
12−	$T : S : (a \circ (b \circ c)) \circ d$	$11, 6, \sigma$	
13−	$T : (S\backslash S)/S : e$	$7, \gamma$	
14−	$T : S\backslash S : e \circ (((((i /\!\!/ a) /\!\!/ b) /\!\!/ c) /\!\!/ d) /\!\!/ e)$	$13, 8, \sigma$	
15−	$T : S : ((a \circ (b \circ c)) \circ d)(e \circ ((((i /\!\!/ a) /\!\!/ b) /\!\!/ c) /\!\!/ d) /\!\!/ e)$	$12, 14, \sigma$	
	$\times$		

This adequacy under different contexts exhibited by the treatment of coordination via implicitly quantified type variables is due to the fact that the CG approach to coordination in general permits non-standard constituents to be derived (Ades and Steedman, 1982). That flexible notion of constituency gets inherited by the quantified calculi in cases where variables occur in an argument position of a polymorphic functor. A problem arises, however, where a variable-free type occupies the argument position of a functor which yields a polymorphic type — e.g. type $x\backslash x/N$ which generalises over the adverbial and adnominal roles of lexical item "from".

In (Morrill, 1994), such cases are dealt with by specifying a *set* of types which can occur above the division operator — for the word "from" in particular these types are: $CN\backslash CN$ for the adnominal function and $(N\backslash S)\backslash(N\backslash S)$ for the adverbial one. This list is connected by a form of (order insensitive) conjunction which has the effect of making either subtype available for further combination, which results in a complex type with the following structure: $((N\backslash S)\backslash(N\backslash S)) \wedge (CN\backslash CN)/N$. Symmetrically, if the choice appears in the argument position, then the types get connected by a form of disjunction, $\vee$, meaning that *either* type

is being sought. Unlike this propositional approach, LLKE^v polymorphism is less selective. It allows *any* type to instantiate type variables yielded in functor position. Example (e.42) shows a derivation where "from" plays an adnominal role.

(e.42)

0−	$F : \underset{John}{N} \bullet \underset{walks}{N\backslash S} \bullet \underset{from}{(x\backslash x)/N} \bullet \underset{Edinburgh}{N} \vdash S : 1$	Assump.	
1−	$T : N \bullet N\backslash S \bullet (x\backslash x)/N \bullet N : i$		
3−	$F : S : i$	$0, \alpha$	
3−	$T : N : a$		
4−	$T : N\backslash S : b$		
5−	$T : (x\backslash x)/N : c$		
6−	$T : N : ((i /\!\!/ a) /\!\!/ b) /\!\!/ c$	$1, \alpha$	
7−	$T : x\backslash x : c \circ ((i /\!\!/ a) /\!\!/ b) /\!\!/ c$	$5, 6, \sigma$	
8−	$T : S : a \circ b$	$3, 4, \sigma$	
9−	$T : (N\backslash S)\backslash(N\backslash S) : c \circ ((i /\!\!/ a) /\!\!/ b) /\!\!/ c$	$7, \gamma$	
10−	$T : N\backslash S : b \circ (c \circ ((i /\!\!/ a) /\!\!/ b) /\!\!/ c)$	$4, 9, \sigma$	
11−	$T : S : a \circ (b \circ (c \circ ((i /\!\!/ a) /\!\!/ b)) /\!\!/ c)$	$3, 10, \sigma$	
	$\times$		

The label closure conditions in this case are satisfied even in NL, since associativity is not required in order to solve the label expression yielded by the labels of lines 3 and 11. As shown in example e.43, one is also able to derive a sentence where "from" plays an adverbial role within NL.

(e.43)

0−	$F : \underset{a\ man}{CN} \bullet \underset{from}{(x\backslash x)/N} \bullet \underset{Edinburgh}{N} \vdash CN : 1$	Assump.	
1−	$T : CN \bullet (x\backslash x)/N \bullet N : i$		
3−	$F : CN : i$	$0, \alpha$	
3−	$T : CN : a$		
4−	$T : (x\backslash x)/N : b$		
5−	$T : N : (i /\!\!/ a) /\!\!/ b)$	$1, \alpha$	
7−	$T : x\backslash x : b \circ ((i /\!\!/ a) /\!\!/ b)$	$4, 5, \sigma$	
8−	$T : CN\backslash CN : b \circ ((i /\!\!/ a) /\!\!/ b)$	$7, \gamma$	
10−	$T : S : a \circ (b \circ ((i /\!\!/ a) /\!\!/ b))$	$8.9, \sigma$	
	$\times$		

Now, suppose we decide to change the instantiation of "$x\backslash x$" in (e.42), line 9, from type "$(N\backslash S)\backslash(N\backslash S)$" to "$S\backslash S$", as shown in (e.44). The former is clearly a transformation of the latter under main-functor division. Therefore, not surprisingly a type "S" can be derived in L (which allows associativity and hence main-functor division) under substitution $[x\backslash x]^x_S$.

$$0- \quad F : \begin{matrix} N \\ John \end{matrix} \bullet \begin{matrix} N\backslash S \\ walks \end{matrix} \bullet \begin{matrix} (x\backslash x)/N \\ from \end{matrix} \bullet \begin{matrix} N \\ Edinburgh \end{matrix} \vdash S : 1 \quad Assump.$$

$$\vdots$$

(e.44)
$$7- \quad T : x\backslash x : c \circ ((i /\!\!/ a) /\!\!/ b) /\!\!/ c \qquad 5, 6, \sigma$$
$$8- \quad T : S : a \circ b \qquad 3, 4, \sigma$$
$$9- \quad T : S\backslash S : c \circ ((i /\!\!/ a) /\!\!/ b) /\!\!/ c \qquad 7, \gamma$$
$$10- \quad T : S : (a \circ b) \circ (c \circ ((i /\!\!/ a) /\!\!/ b) /\!\!/ c) \qquad 8, 9, \sigma$$
$$\times$$

This extra level of generality in L — and possibly also in NL — has the unfortunate effect of allowing anomalous derivations. The problem is not particularly critical in (e.44) if one accepts Steedman's views on flexible constituency, for instance. However, cases such as the L-derivability of example (e.45) seem to pose real problems for LLKE^v.

(e.45) *the from Edinburgh man.

Although it provides an elegant formalisation of polymorphism in coordinator types, the predicational approach lacks mechanisms to block ungrammatical derivations where functor polymorphism has to be encoded. The introduction of devices such as sort restrictions on the domain of quantification — see (Moens et al., 1989) for a treatment of this in *unification categorial grammar* — may eliminate the problem. However, these seem to require techniques beyond what can be achieved by straightforward extensions to the tableau deductive apparatus. Again, formal elegance might have to be sacrificed.

6.2 What does CG parsing tell us about automated deduction in information networks?

We conclude this chapter with a modest discussion of yet another possible extension of LLKE, this time towards an area of application outside the scope of CG parsing: automated deduction in the so called *logics of information flow*. The notion of *information flow* has gained increasing attention from logicians, logically-minded linguists and theoretical computer scientists working in areas where, due to the dynamic nature of the subject, the traditional notion

of logical *truth* might not be the most adequate. In fact, since the appearance of the first works on "multi-valued" logics (Kleene, 1952) up to recent research on artificial intelligence (AI), basic assumptions of classical logic, including the principles of excluded middle and bivalence, as well as structural properties such as monotonicity, have been constantly challenged. Amongst the many examples of this trend we find AI's non-monotonic and multi-valued logics (Ginsberg, 1988) Girard's linear logic (Girard, 1987), Barwise and Perry's situation semantics (Barwise and Perry, 1983), and more recently, channel theory (Barwise, Gabbay, and Hartonas, 1995).

Although a number of techniques have been developed within AI to deal with automated modal and multi-valued reasoning — including tableaux (Fitting, 1983) and algebraic ("bilattice") approaches (Ginsberg, 1988) — there haven't been, to my knowledge, any attempts to implement provers for logics along the lines of the ones presented in (Barwise, Gabbay, and Hartonas, 1995). We will refer to these logics, where the basic models are assumed to be (partially specified, possibly self-referential) *situations* and *constraints* on the inferences allowed from one situation to another, as *logics of information flow*, or simply *LIF*s. It has been suggested (Barwise, Gabbay, and Hartonas, 1994) that the structures handled by the Lambek calculus correspond to paradigmatic LIF structures. We will suggest below that the case of parsing in categorial grammars may be taken as a paradigmatic case for automated theorem proving in LIFs.

6.2.1 Logics of information flow in Artificial Intelligence

In (Luz, 1995) we presented the issue of representing speech-acts by means plan-based formalisms (Allen and Perrault, 1980) as being an example of an AI application whose *ontology*[3] suffers from a lack of uniformity due to the fact that important notions such as *time* and *action* receive only operational specifications. Speech-act representation in dialogue systems is normally based on a theory where speech acts are defined in terms of their constituent *propositional attitudes*: *belief*, *intention*, *knowledge* etc. These notions have received consid-

[3]Defined as: "an explicit formal specification of how to represent the objects, concepts and other entities that are assumed to exist in some area of interest and the relationships that hold among them." (Howe, 1997).

erable attention within the field of possible-worlds semantics with varying degrees of success (Hintikka, 1969). Propositional attitudes alone, however, cannot account for the dynamic aspects of dialogues such as inferences about the speaker's intentions, communicative goals, etc. In order to handle these, a planning formalism is usually employed which operates on a knowledge base comprised by facts and attitudes — i.e. "snapshots" of possible worlds.

From a theoretical point of view, the incorporation of planning algorithms, a choice which might be regarded as a practical design decision, brings about some distortions. The most evident of them is that intentions are not required to be consistent. Since actions and plans reside outside the logical ontology, temporal mechanisms are assumed to operate implicitly, and the passing of time is somehow expressed as updates on the knowledge-base due to the actions performed along with a plan search. In this setting, it is reasonable to say that an agent can intend both "P" *and* "$\neg P$" because the contradictory formulae may turn out to be the case in different situations, i.e. at different snapshots. However, if the passing of time is explicitly encoded as part of the ontology — let's say, if the logic was provided with a temporal operator $\Diamond$, corresponding to "eventually" (at some point in time) — then the apparently inconsistent intentions would be translated into something like: $intent(A, \Diamond P)$ and $intend(A, \Diamond \neg P)$, thus removing the inconsistency.

There are works — such as the influential (Cohen and Levesque, 1990) — which explicitly account for temporal and a number of other indexical aspects. Works along these lines tend to be regarded as frameworks for design of situated agent systems and verification via model-checking, rather than systems for intention/goal inference. Even in verification frameworks, however, the lack of mechanisms to play the functional roles played by plans in "practically-minded" intention recognition systems produces some odd consequences. The analysis of (Cohen and Levesque, 1990)'s attempt to capture Bratman's concept of *filter of admissibility* helps to illustrate this point. "Filter of admissibility" is an expression coined by Bratman to designate the fact that intentions constrain future-directed actions and attitudes "filtering out" actions and plans incompatible with an agent's current intentions. Formula 6.3 is claimed in (Cohen and Levesque, 1990) to be the property of their (model) theory which is supposed to implement this filter. Cohen and Levesque's model combines a first-order, quantified epistemic logic (KD45) with Hoare's dynamic logic to define intentional operators

("intention", "goal" etc) by constraining the epistemic accessibility relations. The operators can be read informally as follows: "$done(x, a)$" stands for "agent x performs action a", "$\Box\phi$" stands for "ϕ is always the case", "**;**" is the dynamic logic operator for action composition, and the propositional attitude operator "bel(lieve)" receive the standard (KD45) interpretation in a possible-world model while "intend" is defined via belief and goal accessibility relations.

$$\models \forall x[intend(x, b) \wedge \Box bel(x, (done(x, a) \Rightarrow \Box\neg done(x, b))) \Rightarrow \neg intend(x, a; b)] \qquad (6.3)$$

It is clear that (6.3) is asserting some sort of filter. However, the property only applies with respect to *a particular intention* "a". There is nothing to guarantee that the agent does not have other intentions whose achievement is eventually prevented by some action the agent might perform in order to achieve a goal covered by an instance of (6.3). An example of situation in which this happens can be found in (Bratman, 1991): agent "x" has the intention ("b") of flying to San Francisco on Sunday evening. As part of a partial plan, this intention poses the problem for further deliberation: How can "x" get from the hotel to the airport? One solution might be ("a") take the limousine to the airport. Now, suppose this solution is not admissible because, say, x is also planning ("c") to meet a friend in the afternoon and the limousine leaves in the morning. If we apply (6.3) to this example, we will see that a is not ruled out, unless of course the intention considered is viewed as a conjunction of intentions "b" and "c". In any case, there seems to be much contextual information escaping through the filter. If the filter were to be improved in the same framework, the most straightforward option would be to introduce an extra level of quantification over intentions, which cannot be done without adding considerable complications into an already complicated model.

Barwise and Perry's situation semantics (Barwise and Perry, 1983) appeared as an attempt to account for those indexical aspects of knowledge representation which have been neglected by many possible-world approaches. The failure of most AI theories of language and action is often blamed on such aspects. The theory initially tackled semantic phenomena in natural language, being later generalised through the (more syntactic) notions of *channels* and *information flow* (Barwise, Gabbay, and Hartonas, 1995). The state of the art in this *channel theory* seems to be one in which semantic intuitions derived from Barwise's (and other peo-

ple's) work on Situation Theory have been integrated into a complex and expressive system, of which there are several versions around — see for instance (Cavedon, 1995) for a recent attempt covering default reasoning in natural language. On the one hand such complexity allows a wide range of phenomena to be dealt with (at least conceptually). On the other hand, the lack of a comprehensive strategy for computational treatment of the theory as well as the "semi-formal" status of some versions seem to have prevented the theory from becoming more widespread in the AI community even though its development was originally motivated by problems brought about by AI research. Consider for example the use of modal logic in some agent theories nowadays[4]: although most of them recognise the need to account for phenomena such as indexicality, which is not adequately handled in possible worlds, there is still a preference for modal logic, perhaps paradoxically, because it is more *limited* than LIFs. This state of affairs would probably change if more usable LIFs were identified along with the their properties with respect to automation.

6.2.2 LIFs in the LLKE framework

The perspective presented in (Barwise, Gabbay, and Hartonas, 1995) is that several reasoning processes can be described as networks in which logic "flows" through constraints that classify *channels* connecting *information sites*. In order to formalise this, the algebraic structure of definition 6.16 is used.

Definition 6.16 *An* Information Network *is a 4-tuple* $\mathcal{N} = \langle S, C, \rightsquigarrow, \circ \rangle$*, where* $S = \{s, t, ...\}$ *is a set of information* sites*,* $C = \{a, b, ...\}$ *is a set of* channels *between sites,* $\rightsquigarrow$ *is a relation on* $S \times C \times S$ *and* $\circ$ *a binary operation on* C *representing channel composition. In addition, it is required that for all channels* a *and* b*:* $\forall s, t[s \overset{a \circ b}{\rightsquigarrow} t \text{ iff } \exists r(s \overset{a}{\rightsquigarrow} r \wedge r \overset{b}{\rightsquigarrow} t)]$

The algebra provides a framework upon which several systems can be represented. In the particular case of the AI modelling of intentions sketched above, of interest is the fact that semantic intuitions derived from channel theory and realised in $\mathcal{N}$ could be used to model

[4] See (Wooldridge and Jennings, 1995) for a survey.

deductive planning along the lines of the approach described in (Bibel, 1997). For instance, an *action* may be regarded as a channel "c" connecting a situation or *site* "s_1" (the state of the knowledge base *before* the action is performed) to a site "s_1" (the state of the knowledge base *after* the action is performed). If planning is modelled as a deductive process (with a semantic counterpart), then the problems of ontological uniformity exhibited by plan-based models of speech acts tend to be minimised.

Before we sketch a formalisation of deductive planning within this information theoretic framework, let's describe the language defined in (Barwise, Gabbay, and Hartonas, 1995) to talk about complex relationships between channels and sites. The language consists of a set of basic types ranging over sites, say $\Sigma = \{A, A_0, ..., A_n, B, ...\}$, a set of types ranging over channels, say $\Gamma = \{C, C_0, ..., D_n\}$ and complex types. Complex types are built as shown in definition 6.17.

Definition 6.17 *The set of types* LIF *is the smallest set satisfying the following conditions:*

(i) *if* $A \in \Sigma \cup \Gamma$ *then* $A \in LIF$
(ii) *if* $A \in \Sigma$ *and* $C \in \Gamma$*, then* $(A \downarrow C) \in LIF$
(iii) *if* $A, B \in \Sigma$ *then* $(A \rightarrow B) \in LIF$
(iv) *if* $A \in \Sigma$ *and* $C \in \Gamma$*, then* $(A \leftarrow C) \in LIF$
(v) *if* $C, D \in \Gamma$ *then* $(C \circ D) \in LIF$

Operators "$\downarrow$" and "$\circ$" suggest forms of non-commutative conjunction, while the arrows resemble implication operators. Their strict interpretation, however, must be given with respect to $\mathcal{N}$. We will not give a formal definition of validity here — the reader is referred to (Barwise, Gabbay, and Hartonas, 1995) for a precise formulation — but the intuitive interpretation of sentences built according to definition 6.17 can be phrased as follows:

- $(A \downarrow C)$ is evaluated as an information site: the site connected to A by means of a channel C
- $(A \rightarrow B)$ is evaluated as a channel connecting A to B

- $(A \leftarrow C)$ is evaluated as a site connected to A via C
- $(C \circ D)$ is the syntactic counterpart of the channel composition operator (the different uses are made clear by the context)

As an example of how speech-act modelling using the LIF just outlined we use the scheme presented in (Barwise, Gabbay, and Hartonas, 1995) for encoding knowledge bases and actions in a planning domain. The approach consists of identifying first-order sentences with types over sites and channels. The former represent states of the knowledge base, while the latter correspond to action operators (also written as first-order predicates). In a propositional attitude setting, basic action (channel) types could be, for instance: $inform(a,b,p)$, $request(a,b,Inform(b,a,q))$ etc. Situation (site) types could be, for example: $believe(a,p)$, $know(a,p)$ etc. Complex types such as the ones shown below could then be composed from these primitive predicates

(e.46) $\quad \neg know(a,p) \rightarrow know(a,p)$

To represent the transition from a state where "agent a doesn't know that p" to a state where "agent a knows that p". A type such as (e.47) could stand for any action that changes the knowledge state of the agent followed by a speech act. The type in (e.48), on the other hand, could represent the post-condition of an informative act, and so on.

(e.47) $\quad (\neg know(a,p) \rightarrow know(a,p)) \circ inform(a,b,p)$

(e.48) $\quad believe(a,p) \leftarrow inform(b,a,p)$

To see the connections between this LIF and the apparatus used in LLKE consider the following[5]:

[5]Further evidence of the connection between LIFs and Lambek calculi is presented in (Barwise, Gabbay, and Hartonas, 1995). It points out that van Benthem's relational semantics for L can be built as an information network if sites and channels are taken to be ordered pairs in a relation R and define $\langle a,b\rangle \circ \langle b,c\rangle = \langle a,c\rangle$ along with an identity element as in remark 1. In this information network we then have $\langle a,b\rangle \overset{\langle b,c\rangle}{\rightsquigarrow} \langle a,c\rangle$ iff $\langle a,b\rangle, \langle b,c\rangle \in R$, where R is transitive

Remark 1 *An information frame $\mathcal{L} = \langle \mathcal{P}, \circ, u, \sqsubseteq \rangle$ (definition 3.2) yields an information network $\mathcal{N} = \langle S, C, \rightsquigarrow, \circ \rangle$ (definition 6.16) in a canonical way. Just let $\mathcal{P} = S = C$ and define a unary channel u which connects a site to itself so that $c \circ u = u \circ c = c$. Now stipulate $a \sqsubseteq b \overset{def}{=} a \overset{a}{\rightsquigarrow} b$. It is routine to verify that the structure defined in this way satisfies definition 6.16, in particular the two cases of matching the requirement on channel composition: if $a \overset{a \circ u}{\rightsquigarrow} b$ then there is a site, namely b, s. t. $a \overset{a}{\rightsquigarrow} b$ $b \overset{u}{\rightsquigarrow} b$ (by definition of u) and conversely if there is a site b s.t. the latter holds then $a \overset{a \circ u}{\rightsquigarrow} b$. The same reasoning applies to the permuted case.*

Now, if we assume that CG types correspond to a LIF's type and that the semantic structure underlying CG is an information network such as $\mathcal{N}$, then we get a framework for the calculi mentioned above by making $S = C$ be a set of tokens supporting CG types, and $\rightsquigarrow$ the semantic counterpart for type composition. We can then assume that $\leftarrow$, $\rightarrow$ and $\circ$ correspond respectively to / \ and $\bullet$, obtaining the CG syntax. From this perspective, many of the theorem proving techniques developed in the previous sections can be imported to deal with (at least fragments) of LIFs. The fragments left out of this characterisation include the system based on a two-sorted language in which the distinction between sites and channels is syntactically expressed, and the rather more complex elaborations in the domain of infinitary logic. It would be interesting to explore these fields in connection with the applications described above. However, this would require a framework much more complex than the formulation of LLKE presented in this thesis.

6.3 Summary and conclusions

In this chapter we presented the (modest) beginnings of two (ambitious) extensions to LLKE: the treatment of quantification, which has ramifications into the CG research on polymorphism and polymorphic types, and theorem proving in Barwise's logics of information flow.

Polymorphism was treated by allowing a restricted modality of universal quantification into the type language. Tableau rules were provided which generalised over universal and existential quantification, though the latter does not seem to play any relevant role in grammar

specification. We then discussed non-predicational implementations of polymorphism and pointed out the fact that even if universal quantification is bound within product limits the logic tends to overgenerate in cases of functor polymorphism. The restriction on quantification is kept, however, since it appears to be in line with the economy we had pursued in LLKE. Such pursuit got somewhat neglected in the second part of the chapter, where we introduced the parallel between parsing in CG and theorem proving in logics of information flow.

The choice of CG parsing as a paradigmatic case for LIF theorem proving seems interesting due to the following factors: (1) the Lambek Calculus, the logic on which most CGs are based, corresponds to a canonical form of information network (as pointed out in (Barwise, Gabbay, and Hartonas, 1995)); (2) As well LIFs, categorial grammars (under the perspective of "parsing as deduction") require very general theorem provers to cope with the variety of calculi, and (3) complexity problems which arise from keeping track of proof structures are likely to appear (in analogous forms) in strategies for automating LIFs. Furthermore, the experience obtained by the better consolidated research in the former may give us some insight into what strategies to use when dealing with applications requiring more complex networks and on assessing the feasibility of defining and implementing them.

Finally, we should remark that other approaches to automated deduction in logics of information flow building on the connections between these logics and Lambek calculi are starting to appear. An interesting tableau-based system which represents semantic relations directly in the tree expansion rules has been presented in (MacCaull, 1997). As noted in (Venema, 1996), "labelling can introduce as many problems as it solves". We have seen some such problems from the complexity perspective in the previous chapters. In addition to these, we have started to see in this chapter that the "semantics" of LDS deduction defined in terms of chapter 3's information frames is not entirely clear. Perhaps labelled deduction could benefit from the more semantical approach brought about by the work in LIFs.

Chapter 7

Conclusions

In this thesis we have presented an account of computational properties of parsing in Lambek calculi based on a version of analytic deduction. The work presented here was intended as a first step towards bridging the gap between categorial grammar parsing (as deduction) and automated theorem proving. Therefore we regard the main contributions of this thesis as concerning these two fields of research in equal proportion.

From the point of view of labelled automated deduction in resource-sensitive logics it is worth mentioning the following:

- The semi-decision procedure for the labelled tableau of (D'Agostino and Gabbay, 1994) has been extended into a full decision procedure for a range of (implicational fragments of) substructural calculi whereas the original soundness and completeness results have been preserved.
- The impact of cut-elimination on efficiency (in the proposed tableau system) has been assessed from the perspective of redundant proof patterns (i.e. spurious ambiguity) and with respect to the label-checking module — recall (section 3.2.2) that LLKE is based on a tableau system which reinstates the cut rule as an effective tool for proof search (D'Agostino and Mondadori, 1994).

- An algorithm tailored to perform label (i.e. tableau closure) checking in Lambek calculi has been described which is aimed at avoiding the complexity pitfalls of associative and commutative unification.

- A natural way to integrate the label unification procedure with the general tableau proof-search regime has been presented so that label variables can be *cached* dynamically which enables branch closure testing to be performed as the tableau is expanded, making full tableau expansion unnecessary in most cases.

- Extensions of the system into the field of predicate logic and logics of information flow have been presented which are motivated by the possibility of doing label-checking concurrently with syntactic proof-search. One of the main obstacles to the automation of labelled deductive systems is the high computational costs associated with unification in the labelling algebra. By emphasizing the connection between Lambek calculi and logics of information flow we sought to suggest that general automated deduction has lessons to learn from categorial parsing as well as the other way around.

From the perspective of categorial grammar parsing, the following contributions should be pointed out:

- An approach to natural language parsing has been presented which builds on a strictly deductive method. This is to our knowledge the first application of tableau theorem proving to categorial grammar

- A strategy for recovering grammatical information (both lexical and combinatorial) from the graph which encodes the proof search has been developed

- Complexity results for the tableau system reminiscent of those for context-free and mildly context-sensitive grammars have been achieved for the class of Lambek calculi in the substructural hierarchy

- Comparisons between the parsing mechanisms presented in this thesis and other deductive systems for categorial grammars such as those based on proof-nets, sequent systems and natural deduction have been presented

- A modest extension of the system to deal with polymorphic types by adding tableau expansion rules to deal with restricted universal quantification has been presented. The limitations of this kind of approach to type polymorphism with respect to linguistic description have been discussed

The analysis of the theoretical issues mentioned above has been accompanied by substantial implementation effort in Lisp [1] to validate practically the techniques described in this thesis.

Much work remains to be done both in our approach to categorial grammar parsing and in the vast area of automated deduction for logics of information flow in general. Among these we could mention: the specialisation of the techniques developed here towards hybrid logics and substructural modalities, a finer grained account of polymorphism, a deeper investigation of how CG's syntax-semantics interface (Curry-Howard isomorphism) can be implemented in LLKE, the study of how linguistic knowledge could be incorporated to the theorem proving mechanisms to improve efficiency, and the definition of information calculi of practical interest (e.g. in artificial intelligence applications) which fall within the class of logics covered by the general deductive techniques developed in this thesis. This thesis will have achieved its goals if it managed to convince the reader that further research along these lines in the deductive framework presented here is a task worth pursuing.

[1] The LLKE prototype is available upon request to S.F.Luz@ed.ac.uk. It runs on Lisp interpreters which comply with (Steele, 1990). The system has been mainly tested in *Allegro Common Lisp*© but it should also run in *CMU Lisp* and *Gnu Common Lisp*.

Appendix A

Sample LLKE proofs

A.1 Characteristic Theorems of L

The following is a printout of the proofs of L-properties (2.1)–(2.7) by LLKE. The trace shows types in Lisp array notation and the final proof tree is printed as a Lisp strucutre. The symbols used are: '!' for '\', '/' for '/', '@' for '•', '<-' for '//' and '0' for '∘'. A trace is shown only for the first proof. The proof-tree containing the expanded SLFs as well as the closure pair (in the slot named "CONSTRAINT") is printed for the remaining proofs. The example was run in Allegro CL and the execution took 1,300 msec CPU time on a SUNW,SPARCstation-20 (SunOS 5.5.1).

```
LK(3): (load "L-theorems.lsp")
; Loading ./L-theorems.lsp
 **** Proving right application #(#(X / Y) @ Y) |- X
*** 22 Jul 1997 19:32:12: 3 starting to build a proof tree for: ***
F  X : L0 =>
T  #(#(X / Y) @ Y) : L0 =>
** Alpha-Reducing:
 T  #(#(X / Y) @ Y) : L0 =>
   Expanding tree with:
 T  #(X / Y) : L1 =>  And: T  Y : (L0 <- L1) =>
** Sigma-combining:
 T  #(X / Y) : L1 =>    With:  T  Y : (L0 <- L1) =>
   And expanding tableu with:
 T  X : (L1 0 (L0 <- L1)) =>
 ** Potential closure between
 T  X : (L1 0 (L0 <- L1)) =>  and  F  X : L0 =>
## Inspecting Potential closure between
 (L1 0 (L0 <- L1)) and  L0
```



```
## Fully reduced terms satisfy:
 ((L0) NIL) <=  ((L0) NIL)
 ** Matching Satisfied in L
*** Linear Expansion ended. Starting b-exp **
*** 22 Jul 1997 19:32:13: Final tableau has no open branches at level 3. ***
#S(TABLEAU :ROOT
           (#S(SLF :SIGNAL F :FORMULA X :LABEL L0 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(#(X / Y) @ Y) :LABEL L0 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(X / Y) :LABEL L1 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA Y :LABEL (L0 <- L1) :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA X :LABEL (L1 0 (L0 <- L1))
                   :LEXICON ""))
           :LEFTBR NIL :RIGHTBR NIL
           :CONSTRAINT
           (#S(SLF :SIGNAL T :FORMULA X :LABEL (L1 0 (L0 <- L1))
                   :LEXICON "")
            #S(SLF :SIGNAL F :FORMULA X :LABEL L0 :LEXICON "")))

**** Proving left application #( Y @ #(Y ! X)) |- X
*** 22 Jul 1997 19:32:13: 3 starting to build a proof tree for: ***
F  X : L2 =>
T  #(Y @ #(Y ! X)) : L2 =>
 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:13: Final tableau has no open branches at level 3. ***
#S(TABLEAU :ROOT
           (#S(SLF :SIGNAL F :FORMULA X :LABEL L2 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(Y @ #(Y ! X)) :LABEL L2 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA Y :LABEL L3 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(Y ! X) :LABEL (L2 <- L3)
                   :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA X :LABEL (L3 0 (L2 <- L3))
                   :LEXICON ""))
           :LEFTBR NIL :RIGHTBR NIL
           :CONSTRAINT
           (#S(SLF :SIGNAL T :FORMULA X :LABEL (L3 0 (L2 <- L3))
                   :LEXICON "")
            #S(SLF :SIGNAL F :FORMULA X :LABEL L2 :LEXICON "")))

**** Proving right composition: #( #(X / Y) @ #(Y / Z)) |- #(X / Z)
*** 22 Jul 1997 19:32:13: 5 starting to build a proof tree for: ***
F  #(X / Z) : L4 =>
T  #(#(X / Y) @ #(Y / Z)) : L4 =>
 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:13: Final tableau has no open branches at level 5. ***
#S(TABLEAU :ROOT
           (#S(SLF :SIGNAL F :FORMULA #(X / Z) :LABEL L4 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(#(X / Y) @ #(Y / Z)) :LABEL L4
                   :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA Z :LABEL L5 :LEXICON "")
            #S(SLF :SIGNAL F :FORMULA X :LABEL (L4 0 L5) :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(X / Y) :LABEL L6 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(Y / Z) :LABEL (L4 <- L6)
                   :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA Y :LABEL ((L4 <- L6) 0 L5)
                   :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA X :LABEL (L6 0 ((L4 <- L6) 0 L5))
                   :LEXICON ""))
```

```
          :LEFTBR NIL :RIGHTBR NIL
          :CONSTRAINT
          (#S(SLF :SIGNAL T :FORMULA X :LABEL (L6 0 ((L4 <- L6) 0 L5))
                  :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA X :LABEL (L4 0 L5) :LEXICON"")))

**** Proving left composition: #( #(Z ! Y) @ #(Y ! X)) |- #(Z ! X)
*** 22 Jul 1997 19:32:13: 5 starting to build a proof tree for: ***
F  #(Z ! X) : L7 =>
T  #(#(Z ! Y) @ #(Y ! X)) : L7 =>
 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:13: Final tableau has no open branches at level 5. ***
#S(TABLEAU :ROOT
          (#S(SLF :SIGNAL F :FORMULA #(Z ! X) :LABEL L7 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA #(#(Z ! Y) @ #(Y ! X)) :LABEL L7
                  :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA Z :LABEL L8 :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA X :LABEL (L8 0 L7) :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA #(Z ! Y) :LABEL L9 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA #(Y ! X) :LABEL (L7 <- L9)
                  :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA Y :LABEL (L8 0 L9) :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA X :LABEL ((L8 0 L9) 0 (L7 <- L9))
                  :LEXICON ""))
          :LEFTBR NIL :RIGHTBR NIL
          :CONSTRAINT
          (#S(SLF :SIGNAL T :FORMULA X :LABEL ((L8 0 L9) 0 (L7 <- L9))
                  :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA X :LABEL (L8 0 L7) :LEXICON "")))

**** Proving right type-raising: X |- #(Y / #(X ! Y))
*** 22 Jul 1997 19:32:13: 3 starting to build a proof tree for: ***
F  #(Y / #(X ! Y)) : L10 =>
T  X : L10 =>
 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:13: Final tableau has no open branches at level 3. ***
#S(TABLEAU :ROOT
          (#S(SLF :SIGNAL F :FORMULA #(Y / #(X ! Y)) :LABEL L10 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA X :LABEL L10 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA #(X ! Y) :LABEL L11 :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA Y :LABEL (L10 0 L11) :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA Y :LABEL (L10 0 L11) :LEXICON ""))
          :LEFTBR NIL :RIGHTBR NIL
          :CONSTRAINT
          (#S(SLF :SIGNAL T :FORMULA Y :LABEL (L10 0 L11) :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA Y :LABEL (L10 0 L11) :LEXICON "")))

**** Proving left type-raising: X |- #(#(Y / X) ! Y)
*** 22 Jul 1997 19:32:13: 3 starting to build a proof tree for: ***
F  #(#(Y / X) ! Y) : L12 =>
T  X : L12 =>

 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:13: Final tableau has no open branches at level 3. ***
#S(TABLEAU :ROOT
          (#S(SLF :SIGNAL F :FORMULA #(#(Y / X) ! Y) :LABEL L12 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA X :LABEL L12 :LEXICON "")
```



```
           #S(SLF :SIGNAL T :FORMULA #(Y / X) :LABEL L13 :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA Y :LABEL (L13 0 L12) :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA Y :LABEL (L13 0 L12) :LEXICON ""))
          :LEFTBR NIL :RIGHTBR NIL
          :CONSTRAINT
          (#S(SLF :SIGNAL T :FORMULA Y :LABEL (L13 0 L12) :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA Y :LABEL (L13 0 L12) :LEXICON
           "")))

**** Proving right division (main functor): #(X / Y) |- #(#(X / Z) / #(Y / Z))
*** 22 Jul 1997 19:32:13: 5 starting to build a proof tree for: ***
F  #(#(X / Z) / #(Y / Z)) : L14 =>
T  #(X / Y) : L14 =>
 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:14: Final tableau has no open branches at level 5. ***
#S(TABLEAU :ROOT
          (#S(SLF :SIGNAL F :FORMULA #(#(X / Z) / #(Y / Z)) :LABEL L14
                  :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA #(X / Y) :LABEL L14 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA #(Y / Z) :LABEL L15 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA Z :LABEL L16 :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA X :LABEL ((L14 0 L15) 0 L16)
                  :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA Y :LABEL (L15 0 L16) :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA X :LABEL (L14 0 (L15 0 L16))
                  :LEXICON ""))
          :LEFTBR NIL :RIGHTBR NIL
          :CONSTRAINT
          (#S(SLF :SIGNAL T :FORMULA X :LABEL (L14 0 (L15 0 L16))
                  :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA X :LABEL ((L14 0 L15) 0 L16)
                  :LEXICON "")))

**** Proving left division (main functor): #(Y ! X) |- #(#(Z ! Y) ! #(Z ! X))
*** 22 Jul 1997 19:32:14: 5 starting to build a proof tree for: ***
F  #(#(Z ! Y) ! #(Z ! X)) : L17 =>
T  #(Y ! X) : L17 =>
 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:14: Final tableau has no open branches at level 5. ***
#S(TABLEAU :ROOT
          (#S(SLF :SIGNAL F :FORMULA #(#(Z ! Y) ! #(Z ! X)) :LABEL L17
                  :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA #(Y ! X) :LABEL L17 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA #(Z ! Y) :LABEL L18 :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA Z :LABEL L19 :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA X :LABEL (L19 0 (L18 0 L17))
                  :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA Y :LABEL (L19 0 L18) :LEXICON "")
           #S(SLF :SIGNAL T :FORMULA X :LABEL ((L19 0 L18) 0 L17)
                  :LEXICON ""))
          :LEFTBR NIL :RIGHTBR NIL
          :CONSTRAINT
          (#S(SLF :SIGNAL T :FORMULA X :LABEL ((L19 0 L18) 0 L17)
                  :LEXICON "")
           #S(SLF :SIGNAL F :FORMULA X :LABEL (L19 0 (L18 0 L17))
                  :LEXICON "")))
```

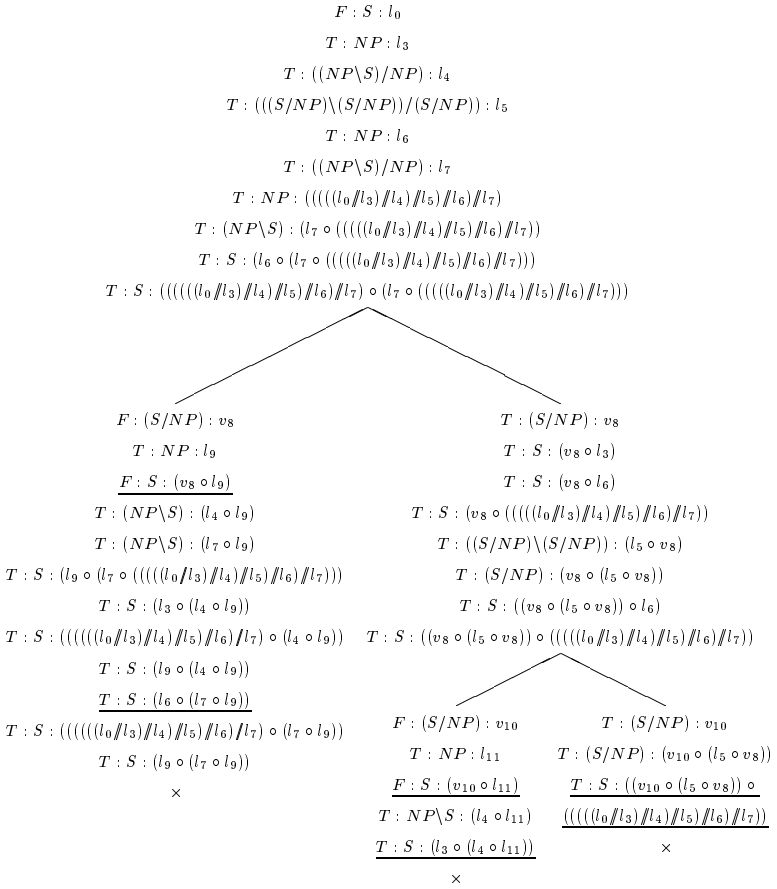
```
**** Proving right division (sub-functor): #(X / Y) |- #(#(Z / X) ! #(Z / Y))
*** 22 Jul 1997 19:32:14: 5 starting to build a proof tree for: ***
F  #(#(Z / X) ! #(Z / Y)) : L20 =>
T  #(X / Y) : L20 =>
 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:14: Final tableau has no open branches at level 5. ***
#S(TABLEAU :ROOT
           (#S(SLF :SIGNAL F :FORMULA #(#(Z / X) ! #(Z / Y)) :LABEL L20
                   :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(X / Y) :LABEL L20 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(Z / X) :LABEL L21 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA Y :LABEL L22 :LEXICON "")
            #S(SLF :SIGNAL F :FORMULA Z :LABEL ((L21 0 L20) 0 L22)
                   :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA X :LABEL (L20 0 L22) :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA Z :LABEL (L21 0 (L20 0 L22))
                   :LEXICON ""))
           :LEFTBR NIL :RIGHTBR NIL
           :CONSTRAINT
           (#S(SLF :SIGNAL T :FORMULA Z :LABEL (L21 0 (L20 0 L22))
                   :LEXICON "")
            #S(SLF :SIGNAL F :FORMULA Z :LABEL ((L21 0 L20) 0 L22)
                   :LEXICON "")))

**** Proving left division (sub-functor): #(Y ! X) |- #(#(Y ! Z) / #(X ! Z))
*** 22 Jul 1997 19:32:14: 5 starting to build a proof tree for: ***
F  #(#(Y ! Z) / #(X ! Z)) : L23 =>
T  #(Y ! X) : L23 =>
 ** Matching Satisfied in L
*** 22 Jul 1997 19:32:14: Final tableau has no open branches at level 5. ***
#S(TABLEAU :ROOT
           (#S(SLF :SIGNAL F :FORMULA #(#(Y ! Z) / #(X ! Z)) :LABEL L23
                   :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(Y ! X) :LABEL L23 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(X ! Z) :LABEL L24 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA Y :LABEL L25 :LEXICON "")
            #S(SLF :SIGNAL F :FORMULA Z :LABEL (L25 0 (L23 0 L24))
                   :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA X :LABEL (L25 0 L23) :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA Z :LABEL ((L25 0 L23) 0 L24)
                   :LEXICON ""))
           :LEFTBR NIL :RIGHTBR NIL
           :CONSTRAINT
           (#S(SLF :SIGNAL T :FORMULA Z :LABEL ((L25 0 L23) 0 L24)
                   :LEXICON "")
            #S(SLF :SIGNAL F :FORMULA Z :LABEL (L25 0 (L23 0 L24))
                   :LEXICON "")))

; cpu time (non-gc) 1,210 msec user, 40 msec system
; cpu time (gc)     90 msec user, 70 msec system
; cpu time (total)  1,300 msec user, 110 msec system
; real time  1,545 msec
; space allocation:
;  34,654 cons cells, 26 symbols, 139,624 other bytes
T
```


A.2 Complex LLKE derivation with multiple branches

A derivation tree for (4.3) with no re-bracketing allowed at the syntactical level:



The label constraints (read off the tree in a depth-first fashion) are as follows:

$$((v_{10} \circ (l_5 \circ v_8)) \circ ((((((l_0 /\!\!/ l_3) /\!\!/ l_4) /\!\!/ l_5) /\!\!/ l_6) /\!\!/ l_7)) \quad \sqsubseteq \quad l_0 \tag{A.1}$$

$$((l_3 \circ l_4) \circ l_{11}) \quad \sqsubseteq \quad (v_{10} \circ l_{11}) \tag{A.2}$$

$$(l_6 \circ (l_7 \circ l_9)) \quad \sqsubseteq \quad (v_8 \circ l_9) \tag{A.3}$$

They are satisfied with substitution mapping below plus associativity:

$$\varsigma \quad = \quad \{v_8 \mapsto (l_6 \circ l_7), v_{10} \mapsto (l_3 \circ l_4)\} \tag{A.4}$$

A.3 Recovering a non-atomic succeedent

The derivation below shows an example of closure pair which recovers the syntactic structure of a non-atomic type, a verb phrase. The entailent to be proved is as follows:

$$NP \bullet (NP \backslash S)/NP \vdash S/NP \tag{A.5}$$

Notice that the labels introduced in the linear expansion of the succeedent (i.e. $|S/NP|_{\alpha\sigma}$) must be removed from recovering formula's labelexp — in the example, `L31` from the constraint on `:LEFTBR` —. After the removal has been done, label variable (`?L32`) instantiated, and `canconst` reduction performed, the resulting label, $(l_{30} \circ (l_{29} /\!\!/ l_{30}))$, contains the relevant ingredients of the proof the target type.

```
LK> (parser '(john loves) '#(S / NP))

*** 23 Jul 1997 21:25:23: starting to build a proof tree for: ***
F  #(S / NP) : L29 =>
T  NP : L30 => JOHN
T  #(#(NP ! S) / NP) : (L29 <- L30) => LOVES
*** Linear Expansion ended. Starting b-exp **
** Finished B-expansion with:
 NP from #(NP ! S)
 ** Matching Satisfied in L
*** 23 Jul 1997 21:25:27: Final tableau has no open
     branches at level 5:
#S(TABLEAU :ROOT
           (#S(SLF :SIGNAL F :FORMULA #(S / NP) :LABEL L29 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA NP :LABEL L30 :LEXICON JOHN)
            #S(SLF :SIGNAL T :FORMULA #(#(NP ! S) / NP) :LABEL (L29 <- L30)
                   :LEXICON LOVES)
            #S(SLF :SIGNAL T :FORMULA NP :LABEL L31 :LEXICON "")
            #S(SLF :SIGNAL F :FORMULA S :LABEL (L29 0 L31) :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(NP ! S) :LABEL ((L29 <- L30) 0 L30)
                   :LEXICON ""))
           :LEFTBR
```



```
#S(TABLEAU :ROOT
           (#S(SLF :SIGNAL T :FORMULA NP :LABEL ?L_32 :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA #(NP ! S)
                   :LABEL ((L29 <- L30) 0 ?L_32) :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA S
                   :LABEL (?L_32 0 ((L29 <- L30) 0 L30)) :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA S
                   :LABEL (L30 0 ((L29 <- L30) 0 ?L_32)) :LEXICON "")
            #S(SLF :SIGNAL T :FORMULA S
                   :LABEL (?L_32 0 ((L29 <- L30) 0 ?L_32))
                   :LEXICON ""))
           :LEFTBR NIL :RIGHTBR NIL
           :CONSTRAINT ( #S(SLF :SIGNAL T :FORMULA S
                                :LABEL (L30 0 ((L29 <- L30) 0 L31))
                                :LEXICON "")
                         #S(SLF :SIGNAL F
                                :FORMULA S
                                :LABEL (L29 0 L31)
                                :LEXICON "")    )
:RIGHTBR
#S(TABLEAU :ROOT
           (#S(SLF :SIGNAL F :FORMULA NP :LABEL ?L_32 :LEXICON ""))
           :LEFTBR NIL :RIGHTBR NIL
           :CONSTRAINT (#S(SLF :SIGNAL T
                               :FORMULA NP
                               :LABEL L31 :LEXICON JOHN)
                       #S(SLF :SIGNAL F
                              :FORMULA NP
                              :LABEL ?L_32 :LEXICON "")))
:CONSTRAINT NIL)
```

A.4 LLKE and proof redundancy

The L-theorem $X/(Y/Z) \bullet Y/W \bullet W/Z \vdash X$ receives 2 semantically equivalent derivations in a non-normal form theorem prover based on sequents. In the system of (Hepple, 1990) this redundancy is eliminated through proof normalisation. The LLKE-derivation (e.49) below shows how reasoning by lemmas is used to produce a short (and unique) proof for the theorem.

(e.49)

$$
\begin{array}{llll}
0. & F : X/(Y/Z) \bullet Y/W \bullet W/Z \vdash X & : 1 & \\
1. & T : X/(Y/Z) : a & & \ldots \\
2. & T : Y/W : b & & \ldots \\
3. & T : W/Z : (i /\!\!/ a) /\!\!/ b & & \ldots \\
4. & F : X : i & & 1, \alpha_{iii}
\end{array}
$$

$$
\begin{array}{llll}
5. & F : Y/Z : x & 1, \theta & \\
6. & T : Z : c & \ldots & \\
7. & F : Y : x \circ c & 5, \alpha_{ii} & \\
8. & T : W : ((i /\!\!/ a) /\!\!/ b) \circ c & 3, 6, \sigma_{iv} & \\
9. & T : Y : b \circ (((i /\!\!/ a) /\!\!/ b) \circ c) & 2, 8, \sigma_{iv} & \\
 & \times & &
\end{array}
\qquad
\begin{array}{llll}
10. & T : Y/Z : x & 1, \theta & \\
11. & T : X : a \circ x & 1, 10, \sigma_{iv} & \\
 & \times & &
\end{array}
$$

The closing pairs are found in the following lines: 7 and 9, on the righ-hand branch, and 4 and 11, on the branch which recovers the structure of X. The substitution mapping which guarantees closure is given by $\varsigma = \{x \mapsto b \circ ((i /\!\!/ a) /\!\!/ b)\}$, and the constraints for both branches are solved by straightforward associativity plus property (3.1).

Appendix B

LLKE Time profile

Figure B.1 shows the typical execution time profile of an LLKE proof. All closure checks were forced to fail so that a fully expanded tableau would be generated, illustrating a worst case situation.

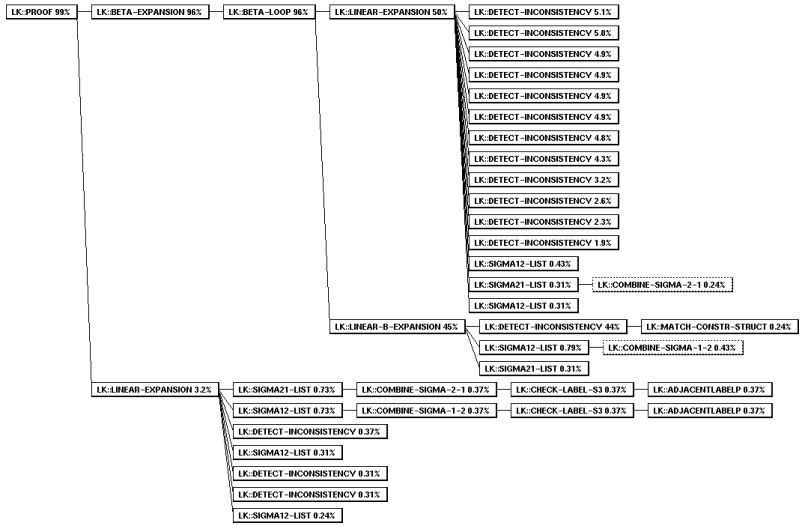


Figure B.1: LLKE execution profile

Input/output functions, as well as low-level functions whose contribution is considered in-

significant in the total profile data sample[1], details and nested calls have been omitted for the sake of space. Notice that label-checking, i.e. `LK::DETECT-INCONSISTECY` contributes most of the execution time in linear expansion, while sysntactic expansion itself (`LK::SIGMA12-LIST, LK::SIGMA21-LIST, LK::COMBINE-SIGMA-1-2-LIST`) takes a relatively small share of the total profile data sample.

The graph was created using the Allegro Common Lisp profiling tool.

[1]I.e. those whose ratio of the sum of the function and all of its callees, recursively and the total data recorded in the whole profile sample is less than 0.002.

References

Ades, A. and M. Steedman. 1982. On the order of words. *Linguistics and Philosphy*, 4:517–558.

Ajdukiewicz, K. 1935. Die syntaktische Konnexität. *Studia Philosophica*, 1(1–27). (Translation in S. McCall (ed) *Polish Logic 1920-1939* Oxford).

Allen, James and Raymond Perrault. 1980. Analyzing intention in utterances. *Artificial Intelligence*, 15:143–178.

Allwein, Gerard and Jon M. Dunn. 1993. Kripke models for linear logic. *Journal of Symbolic Logic*, 58(2):514–545.

Andreka, H. and S. Mikulas. 1994. Lambek calculus and its relational semantics: completeness and incompleteness. *Journal of Logic, Language and Information*, 3(1):1–37.

Association for Computational Linguistics. 1995. 7^{th} *Conference of the European Chapter of the ACL*, Dublin, Ireland, March. Morgan Kaufmann.

Baader, F. and J.H. Siekmann. 1993. Unification theory. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, Oxford, UK.

Bar-Hillel. 1953. A quasi-arithmetical notation for syntactic description. *Language*, 29:47–58.

Bar-Hillel, Y, C Gayfman, and E Shamir. 1960. On categorial and phrase structure grammars. *Bulletin of the Research Council of Israel*, 9F:1–16.

Barry, Guy and Glyn Morrill, editors. 1990. *Studies in Categorial Grammar*, volume 5 of *Edinburgh Working Papers in Cognitive Science*. Centre for Cognitive Science, University of Edinburgh.

Barwise, J., D. Gabbay, and C. Hartonas. 1994. Information flow and the lambek calculus. In J. Seligman and D. Westerstahl, editors, *Logic, Language and Computation: The 1994 Moraga Proceedings*. CSLI, Stanford, CA.

Barwise, J., D. Gabbay, and C. Hartonas. 1995. On the logic of information flow. *Journal of the Interest Group in Pure and Applied Logic (IGPL)*, 3(1):7–50.

Barwise, Jon and John Perry. 1983. *Situations and attitudes*. MIT Press, Cambridge, MA.

Bibel, Wolfgang. 1981. On matrices with connections. *Journal of the Association for Computing Machinery*, 28:633–645.

Bibel, Wolfgang. 1997. Let's Plan It Deductively. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, volume 2, pages 1549–1562. Morgan Kaufmann Publishers, Inc., August 23–29.

Boolos, George S. 1984. Don't eliminate cut. *Journal of Philosophical Logic*, 13:373–378.

Bratman, Michael E. 1991. Planning and the stability of intention. Technical Report CSLI-91-159, CSLI.

Buszkowski, Wojciech. 1986. Completeness results for Lambek syntactic calculus. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, 32:13–28.

Buszkowski, Wojciech. 1988. Generative power of categorial grammars. In Richard Oehrle et al., editor, *Categorial Grammars and Natural Language Structures*. Reidel, pages 69–94.

Buszkowski, Wojciech. 1996. Extending Lambek grammars to basic categorial grammars. *Journal of Logic, Language and Information*, 5:279–295.

Calder, Jonathan, Ewan Klein, and Henk Zeevat. 1988. Unification categorial grammar: A concise, extendable grammar for natural language processing. In *Proceedings of COLING '88*, Budapest.

Carpenter, Bob. 1995. The Turing-completeness of categorial grammars. Manuscipt available at http://macduff.andrew.cmu.edu/cg/.

Carpenter, Bob. 1997. *Lectures on Type-Logical Semantics*. MIT Press.

Cavedon, Laurence. 1995. *A Channel Theoretic Approach to Conditional Reasoning*. Ph.D. thesis, University of Edinburgh, Centre for Cognitive Science.

Chomsky, Noam and George A. Miller. 1963. Introduction to the formal analysis of natural language. In *Handbook of mathematical psychology*. Wiley, New York, pages 269–322.

Christian, Jim. 1989. Fast Knuth-Bendix completion. In *Proceedings of the Conference on Rewriting Techniques and Applications*, page April, Chapel Hill, North Carolina.

Cohen, J M. 1967. The equivalence of two concepts of categorial grammar. *Information and Control*, 10:475–484.

Cohen, P. and H. Levesque. 1990. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261.

Cook, S. A. and R Reckhow. 1979. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, pages 36–50.

D'Agostino, Marcello. 1992. Are tableaux an improvement on truth-tables? *Journal of Logic, Language and Information*, 1:225–252.

D'Agostino, Marcello and Dov Gabbay. 1994. A generalization of analytic deduction via labelled deductive systems I: Basic substructural logics. *Journal of Automated Reasoning*.

D'Agostino, Marcello and Marco Mondadori. 1994. The taming of the cut: Classical refutations with analytic cut. *Journal of Logic and Computation*, 4:285–319.

Dershowitz, Nachum and Jean-Pierre Jouannaud. 1990. Rewrite systems. In *Handbook of theoretical computer science*, volume Vol.B Formal models and semantics. The MIT Press: Cambridge, MA, chapter 6, pages 245–320.

Došen, Kosta. 1992. A brief survey of frames for the Lambek calculus. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, 38:179–187.

Dowty, David. 1988. Type raising, functional composition, and non-constituent conjunction. In Deirdre Wheeler Richard T. Oehrle, Emmon Bach, editor, *Categorial Grammars and Natural Language Structures*. Reidel Publishing Co, Dordrecht, pages 153–197.

D.R. Dowty, R.E. Wall and S. Peters. 1981. *Introduction to Montague Semantics*. D. Reidel, Dordrecht, Holland.

Eisner, Jason. 1996. Efficient normal-form parsing for combinatory categorial grammar. In *Proceedings of ACL (34th Meeting of the Association for Computational Linguistics)*, Santa Cruz. cmp-lg/9605038.

Emms, Martin. 1990. Polymorphic quantifiers. In Barry and Morrill (Barry and Morrill, 1990), pages 65—111.

Emms, Martin. 1994. Completeness results for polymorphic Lambek calcullus. In Moortgat (Moortgat, 1994a).

Fitting, Melvin. 1983. *Proof methods for modal and intuitionistic logics*. D. Reidel, Dordrecht Lancaster.

Fitting, Melvin. 1990. *First-order Logic and Automatic Theorem Proving*. Texts and Monographs in Computer Science. Springer-Verlag, New York.

Flynn, Michael. 1983. A categorial theory of structure building. In Geoffrey K Pullum Gerald Gazdar, Ewan Klein, editor, *Order, Concord and Constituency*. Foris Publications, pages 138–174.

Gabbay, Dov M. 1994. LDS – Labelled Deductive Systems, volume 1 — foundations. Technical Report MPI-I-94-223, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

Geach, P. 1972. A program for syntax. In G. Harman D. Davidson, editor, *Semantics of Natural Language*. Reidel, Dordrecht, pages 483–497.

Gentzen, Gerhard. 1969. Investigations into logical deductions. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, Studies in logic and the foundations of mathematics. North-Holland Pub. Co., pages 68–131.

Ginsberg, Matthew L. 1988. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316.

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

Girard, Jean-Yves. 1995. Linear logic: its syntax and semantics. In *Advances in linear logic* (Girard, Lafont, and Regnier, 1995), pages 1–42.

Girard, Jean-Yves, Ives Lafont, and L Regnier. 1995. London Mathematical Society, Lecture Note Series. Cambridge University Press.

Haken, A. 1985. The intractability of resolution. *Theoretical Computer Science*, 39:297–308.

Hendriks, Herman. 1993. Lambek semantics. In Hans Leiß, editor, *Categorial Parsing and Normalisation*, Dyana Deliverable R1.1.A.

Hepple, Mark. 1990. *The grammar and processing of order and dependency.* Ph.D. thesis, University of Edinburgh, CCS.

Hepple, Mark. 1994a. Comments on multimodal systems. In Moortgat (Moortgat, 1994a).

Hepple, Mark. 1994b. Labelled deduction and discontinuous constituency. In M. Abrusci, C. Casadio, and M. Moortgat, editors, *1st Rome Workshop on Linear Logic and Lambek Calculus*, Rome. OTS/Dyana.

Hepple, Mark. 1995. Hybrid categorial logics. *Journal of the Interest Group in Pure and Applied Logic (IGPL)*, 3(2):343–355. Special Issue on Deduction and Language.

Hepple, Mark and Glyn Morrill. 1989. Parsing and derivational equivalence. In *Proceedings of the 4^{th} Conference of the EACL*, pages 10–18, Manchester, UK.

Hintikka, Jaakko. 1969. *Knowledge and Belief; an introduction to the logic of the two notions.* Cornell University Press, New York.

Hodas, Joshua and Dale Miller. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365.

Howe, Denis. 1997. Free on-line dictionary of computing. http://wombat.doc.ic.ac.uk/foldoc/.

Johnson, Mark. 1991. Features and formulae. *Computational Linguistics*, 17(2):131–151.

Kapur, D and P Narendran. 1986. NP-completenes of the set unification and matching problems. In J Siekmann, editor, *Proceedings of 8th Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*. Springer-Verlag.

Kirchner, Claude. 1994. *Rewriting, Solving. Proving.* ESSLLI, European Summer School in Logic, Language and Information, Copenhagen Business School, August.

Kleene, S C. 1952. *Introduction to metamathematics*, volume 1 of *Bibliotheca mathematica: a series of monographs on pure and applied mathematics.* North-Holland Pub. co., Groningen.

Klein, E. and I Sag. 1985. Type-driven translation. *Linguistics and Philosophy*, 8:163–202.

Knuth, Donald E. and P. B. Bendix. 1983. Simple word problems in universal algebras. In J. Siekmann and G. Wrightson, editors, *Classical Papers in Computational Logic.* Springer-Verlag, New York, pages 342–376.

König, E. 1989. Parsing as natural deduction. In *Proceedings of the 27^{th} Annual Meeting of the ACL*, pages 272–279, Vancouver.

König, E. 1991. Parsing categorial grammar. In *Dyana - Dynamic Interpretation of Natural Language*, volume R1.2.C. ESPRIT, January.

König, E. 1995. Lexgram - a practical categorial grammar formalism. In *Proceedings of CLNLP95*, Edinburgh. (16 pp.) cmp-lg/9504014.

Kurtonina, Natasha and Michael Moortgat. 1995. Structural control. In Moortgat (Moortgat, 1995).

Lambek, Joachim. 1958. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170.

Lambek, Joachim. 1961. On the calculus of syntactic types. In *Proceedings of the Symposia in Applied Mathematics*, volume XII, pages 166–178, Providence, Rhode Island. American Mathematics Society.

Lambek, Joachim. 1988. Categorial and categorical grammars. In Richard Oehrle et al., editor, *Categorical Grammars and Natural Language Structures.* D. Reidel Publishing Company: Dordrecht, The Netherlands, pages 297–317.

Lambek, Joachim. 1995. Bilinear logic. In *Advances in linear logic* (Girard, Lafont, and Regnier, 1995), pages 43–59.

Lambek, Joachim and P. J. Scott. 1988. *Introduction to higher order categorical logic*, volume 7 of *Cambridge studies in advanced mathematics.* Cambridge University Press, Cambridge. ISBN 0521356539.

Leslie, Neil. 1990. Contrasting styles of categorial derivations. In Barry and Morrill (Barry and Morrill, 1990), pages 113–126.

Lincoln, Patrick, John Mitchel, Andre Scedrov, and Natarajan Shankar. 1990. Decision problems in propositional linear logic. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 662–672.

Luz, Saturnino F. 1995. Reasoning about intentions: theoretical and practical issues. In *2^{nd} Workshop on Automated Reasoning at AISB'95*, pages 31–34, Sheffield, UK.

Luz, Saturnino F. 1996a. Automated deduction and labelling: Case studies in categorial logics. *Journal of the Interest Group in Pure and Applied Logic (IGPL)*, 4(3):508–510. Conference report: 3rd WoLLIC, Salvador, Brazil.

Luz, Saturnino F. 1996b. Grammar specification in categorial logics and theorem proving. In M. McRobbie and J.K. Slaney, editors, *Proceedings of 13th Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 703–717. Springer-Verlag.

Luz, Saturnino F. 1997. Using tableaux to automate the lambek and other categorial calculi. Accepted for publication in journal *Information and Computation*, Academic Press.

Luz, Saturnino F. and Patrick Sturt. 1995. A labelled deductive theorem proving environment for categorial grammar. In *Proceedings of the IV International Workshop on Parsing Technologies*, pages 152–161, Prague, Czech Republic, September. ACL/SIGPARSE.

MacCaull, Wendy. 1997. Relational tableaux for tree models, language models and information networks. Preprint, August.

MacLane, Saunders. 1971. *Categories for the working mathematician.* Springer-Verlag, New York.

MacLane, Saunders. 1982. Why commutative diagrams coincide with equivalent proofs. *Contemporary Mathematics*, 13:387–401.

Milward, David. 1995. Incremental interpretation of categorial grammar. In 7^{th} *Conference of the European Chapter of the ACL* (Association for Computational Linguistics, 1995), pages 119–126.

Moens, Marc, Jo Calder, Ewan Klein, Mark Reape, and Henk Zeevat. 1989. Expressing generalizations in unification-based formalisms. In *Proceedings of the 4th. Conference of the European Chapter of the Association for Computational Linguistics*, pages 174–181, Manchester, England. Association for Computational Linguistics.

Montague, Richard. 1974. The proper treatment of quantification of quantification in ordinary English. In R. Thomason, editor, *Formal Philosophy: selected papers of Richard Montague*. Yale University Press, New Haven.

Moortgat, Michael. 1988. *Categorial Investigations*. Foris Publications, Dordrecht.

Moortgat, Michael. 1990a. Categorial logics: a computational perspective. In *Computer Science in the Netherlands*, Amsterdam.

Moortgat, Michael. 1990b. Unambiguous proof representations for the lambek calculus. In Martin Stokhof and Leen Torenvliet, editors, *Seventh Amsterdam Colloquium*, pages 389–401, Amsterdam. Institute for Language, Logic and Information.

Moortgat, Michael. 1992. Labelled deductive systems for categorial theorem proving. Technical Report OTS-WP-CL-92-003, OTS, Utrecht, NL.

Moortgat, Michael, editor. 1994a. *Lambek Calculus: Multimodal and Polymorphic Extensions*, Dyana Deliverable R1.1.B.

Moortgat, Michael. 1994b. Residuation in mixed Lambek systems. In *Logics of Structured Resources* (Moortgat, 1994a).

Moortgat, Michael, editor. 1995. *Logics of Structured Resources*, Dyana Deliverable R1.1.C.

Moortgat, Michael and R Oehrle. 1993. Logical parameters and linguistic variation. 5^{th} ESSLLI - University of Lisbon, August.

Morrill, Glyn. 1994. *Type logical grammar*. Kluwer Academic Publishers: Boston, MA.

Morrill, Glyn. 1995a. Clausal proofs and discontinuity. *Journal of the Interest Group in Pure and Applied Logic (IGPL)*, 3(2):403–427. Special Issue on Deduction and Language.

Morrill, Glyn. 1995b. Higher-order logic programming of categorial deduction. In *Proceedings of the 7^{th} EACL* (Association for Computational Linguistics, 1995), pages 133–140.

Morrill, Glyn, Neil Leslie, Mark Hepple, and Guy Barry. 1990. Categorial deductions and structural operations. In Barry and Morrill (Barry and Morrill, 1990), pages 1 – 21.

Pareschi, R. and M. Steedman. 1987. A lazy-way to chart-parse with categorial grammars. In *Proceedings of the 25^{th} Meeting of the ACL*, pages 81–88, Stanford.

Pentus, Mati. 1993. Lambek grammars are context free. In Robert L. Constable, editor, *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 371–373, Montreal, Canada, June. IEEE Computer Society Press.

Pentus, Mati. 1994a. The conjoinability relation in Lambek calculus and linear logic. *Journal of Logic, Language and Information*, 3(2):121–140, April.

Pentus, Mati. 1994b. Language completeness of the lambek calculus. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer science*, Paris, July. IEEE.

Peterson, Gerald and Mark Stickel. 1981. Complete sets of reductions for some equational theories. *Journal of the Association for Computing Machinery*, 28(2):233–264, April.

Pickering, Martin and Guy Barry. 1993. Dependency categorial grammar and coordination. *Linguistics*, 31(5):855–902.

Prawitz, D. 1965. *Natural Deduction. A Proof-Theoretical Study*. Almvist and Wiksell, Stockholm.

Roorda, D. 1991. *Resource Logics: Proof-theoretical Investigations*. Ph.D. thesis, University of Amsterdam.

Shieber, S., Y. Schabes, and F. Pereira. 1994. Principles and implementation of deductive parsing. Technical Report TR-11-94, Center for Research in Computing Technology, Harward University.

Siekmann, Jörg. 1989. Unification theory. *Journal of Symbolic Computation*, 7:207–274.

Smullyan, Raymond M. 1968. *First-Order Logic*, volume 43 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, Berlin.

Smullyan, Raymond M. 1969. Abstract quantification theory. In *Conference on Intuitionism and Proof Theory*, North Holland.

Steedman, Mark. 1987. Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory*, 13:207–263.

Steedman, Mark. 1990. Gapping as constituent coordination. *Linguistics and Philosophy*, 13:207–264.

Steedman, Mark. 1991. Structure and Intonation. *Language*, 68(2):260–296.

Steele, Guy L. 1990. *Common LISP : the language*. Digital Press, Bedford, Mass., 2nd. edition.

van Benthem, Johan. 1986. Categorial grammar. In *Essays in Logical Semantics*. D. Reidel Publishing Company: Dordrecht, The Netherlands, chapter 7, pages 123–150.

van Benthem, Johan. 1987. Semantic type change and syntactic recognition. In Cherchia, Partee, and Turner, editors, *Categories, Types and Semantics*. Reidel, Dordrecht.

van Benthem, Johan. 1988. Categorial grammar and type-theory. *Journal of Philosophical Logic*, 19:115–168.

van Benthem, Johan. 1991. *Language in Action: Categories, Lambdas and Dynamic Logics*. North Holland, Amsterdam.

van Benthem, Johan. 1996. *Exploring Logical Dyamics.* Studies in Logic, Language and Information. CSLI, Stanford, USA.

Venema, Yde. 1996. Tree models and (labeled) categorial grammar. *Journal of Logic, Language and Information*, 5:253–277.

Versmissen, Koen. 1994. *Grammar composition: modes, models, modalities.* OTS Dissertation Series. Research Institute for Language and Speech, Utrecht, Holland.

Wooldridge, M. and N. R. Jennings. 1995. Agent theories, architectures, and languages: A survey. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 1–39. Springer-Verlag: Heidelberg, Germany, January.

Zeevat, H., E. Klein, and J. Calder. 1987. Unification categorial grammar. In *Categorial Grammar, Unification Grammar and Parsing*, volume 1 of *Edinburgh Working Papers in Cognitive Science*. Centre for Cognitive Science, University of Edinburgh.

Zielonka, Wojciech. 1981. Axiomatizability of Ajdukiewicz-Lambek calculus by means of cancellation schemes. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, pages 215–224.