Electronic Notes in Theoretical Computer Science 53 (2005)
URL: http://www.elsevier.nl/locate/entcs/volume53.html 22 pages

A Graph-Theoretic Approach to Sequent Derivability in the Lambek Calculus

Gerald Penn [1]

Department of Computer Science
University of Toronto
10 King's College Rd.
Toronto M5S 3G4, Canada

Abstract

A graph-theoretic construction for representing the derivational side-conditions in the construction of axiomatic linkages for Lambek proof nets is presented, along with a naive algorithm that applies it to the sequent derivability problem for the Lambek Calculus. Some basic properties of this construction are also presented, and some complexity issues related to parsing with Lambek Categorial Grammars are discussed.

1 Introduction

This paper considers the question of whether a string of words can be parsed relative to a Lambek Categorial Grammar (LCG) in polynomial time. Although LCGs are known to be weakly equivalent to context-free grammars (CFG), the most relevant formal construal of this parsing question is still open, namely, "Is the sequent $A_1 \ldots A_n \vdash B$, derivable in the Lambek Calculus?," where $A_1, \ldots, A_n, B$ are (possibly complex) categories. Given an LCG, G, and a string $w_1 \ldots w_n$, with unique lexical entries, $A_1 \ldots A_n$, in G, this amounts to string recognition when $B = s$, the distinguished category of G.

A simple graph-theoretic construction for representing the well-formedness constraints of an LCG derivation, called *LC-graphs*, is presented here. An algorithm that resembles a standard CFG chart-parser is also provided for answering the above sequent derivability question. Crucially, this algorithm is not polynomial-time, so it does not solve the open problem, but it is hoped that it will contribute to an eventual solution to the

[1] Email: gpenn@cs.toronto.edu

sequent derivability problem. A few related parsing problems are also discussed.

Chart-parsing with LCGs is not a new approach. Koenig [5] proposed a chart parser augmented with meta-rules that would spawn a new chart whenever an introduction rule was applied. Hepple [4] proposed combining these charts into a single "multi-dimensional" chart. In this chart, lexical edges form a totally ordered sequence of primitive intervals, as usual, but whenever a hypothetical category is assumed, a new primitive interval is added with one free end. Introduction then amounts to abstracting the edges that use this new hypothetical interval back onto the totally ordered sequence of intervals that it was added to. Morrill [8] used a chart-like representation in combination with proof nets for the Lambek Calculus [14]. Penn [10] used a similar representation for representing deductions in the Lambek Calculus in the Elf programming language [12]. Morrill [9] also provided an actual parsing algorithm, again based on proof nets.

Proof nets have the advantage that they abstract away from all of the spurious ambiguities that arise from proof search techniques based directly on natural deduction or sequent presentations of the Lambek Calculus. As a result, they expose the essential sources of non-determinism that a worst-case complexity analysis of LCG recognition must face. Much of the recent work on parsing with LCGs, however, has chosen to dwell on elegant implementations of LCG proof search in higher-order (linear) logic programming languages. While these are indeed elegant, they are perhaps not the best choice for discovering the complexity of the problem at hand.

It is for this reason that the present article has opted for a "back-to-basics" approach, using only a few simple algorithms and basic graph theory to characterise the problem. LC-graphs represent information about substitutions associated with linkages of axiomatic formulae in the proof nets of LCG derivations. Given our extensive knowledge about algorithmic efficiency and NP-completeness in the domain of graph theory, it is hoped either that the algorithm given here can be enhanced and proven to be polynomial, or that a failure to so enhance it will reveal an embedding of a known NP-hard problem into LCG recognition.

The use of graph theory in the context of proof search in substructural logics is also not a new one. LC-graphs and their well-formedness constraints are certainly reminiscent of Girard's original "long-trip condition" [3], and later correctness criteria for multiplicative linear logic [1]. Moot and Puite [7,13] propose a graph rewriting system that encompasses LCG and all of its multimodal extensions. LC-graphs are much simpler, but their extension to multimodal LCG remains a topic for further research.

The time complexity of various LCG parsing problems is part of a broader theoretical picture that is extremely interesting in its own right. The Lambek Calculus is only one of a large number of substructural logics that have been studied to date, all of which can be related to each other by the relative presence or absence of modal operators along with structural rules of inference that control the behaviour of these operators [6]. What is not completely understood is how the presence or absence of these operators and rules affect the complexity of proof search. Just as with each of the better-known members of the Chomsky Hierarchy of formal languages we have a characterisation of that class of languages in terms of the automata and stacks required to compute string membership, an operational characterisation of this class of logics would also be extremely useful, both as a dual form of representation and as a guide for the construction of other logics with certain operational properties for some application.

Within this broader picture, the Lambek Calculus stands out as one logic of great historical interest for which the time complexity of sequent derivability is still unknown. Because of the known weak equivalence of LCGs to the context-free languages, moreover, it has a number of very interesting potential applications within computational linguistics and computational biology, where CFGs are already being used. In particular, LCGs could in principle serve as an underlying discrete structure for a context-free-equivalent statistical model that naturally exposes a very different selection of numerical parameters from those of a standard CFG, or for which certain parameters can more easily be estimated from data.

For simplicity, the presentation here will consider the product-free fragment of the Lambek calculus, in which sequents with empty premises cannot be derived. Section 2 begins with an introduction to proof nets, and how to build them. Section 3 then provides an introduction to LC-graphs and their well-formedness criteria. Section 4 gives some basic properties of LC-graphs, and shows their connection to the correctness criteria for proof nets. Section 5 shows how to combine these graphs with a simple parsing algorithm that resembles a context-free chart-parser. Section 6 then discusses the worst-case parsing complexity of a few related LCG parsing problems.

2 Proof Nets

Much of this section is taken from the excellent dissertation of Roorda [14]. The presentation here is biased towards parsing with LCGs, however.

In parsing, there are four major steps involved in constructing a proof net:

(i) create a sequence of *terminal formulae* from a candidate sequent,

(ii) *lexically unfold* the sequence of terminal formulae to a sequence of *axiomatic formulae*,

(iii) build an *axiomatic linkage* on the axiomatic formulae, and then

(iv) apply the *variable substitution* rules derived from the lexical unfolding and axiomatic linkage.

The first two steps are very quick, simple and deterministic, and they always succeed. Axiomatic linkages are where the trouble begins: this step does not always succeed, and even when it does, the variable substitution rules it creates may not be well-formed, which means that another axiomatic linkage must be found.

2.1 *Terminal Formulae*

Given a sequent of potentially complex categories, $A_1 \ldots A_n \vdash B$, we first write them as the following sequence of polarised formulae:

$$A_1^- \; A_2^- \; \ldots \; A_n^- \; B^+,$$

i.e., all premises receive negative polarity, and the consequent receives positive polarity. These are called the **terminal formulae** of the sequent. For example, beginning with the sequent:

$$(A/(A\backslash A))/A \; A \; A\backslash A \; A\backslash A \; \vdash A,$$

where A is a basic category in the grammar, we obtain the following sequence of terminal formulae:

$$((A/(A\backslash A))/A)^- \; A^- \; (A\backslash A)^- \; (A\backslash A)^- \; A^+.$$

We must also label each formula in the sequence with a variable:

$$((A/(A\backslash A))/A)^{-}\!:\!b \; A^{-}\!:\!a \; (A\backslash A)^{-}\!:\!h \; (A\backslash A)^{-}\!:\!l \; A^{+}\!:\!m$$

2.2 *Lexical Unfolding*

The next step is to transform the sequence of terminal formulae by substituting a string of simpler formulae for each formula in the sequence using the following rules, until no more substitutions can be made (in this paper $A\backslash B$ means, "looking for an A on the left to yield a B"):

$$(A\backslash B)^- \longrightarrow A^+ \; B^-$$
$$(A\backslash B)^+ \longrightarrow B^+ \; A^-$$
$$(A/B)^- \longrightarrow A^- \; B^+$$
$$(A/B)^+ \longrightarrow B^- \; A^+$$

Beginning with the sequence of terminal formulae above, we obtain the transformations:

$$
\begin{aligned}
&((A/(A\backslash A))/A)^{-}\; A^{-}\; (A\backslash A)^{-}\; (A\backslash A)^{-}\; A^{+} \;\longrightarrow\\
&(A/(A\backslash A))^{-}\; A^{+}\; A^{-}\; (A\backslash A)^{-}\; (A\backslash A)^{-}\; A^{+} \;\longrightarrow\\
&A^{-}\; (A\backslash A)^{+}\; A^{+}\; A^{-}\; (A\backslash A)^{-}\; (A\backslash A)^{-}\; A^{+} \;\longrightarrow\\
&A^{-}\; A^{+}\; A^{-}\; A^{+}\; A^{-}\; (A\backslash A)^{-}\; (A\backslash A)^{-}\; A^{+} \;\longrightarrow\\
&A^{-}\; A^{+}\; A^{-}\; A^{+}\; A^{-}\; A^{+}\; A^{-}\; (A\backslash A)^{-}\; A^{+} \;\longrightarrow\\
&A^{-}\; A^{+}\; A^{-}\; A^{+}\; A^{-}\; A^{+}\; A^{-} A^{+}\; A^{-}\; A^{+}
\end{aligned}
$$

The categories in these formulae become simpler with each rule application, so this trivially terminates, and no matter in which order the redexes of the transformations are chosen, the same final sequence results. In the final sequence, all formulae consist of polarised atomic/basic categories. This is the sequence of **axiomatic formulae**.

In general, each positive formula will be labelled with a variable, and each negative formula will be labelled with a term from the untyped lambda calculus. We assigned variables to all formulae, positive and negative, when we created the sequence of terminal formulae. During lexical unfolding, we must specify what labels to assign the formulae resulting from the transformation rules. This involves using new variables and forming new terms from these new variables and the old terms labelling the unfolded formulae. Whenever we unfold a positive formula, we must also specify an additional variable substitution as a side condition that relates the variables used by that rule:

$$
\begin{aligned}
&(A\backslash B)\overset{-}{:}t \;\longrightarrow\; A\overset{+}{:}u\; B\overset{-}{:}tu\\
&(A\backslash B)\overset{+}{:}v \;\longrightarrow\; B\overset{+}{:}v'\; A\overset{-}{:}u\; [v := \lambda u.v']\\
&(A/B)\overset{-}{:}t \;\longrightarrow\; A\overset{-}{:}tu\; B\overset{+}{:}u\\
&(A/B)\overset{+}{:}v \;\longrightarrow\; B\overset{-}{:}u\; A\overset{+}{:}v'\; [v := \lambda u.v']
\end{aligned}
$$

The above labelled sequence of terminal formulae then unfolds like this:

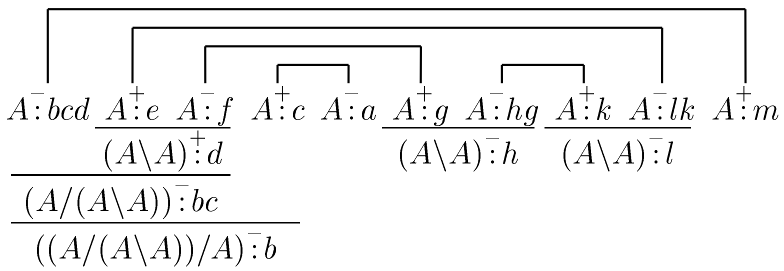
$$
\begin{aligned}
&((A/(A\backslash A))/A)\overset{-}{:}b\; A\overset{-}{:}a\; (A\backslash A)\overset{-}{:}h\; (A\backslash A)\overset{-}{:}l\; A\overset{+}{:}m \;\longrightarrow\\
&(A/(A\backslash A))\overset{-}{:}bc\; A\overset{+}{:}c\; A\overset{-}{:}a\; (A\backslash A)\overset{-}{:}h\; (A\backslash A)\overset{-}{:}l\; A\overset{+}{:}m \;\longrightarrow\\
&A\overset{-}{:}bcd\; (A\backslash A)\overset{+}{:}d\; A\overset{+}{:}c\; A\overset{-}{:}a\; (A\backslash A)\overset{-}{:}h\; (A\backslash A)\overset{-}{:}l\; A\overset{+}{:}m \;\longrightarrow\; (*)\\
&A\overset{-}{:}bcd\; A\overset{+}{:}e\; A\overset{-}{:}f\; A\overset{+}{:}c\; A\overset{-}{:}a\; (A\backslash A)\overset{-}{:}h\; (A\backslash A)\overset{-}{:}l\; A\overset{+}{:}m \;\longrightarrow\\
&A\overset{-}{:}bcd\; A\overset{+}{:}e\; A\overset{-}{:}f\; A\overset{+}{:}c\; A\overset{-}{:}a\; A\overset{+}{:}g\; A\overset{-}{:}hg\; (A\backslash A)\overset{-}{:}l\; A\overset{+}{:}m \;\longrightarrow\\
&A\overset{-}{:}bcd\; A\overset{+}{:}e\; A\overset{-}{:}f\; A\overset{+}{:}c\; A\overset{-}{:}a\; A\overset{+}{:}g\; A\overset{-}{:}hg\; A\overset{+}{:}k\; A\overset{-}{:}lk\; A\overset{+}{:}m
\end{aligned}
$$

with the substitution, $d := \lambda f.e$, arising from step $(*)$.

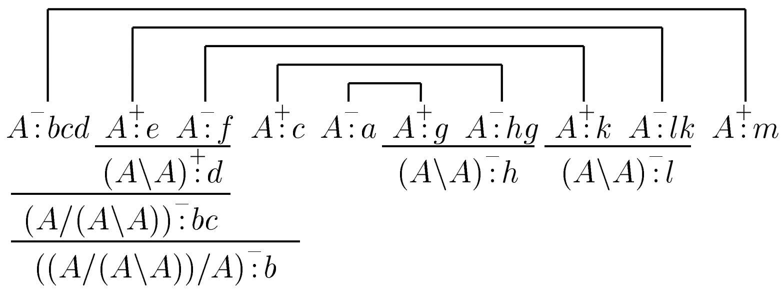
2.3 Axiomatic Linkage

After lexical unfolding, we link matching pairs of axiomatic polar formulae together (X^+ and X^-, for a basic category X). A subsequence of axiomatic formulae plus a complete matching by these links is called an **(axiomatic) linkage**. If the subsequence is the entire sequence obtained from unfolding a sequence of terminal formulae, then we distinguish it as a **spanning linkage**. Within a linkage, we can also identify **sublinkages**, contiguous subsequences whose borders every link crosses either zero or two times, i.e., no link straddles the subsequence's boundaries. Here are two spanning linkages for the above example:

(1)



(2)



Usually, we write the lexical unfolding underneath the axiomatic sequence, and the linkage above. The first spanning linkage contains a sublinkage between $A \overset{-}{:} f$ and $A \overset{+}{:} g$. The second one does not, although there is one between $A \overset{-}{:} f$ and $A \overset{+}{:} k$.

2.4 Variable Substitution

Each axiomatic link:

$$X \overset{+}{:} v \quad X \overset{-}{:} t$$

can be associated with a substitution, $v := t$. The substitutions for the two spanning linkages above are:

$$(1)\ d := \lambda f.e,\ m := bcd,\ e := lk,\ g := f,\ k := hg,\ c := a$$

$$(2)\ d := \lambda f.e,\ m := bcd,\ e := lk,\ k := f,\ c := hg,\ g := a$$

The first substitution comes from the side condition acquired during lexical unfolding.

The final step is to apply the associated substitutions iteratively to the variable labelling the positive terminal formula until no more substitutions are applicable. In our running example, this label is m:

$$(1)\; m \rightarrow bcd \rightarrow bc(\lambda f.e) \rightarrow ba(\lambda f.e) \rightarrow ba(\lambda f.lk) \rightarrow ba(\lambda f.lhg) \rightarrow ba(\lambda f.lhf)$$

$$(2)\; m \rightarrow bcd \rightarrow bc(\lambda f.e) \rightarrow bhg(\lambda f.e) \rightarrow bha(\lambda f.e) \rightarrow bha(\lambda f.lk) \rightarrow bha(\lambda f.lf)$$

2.5 Correctness Criteria for Proof Nets

A **(Lambek) proof net** consists of a lexically unfolded sequence of terminal formulae, a spanning linkage of the resulting sequence of axiomatic formulae and a variable substitution yielding a term t for which:

- **PN(1)** there is precisely one positive terminal formula,
- **PN(2)** variable substitution terminates (t is a finite term),
- **PN(3)** if t contains subterm $\lambda v.s$, then v occurs in s and does not occur outside s,
- **PN(4)** every variable assigned to a negative terminal formula occurs in t,
- **PN(CT)** t has no closed subterms, and
- **PN(L)** the axiomatic linkage is planar, i.e., the axiomatic links can be drawn as in (1) and (2) above such that no two links cross.

Roorda [14, pp. 31–34] proved that a sequent is derivable in the Lambek Calculus iff there is a proof net whose terminal formulae correspond to it.

3 LC-Graphs

Successively generating spanning linkages and testing them against the correctness criteria for proof nets is clearly not an efficient way to parse with LCGs. Ideally, what we would like is a dynamic programming method for incrementally constructing proof nets, with some way of representing the state of our knowledge about correctness and incrementally and compactly combining that knowledge too.

This section defines a graph for representing the state of our knowledge about correctness. Section 5 presents a dynamic programming method that uses these graphs. First, we need some more terminology.

Given a sequence of terminal formulae and a lexical unfolding, a variable, v, that labels a formula $X\overset{+}{:}v$ anywhere in the unfolding is called a

plus-variable. If X is a basic category, we distinguish v by saying it is a **simple plus-variable**. If X is a complex category, then we distinguish v by saying it is a **lambda-variable**. A variable, v, that labels a formula $X\overset{-}{:}v$ is a **minus-variable**. In general, negative formulae are labelled by terms, and in general these terms contain both minus-variables and plus-variables.

During lexical unfolding, a substitution is added for every lambda-variable, v, of the form $v := \lambda u.w$. In this case, u and w are called the **daughter variables** of v.

Given a sequence of axiomatic formulae, α, $V(\alpha)$ is the set of variables occurring as subterms of labels of the axiomatic formulae in α.

Given a linkage, M, over a sequence of axiomatic formulae, α, its **LC-graph** is a directed graph $G = \langle V, E\rangle$, such that V is the smallest set for which:

- $V(\alpha) \subseteq V$,
- V contains every lambda-variable with at least one daughter-variable in V,

and E is the smallest set for which:

- for every pair $u, v \in V$, if v is a lambda-variable and u is one of the daughter variables of v, then $(v, u) \in E$, and
- for every axiomatic link in M:

$$X\overset{+}{:}v \;\; X\overset{-}{:}t$$

  and for every variable u in t, $(v, u) \in E$.

When there is a path from some u to v in G, we say $u \overset{G}{\rightsquigarrow} v$, or $u \rightsquigarrow v$, when it is clear which G is meant. If $u = v$, then trivially $u \rightsquigarrow v$.

In the context of LC-graphs, when we see a plus-variable (resp. minus-variable, lambda-variable etc.), we call it a **plus-node** (resp. **minus-node**, **lambda-node** etc.). In addition, we call a node in G a **terminal node** iff it has an out-degree of 0. Note, however, that plus-nodes often occur in terms that label negative axiomatic formulae (a **negative occurrence** of a plus node), but they are still plus-nodes. These arise from the following two lexical unfolding rules:

$$(A\backslash B)\overset{-}{:}t \;\longrightarrow\; A\overset{+}{:}u\; B\overset{-}{:}tu$$
$$(A/B)\overset{-}{:}t \;\longrightarrow\; A\overset{-}{:}tu\; B\overset{+}{:}u,$$

where u labels a positive formula, and is thus a plus-node, but also occurs in the term, tu, which labels a negative formula. A minus-node corresponds to a variable that occurs *by itself* as the label of a negative formula, $A\overset{-}{:}v$. Minus-nodes never have positive occurrences.

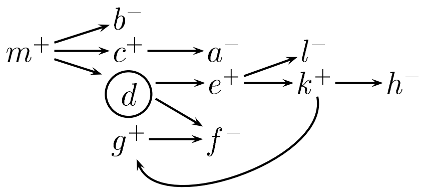
We say that a linkage, M, with LC-graph, $G = \langle V, E\rangle$, is **integral** iff:

- **I(1)** there is a unique node in G with in-degree 0, from which all other nodes are path-accessible,

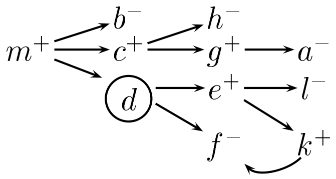
- **I(2)** G is acyclic,
- **I(3)** for every lambda-node $v \in V$, there is a path from its plus-daughter, u, to its minus-daughter, w, and
- **I(CT)** for every lambda-node $v \in V$, there is a path in G, $v \rightsquigarrow x$, where x is a terminal node and there is no lambda-node $v' \in V$ such that $v \rightsquigarrow v' \to x$.

Both of the spanning linkages presented for the example above are integral. They have the following LC-graphs:

(1)



(2)



Lambda-nodes are circled. d is a lambda-node, with daughter nodes e and f. Minus-nodes and simple plus-nodes are indicated with a sign. Note that the polarity of a node is not affected in any way by the choice of axiomatic linkage — it derives solely from the terminal sequence and its lexical unfolding.

We will abuse notation by saying that a node occurs in a sublinkage, when we mean that it occurs in the LC-graph of a sublinkage.

4 Basic Properties of LC-Graphs

LC-graphs represent the substitutions that must be carried out in the final step of proof net construction. Each plus-node corresponds to a redex for a substitution that must be replaced with a term composed of the plus-nodes and minus-nodes that it points to. With this intuition in mind, the relevance of the LC-graph integrity criteria to proof net construction can be demonstrated.

4.1 Degree

Proposition 4.1 *For every LC-graph, every plus-node either:*

(i) *labels a positive terminal formula, in which case it has an in-degree of*

0, or

(ii) *has an in-degree of at most 1.*

In a spanning linkage, the only plus-nodes with an in-degree of 0 are labels of positive terminal formulae.

Proof. Plus-nodes acquire incoming arcs either through a negative occurrence (which receives one arc from an axiom link) or through a lambda-node parent. A plus-node labelling a positive terminal formula has neither of these, and therefore has an in-degree of 0.

By inspection of the lexical unfolding rules, it can be observed that every other plus-node in a spanning linkage has either one negative occurrence or is the plus-daughter of one lambda-node, but not both. So the others have an in-degree of 1.

In a non-spanning sublinkage, it can happen that a plus-node in the sublinkage has a negative occurrence, but the negative occurrence falls outside the sublinkage. In the LC-graph for this sublinkage, this plus-node has an in-degree of 0. By definition, if a plus-daughter is in a sublinkage's LC-graph, then so is its lambda-node parent. □

Proposition 4.2 *For every LC-graph, every minus-node either:*

(i) *is a minus-daughter of a lambda-node, in which case it has an in-degree of 2, or*

(ii) *has an in-degree of 1.*

Proof. By inspection of the lexical unfolding rules, it can be observed that every minus-node appears in the label of one axiomatic formula. Since a linkage must be a complete matching of axiomatic formulae by axiom links, every minus-node receives one incoming arc from an axiom link. Minus-daughters additionally receive one incoming arc from their lambda-node parents. □

We say that a node is a **root node** in a linkage iff it has an in-degree of 0 in that linkage's LC-graph. By Proposition 4.2, root nodes are always plus-nodes. If it labels a positive terminal formula, then in deference to Proposition 4.1, we call it a **proper root node** in any linkage in which it appears. If it is a plus-node whose negative occurrence falls outside a sublinkage, then we call it an **improper root node**.

Proposition 4.3 *For every LC-graph, every minus-node has an out-degree of 0.*

Proof. All arcs point from a plus-node to either a minus-node or a plus-node. Minus-nodes correspond to labels of terminal formulae arising from sequent premises and to bound variables of lambda-terms. These are never redexes for substitution. □

Proposition 4.4 *For every LC-graph, every lambda-node has an out-degree of 1 or 2. In a spanning linkage, every lambda-node has an out-degree of 2.*

Proof. Whether a lambda-node has an out-degree of 1 or 2 depends on whether one or both of its daughters are in the linkage. If the linkage is a spanning linkage, then both of its daughters are in it. □

Proposition 4.5 *In a spanning linkage, every simple plus-node has an out-degree of at least 1, i.e., there are no terminal plus-nodes.*

Proof. Every simple plus-node labels a positive axiomatic formula. Since linkages are complete matchings of axiomatic formulae, every positive axiomatic formula has an axiom link, from which arises in the LC-graph at least one outgoing arc from its plus-node label.

By Proposition 4.4, every lambda-node has an out-degree of 1 or 2, so there are no terminal plus-nodes. □

Simple plus-nodes have unbounded out-degree. The reason for this is that simple plus-nodes owe their outgoing arcs to axiom links. Axiom links map to all of the variables that appear in the term labelling a negative axiomatic formula, and there is no limit on the number of variables in that term. We can distinguish one particular outgoing arc, however:

Proposition 4.6 *For every LC-graph, every non-terminal simple plus-node has exactly one outgoing arc to a minus-node.*

Proof. The terms that these axiom links map to contain exactly one free minus-node. This can be proven by induction on the number of lexical unfolding steps used to derive them. The ones that were created by a positive unfolding rule are minus-daughters, and therefore consist of a single minus-node. The ones that were created by a negative unfolding rule consist of a term with a lower number of lexical unfolding steps applied to a new plus-node. □

4.2 Paths

Proposition 4.7 *For every LC-graph, every path begins at a plus-node and passes exclusively through plus-nodes, terminating at either a plus-node or a minus-node.*

Proof. A trivial consequence of Proposition 4.3. □

Proposition 4.8 *If $u \leadsto v$ and $w \leadsto v$, and v is not the minus-daughter of a lambda-node, then either $u \leadsto w$ or $w \leadsto u$.*

Proof. If v is not a minus-daughter of a lambda-node, then by Proposition 4.1 and Proposition 4.2, it has an in-degree of 1. Furthermore, by Proposition 4.7, every intermediate node on the paths $u \leadsto v$ and $w \leadsto v$ is a plus-node, and by Proposition 4.1, has an in-degree of 1. □

Proposition 4.9 *Given a proper root node, there is a unique plus-node labelling an axiomatic formula which is accessible from it by a path passing exclusively through lambda-nodes.*

Proof. This is the label of the axiomatic formula obtained by following the lexical unfolding of the positive terminal formula along its positive daughters. □

We call the axiomatic label referred to in Proposition 4.9 the **axiomatic reflection** of the proper root node. If a positive terminal formula is a basic category (and thus also an axiomatic formula), then the axiomatic reflection of its proper root node is the proper root node itself.

4.3 Correctness

Proposition 4.10 *If a spanning linkage satisfies I(2), then it satisfies PN(2).*

Proof. Since the LC-graph is acyclic, its nodes can be topologically sorted. If we always choose the most highly ranked redex according to this order to expand next, then the rank of the most highly ranked redex strictly decreases at each step of variable substitution, and thus variable substitution terminates. □

Proposition 4.11 *If a spanning linkage satisfies I(2) and I(3), then it satisfies PN(3).*

Proof. Given I(3), there is likewise an occurrence of v in s. If v also occurred outside s, then there would be a path from some plus-node $w \leadsto v$, such that neither $u \leadsto w$ (and thus expands to a subterm of s) nor $w \leadsto u$ (with $\lambda v.s$ being a subterm of the expansion of w).

By Proposition 4.2, the in-degree of v is 2, with one arc coming from its lambda-node, l, and one arc coming via an axiom link from some other plus node, x. If the path $w \leadsto v$ were via l, then there would be a path $w \leadsto l \rightarrow u$ as well.

If the path $u \leadsto v$ passed through l, then there would be a cycle $u \leadsto l \rightarrow u$, which is excluded by I(2). Thus the path $u \leadsto v$ passes through x. Since x is not a minus-node, then by Proposition 4.8, if the path from w to v passes through x, then there is either a path $u \leadsto w \leadsto x \rightarrow v$ or a path $w \leadsto u \leadsto x \rightarrow v$, which contradicts our assumption. □

Proposition 4.12 *If a spanning linkage satisfies I(1), then it satisfies PN(1) and PN(4).*

Proof. If there is a unique node with in-degree 0, then by Proposition 4.1, there is a unique positive terminal formula. All nodes are path-accessible from the unique proper root node, including the minus-nodes. Since variable substitution begins with this proper root node, and since, by Proposition 4.3, every minus-node has out-degree 0, applying all substitutions until no more can be applied will result in a term that contains every minus-node, including all of the labels of negative terminal formulae. □

Proposition 4.13 *If a spanning linkage satisfies PN(3), then it satisfies I(3).*

Proof. Given PN(3), each subterm $\lambda v.s$ corresponds to a lambda-node with a minus-daughter v, and a plus-daughter u, whose expansion under variable substitution yields s. The occurrence of v in s then means that there is a path $u \rightsquigarrow v$. □

Proposition 4.14 *If a spanning linkage satisfies PN(1) and PN(2), then no node that is path-accessible from the label of the positive terminal formula is contained in a cycle.*

Proof. If a node v that is path-accessible from the label of the (unique) positive terminal formula were contained in a cycle, then variable substitution, which begins at that label, would not terminate, since the redex corresponding to n would expand to a term containing v. □

Proposition 4.15 *If a spanning linkage satisfies PN(1), PN(2), PN(3), and PN(4), then it satisfies I(1).*

Proof. If the spanning linkage satisfies both PN(1) and PN(4), then since variable substitution begins with the label of the unique positive terminal formula, and results in a term containing all of the minus-variables labelling the negative terminal formulae, then the corresponding minus-nodes must be path-accessible from the corresponding proper root node, which has in-degree 0.

The axiomatic reflection of the proper root node, and all of the lambda-nodes in between are accessible from the proper root node. Each of these lambda-nodes has a minus-daughter, and these are likewise accessible, since there is a path from every lambda-node to both of its daughters.

All other nodes in the LC-graph were introduced at a location in the lexical unfolding with a negative formula somewhere beneath them. The proof that these remaining nodes are path-accessible from the proper root node is by induction on the number of lexical unfolding steps down to the lowest such negative formula. In order for the induction to carry through, we must strengthen the claim by adding the condition that for minus-nodes, there exists a path whose last step arises from an axiom link.

The base cases are precisely the minus-nodes labelling negative terminal formulae and the minus-daughters of lambda-nodes along the path from the proper root to its axiomatic reflection. For the former category, the last step must be an axiom link because the negative terminal formulae are not daughters of a lambda-node. For the latter category, I(3) holds by Proposition 4.13, so each of these minus-daughters, m, has a plus-daughter sister, p, such that there is a path $p \rightsquigarrow m$. The last step of this path must be due to an axiom link, since the lambda-node parent of p and

m lies on the path from the proper root node to its axiomatic reflection, and none of the nodes on this path have negative occurrences.

Suppose the last unfolding step is a positive unfolding:

$$(A\backslash B)\overset{+}{:}v \;\longrightarrow\; B\overset{+}{:}v'\; A\overset{-}{:}u$$

$$(A/B)\overset{+}{:}v \;\longrightarrow\; B\overset{-}{:}u\; A\overset{+}{:}v'$$

By the inductive hypothesis, there is a path to the lower lambda-node, v and thus a path to the plus-daughter, v'. By I(3), there is also a path from v' to u. By Proposition 4.2, the in-degree of u is 2, but if the last step of the path $v' \rightsquigarrow u$ were via $v \to u$, then there would be a cycle $v' \rightsquigarrow v \to v'$. By Proposition 4.14, this is a contradiction, since v is path-accessible from the proper root node. So the last step of the path $v' \rightsquigarrow u$ arises from an axiom link.

Otherwise, the last unfolding step is a negative unfolding:

$$(A\backslash B)\overset{-}{:}t \;\longrightarrow\; A\overset{+}{:}u\; B\overset{-}{:}tu$$

$$(A/B)\overset{-}{:}t \;\longrightarrow\; A\overset{-}{:}tu\; B\overset{+}{:}u$$

By the inductive hypothesis, there is a path to the minus node in t whose last step arises from an axiom link. Since t does not label an axiomatic formula, the link must attach to a negative formula containing tu, and thus there is a path to u as well. □

Proposition 4.16 *If a spanning linkage satisfies I(1) and PN(2), then it satisfies I(2).*

Proof. By Proposition 4.14, no node that is path-accessible from the label positive terminal formula is contained in a cycle, and by I(1), every node is so accessible. □

Proposition 4.17 *If a spanning linkage satisfies I(2) and I(3), then it satisfies I(CT) iff it satisfies PN(CT).*

Proof. Given I(CT), it is claimed that no lambda-node expands under variable substitution to a closed term. Given a lambda-node v, with minus-daughter w, there is a path $v \rightsquigarrow x$, where x is some terminal node for which there is no lambda-node $v' \in V$ such that $v \rightsquigarrow v' \to x$. By Proposition 4.5, x is a minus-node.

If x is not the minus-daughter of a lambda-node, then x corresponds to the label of a negative terminal formula, and so v trivially does not expand to a closed term. Otherwise, suppose that every such x is the minus-daughter of a lambda-node, i.e., corresponds to the bound variable of some lambda-term. Let v' be its lambda-node, and u' be the plus-daughter of v'. $x \neq w$, and $v' \neq v$, or else trivially $v \rightsquigarrow v' \to x$. By I(3), there is a path $v' \to u' \rightsquigarrow y \to x$, where $y \neq v'$, or else there is a cycle, which contradicts I(2).

By Proposition 4.2, x has an in-degree of 2, so either $v \rightsquigarrow v' \to x$, which contradicts our choice of x, or $v \rightsquigarrow y \to x$. In the latter case, by Proposition 4.8, $v' \rightsquigarrow v$. Thus v expands to a term that contains a free instance of x, and so is not closed.

Given PN(CT), no lambda-term l is a closed term, so there must be a path from its corresponding lambda-node v to a minus-node other than minus-daughters of the lambda-nodes that are reachable from v. Thus I(3) holds. □

PN(L) will be discussed in Section 5.

4.4 Non-spanning linkages

While many of the above results pertain only to spanning linkages, there are a few remarks we can make about linkages in general. As mentioned above, it can happen that a plus-node occurs in a linkage, but its negative occurrence falls outside the linkage's boundaries. These are improper root nodes. The reverse can also happen: a negative occurrence of a plus-node occurs in a linkage, but the positive occurrence falls outside. These are terminal plus-nodes. A spanning linkage is a special case in which there is only one root node — a proper one — and there are no terminal plus-nodes. In addition, lambda-nodes in a sublinkage can have an out-degree of either 1 or 2, depending on how many of their daughters fall outside.

Proposition 4.18 *In any linkage satisfying I(2), there is at least one root node, and every node is path-accessible from at least one root node.*

Proof. A trivial consequence of acyclicity and the fact that LC-graphs have finitely many nodes. □

Proposition 4.19 *In any linkage, every node is path-accessible from at most one root node, except minus-daughters of lambda-nodes.*

Proof. A trivial consequence of Proposition 4.8. □

We call a terminal formula, T, **peripheral** in a sublinkage iff the rightmost or leftmost axiomatic formula in the sublinkage's axiomatic sequence derives from T's lexical unfolding. We call a node **peripheral** in a sublinkage iff it derives from the lexical unfolding of a peripheral terminal formula. This obviously includes the nodes that appear in the term labelling the rightmost or leftmost axiomatic formulae in the sublinkage, but it may include more.

The following is an interesting characterisation of root nodes and terminal plus-nodes for the purposes of sequent derivability:

Proposition 4.20 *If a sequence of terminal formulae has exactly one positive terminal formula, and it is the rightmost terminal formula, then in any of that sequence's sublinkages, every root node and terminal plus-node is peripheral.*

Proof. Terminal plus-nodes and improper root nodes have a negative (resp. positive) occurrence inside a given sublinkage, and a positive (resp. negative) occurrence outside the sublinkage. But the positive and negative occurrences of a node, when both exist, necessarily derive from the same lexical unfolding, i.e., the lexical unfolding of the same terminal formula. That means that this lexical unfolding must be peripheral, since one occurrence, and thus part of the unfolding, falls outside the sublinkage.

Proper root nodes are different — they do not have negative occurrences anywhere. By assumption, however, the proper root node labels the rightmost terminal formula, so when they occur in a sublinkage, they are necessarily part of the rightmost unfolding. □

5 Building Spanning Linkages

The definition of LC-graphs themselves does not shed any light on parsing complexity if we simply use them to check variable substitution after building a spanning linkage. What we really need is a method for building linkages that allows us to check the integrity of the associated LC-graphs incrementally.

To assist us in this task, we can use the one remaining proof net correctness criterion, PN(L), which requires that axiomatic linkages can be drawn as a planar graph above the sequence of axiomatic formulae. This is exactly how the bracketing of a string according to a context-free grammar must look. So we can use a chart parser over a fixed grammar to tabulate linkages and their LC-graphs:

$$
\begin{aligned}
&1)\ L \rightarrow B\,L\\
&2)\ B \rightarrow X^-\,L\,X^+,\ \text{for every basic category X}\\
&3)\ B \rightarrow X^+\,L\,X^-,\ \text{for every basic category X}\\
&4)\ B \rightarrow X^-\,X^+,\ \text{for every basic category X}\\
&5)\ B \rightarrow X^+\,X^-,\ \text{for every basic category X}\\
&6)\ L \rightarrow B
\end{aligned}
$$

L corresponds to those subsequences of axiomatic formulae over which a sublinkage exists. B corresponds to those subsequences over which a sublinkage bracketed by a single axiom link exists. Any spanning linkage or sublinkage constructed with this grammar will satisfy PN(L). In order to satisfy the other correctness criteria, we must describe how each rule should combine the LC-graphs of the edges on its right-hand side.

5.1 Base Cases: Rules (4) and (5)

For these rules, the resulting linkage consists of a single axiom link — there are no other LC-subgraphs to combine. The LC-graph for this link is the smallest graph consisting of:

- a set of arcs, each emanating from the plus-node labelling X^+, with one arc mapping to each node in the label of X^-, and
- for each daughter node in the LC-graph, an arc from its parent lambda-node to itself.

5.2 Bracketing: Rules (2) and (3)

Here, the resulting linkage has an LC-graph which is the smallest graph consisting of the above two classes of arcs plus:

- the LC-graph of the right-hand-side category, L.

5.3 Adjunction: Rule (1)

For rule (1), the resulting linkage has an LC-graph which is the smallest graph consisting of:

- the LC-graph of the right-hand-side category, B,
- the LC-graph of the right-hand-side category, L, and
- for each daughter node in the LC-graph, an arc from its parent lambda-node to itself.

5.4 Trivial Adjunction: Rule (6)

In rule (6), the LC-graph of the result is the same as the LC-graph of the right-hand-side category, B.

5.5 Parsing Complexity

Chart-parsing with edges that have attached LC-graphs violates a very important invariant of context-free chart-parsing, namely that any two edges covering the same subsequence can be treated as equals regardless of how they were derived. Here, either through two different adjunctions or through an adjunction and a bracketing, it is possible to obtain two edges covering the same subsequence with different LC-graphs. No method is known for treating these edges as equals, although see Section 7 for further discussion of this point. This means that before adding a new edge, we must check for an existing edge covering the same interval and combine their LC-graphs into a set.

While the number of edges that can be added to the chart is still quadratic in the length of the sequence of axiomatic formulae (which in turn grows with the length of an initial sequent), combining two existing edges by Adjunction involves combining all possible pairs of LC-graphs in their respective sets, with the result that the size of these sets may grow exponentially as a function of the interval covered by an edge. So this algorithm is not polynomial-time in the worst case.

5.6 Incremental Enforcement of Integrity Criteria

In spite of its exponential worst-case complexity, this algorithm does permit some degree of incrementality in the enforcement of the integrity criteria for LC-graphs.

Among the three integrity criteria, I(2) stands out because it demands that a particular kind of path, namely a cycle, does not exist, whereas I(3) and I(CT) demand that a particular kind of path does exist. This makes I(2) easy to enforce incrementally. Before asserting an edge, we simply discard the the LC-graphs with cycles from its set. If no LC-graphs remain, then the edge itself can be discarded.

I(3) and I(CT) can be enforced incrementally to a lesser extent with LC-graphs, although see Section 7 for further discussion of this point. If no terminal plus-node is path-accessible from the plus-daughter of a lambda-node, and that plus-daughter does yet have the paths required by I(3) and I(CT), then it will never have them, and the LC-graph can be discarded. If the required paths already exist, then they will never disappear, so this plus-daughter does not need to be checked again.

I(1) cannot be checked incrementally, but:

Proposition 5.1 *If a spanning linkage satisfies I(2) and PN(1), then it satisfies I(1).*

Proof. Given PN(1), there is only one node with in-degree 0. Since there are no cycles, every node must be path-accessible from that node. □

PN(1) can be enforced at the outset by ensuring that the sequence of terminal formulae has only one positive formula. This means that provided we are checking I(2), we can ignore I(1).

6 Related Parsing Problems

It is important to realise that the sequent derivability decision problem is only one aspect of parsing with LCGs. There are other sources of complexity, and other restrictions that can be made, which can affect the complexity of the overall problem.

6.1 Fixing the Grammar

Pentus [11] proved that LCGs are weakly equivalent to CFGs by showing how to construct an equivalent CFG from any LCG. The size of the resulting CFG is exponentially larger than the original LCG in the worst case, so this construction cannot be used to establish a polynomial-time bijection between the two classes of parsing problems.

On the other hand, if we fix a particular LCG, G, apply Pentus's construction off-line, and then ask "Given a string of words w, does w belong to $L(G)$?," then this fixed LCG recognition problem is polynomial-time because CFG recognition can also be performed in polynomial-time. This is described in detail by Finkel and Tellier [2].

6.2 Lexical Ambiguity

Given a string of words, w, and a grammar, G (unfixed), if we want to know whether w belongs to $L(G)$, we must first find the categories associated with each word of w by the lexicon of G before asking the sequent derivability question. It is often the case, however, that there is more than one category associated with the words of w. Even if the sequent derivability problem should turn out to be solvable in polynomial time, iteratively choosing categories and performing a sequent derivability check could lead to 2^n possible category selections given a string of n words. At present, no method is known for somehow combining multiple lexical categories in the course of a single proof search, although it is likely that one exists.

6.3 Parsing vs. Recognition

Sequent derivability, just as context-free chart parsing, asks a yes-or-no question. In the context of CFG parsing, this is known as the string recognition problem. True CFG parsing involves not just determining whether a string is parseable, but providing the parse trees for the string if it is. In the worst case, this takes exponential time, because although a parsing chart can be built in polynomial time, unpacking the chart and enumerating each tree contained in it can take exponentially long.

The analogue in the case of sequent derivability is to provide an actual semantic term for a derivation rather than just 'yes' or 'no.' This term is the one obtained after variable substitution is performed on the label annotating the rightmost (positive) terminal formula. Just as there are inherently ambiguous strings in some CFGs, there are also sequents with inherently more than one semantic reading in the Lambek Calculus. In fact, the number of readings can grow exponentially as a function of the length of the candidate sequent.

Define S_i to be the following family of sequents, parametrised by i:

$$\{(A/(A\backslash A)/A){:}b_i\}^i \; A{:}a \; \{A\backslash A{:}h_i \; A\backslash A{:}l_i\}^i \;\vdash\; A{:}m$$

For example, S_2 is the sequent:

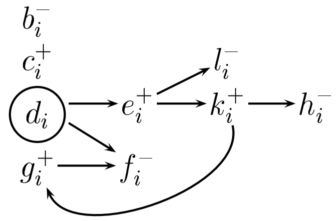
$$(A/(A\backslash A)/A){:}b_1\ (A/(A\backslash A)/A){:}b_2\ A{:}a\ A\backslash A{:}h_1\ A\backslash A{:}l_1\ A\backslash A{:}h_2\ A\backslash A{:}l_2\ \vdash\ A{:}m$$

Proposition 6.1 *For all natural numbers i, the sequent S_i is derivable in the Lambek Calculus, with semantic readings given by strings:*

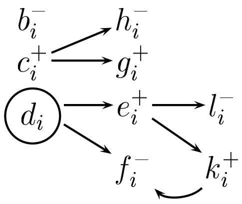
$$b_1x_1\ldots b_ix_ia(\lambda f_i.l_iy_if_i)\ldots(\lambda f_1.l_1y_1f_1)$$

where for all $1 \leq j \leq i$, either ($x_i = \epsilon$ and $y_i = h_i$), or ($x_i = h_i$ and $y_i = \epsilon$).

Proof. Let G_1^i be the graph:



and G_2^i be the graph:



and let $\text{tail}_1(i) = c_i$ and $\text{tail}_2(i) = g_i$. It can be seen that G_1^i and G_2^i internally possess the paths required by I(3) and I(CT) for the lambda-nodes that they contain, and that both are acyclic. Let $j_1 \ldots j_i$ be a sequence of 1s and 2s, and let G be the smallest graph containing the following subgraphs and arcs:

$G_{j_1}^1, G_{j_2}^2, \ldots, G_{j_i}^i,$

$m \to b_1, m \to c_1, m \to d_1, \text{tail}_{j_i}(i) \to a,$

and for all $1 \leq k \leq i-1$,

$\text{tail}_{j_k}(k) \to b_{k+1}, \text{tail}_{j_k}(k) \to c_{k+1}, \text{tail}_{j_k}(k) \to d_{k+1}.$

Regardless of the choice of $j_1 \ldots j_i$, G is acyclic because all of the $G_{j_k}^k$ are acyclic for $1 \leq k \leq i$, and no arc traverses from a $k+1$-indexed node to a lesser-indexed node for any $1 \leq k \leq i-1$. Also, any node from which all three of b_k, c_k and d_k are path-accessible has access to every node of $G_{j_k}^k$, including $\text{tail}_{j_k}(k)$, so every node of G is path-accessible from m. Planar linkages exist for G for any choice of $j_1 \ldots j_i$, roughly similar to those given for the running example in Section 2 of this paper, which is actually S_1. When $j_k = 1$, $y_k = h_k$ and $x_k = \epsilon$. When $j_k = 2$, $x_k = h_k$ and $y_k = \epsilon$. □

The choices of j_k are mutually independent, so there are 2^i possible readings for S_i. Notice that this is stated purely in terms of sequents, so the exponential blow-up is independent of lexical ambiguity.

7 Future Work

Clearly, the most significant remaining question is how to combine LC-graphs during parsing so as to avoid an exponential explosion in the size of the set of LC-graphs attached to each edge. One partial step in this direction would be the conjecture that if two LC-graphs for the same edge can both produce an integral spanning linkage, then both of them share the same set of paths from improper root nodes to terminal plus-nodes. In other words, they behave the same way when considering only their peripheral formulae. By itself, this piece of knowledge cannot be used to prune away LC-graphs of sublinkages, however, because, given two LC-graphs with different sets of paths from improper root nodes to terminal plus-nodes, it does provide a way of determining which set is the correct one.

Another possible line of enquiry would be to develop a related graph structure to enforce I(3) and I(CT) at a finer grain of incrementality, much as LC-graphs work for I(2). This might involve, for example, switching the direction of the arc from a lambda-node to its minus-daughter so that these path existence conditions would become cycle existence conditions, and then using some sort of complement graph structure.

References

[1] Danos, V. and L. Regnier, *The structure of multiplicatives*, Archive for Mathematical Logic **28** (1989), pp. 181–203.

[2] Finkel, A. and I. Tellier, *A polynomial algorithm for the membership problem with categorial grammars*, Theoretical Computer Science (1996), pp. 207–221.

[3] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **56** (1987), pp. 1–102.

[4] Hepple, M., *Chart parsing lambek grammars: Model extensions and incrementatlity*, in: *Proceedings of the 14th International Conference on Computational Linguistics*, 1992.

[5] Koenig, E., *The complexity of parsing with extended categorial grammars*, in: *Proceedings of the 13th International Conference on Computational Linguistics*, 1990.

[6] Kurtonina, N. and M. Moortgat, *Structural control*, in: P. Blackburn and M. de Rijke, editors, *Specifying Syntactic Structures*, CSLI Publications, 1996 .

[7] Moot, R. and Q. Puite, *Proof nets for multimodal categorial grammars*, in: G.-J. M. Kruijff and R. T. Oehrle, editors, *Proceedings of Formal Grammar 1999*, 1999.

[8] Morrill, G., *Higher-order linear logic programming of categorial deduction*, in: *Proceedings of the 7th Conference of the European Chapter of the Association for Computational Linguistics*, 1995.

[9] Morrill, G., *Memoisation of categorial proof nets: parallelism in categorial processing*, Technical Report LSI-96-24-R, Dept. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya (1996).

[10] Penn, G., *On the connection between context-free grammars and lambek categorial grammars*, Technical Report CMU-LCL-96-2, Laboratory for Computational Linguistics, Carnegie Mellon University (1996).

[11] Pentus, M., *Lambek grammars are context free*, Technical report, Dept. of Mathematical Logic, Moscow University (1992).

[12] Pfenning, F., *Logic programming in the lf logical framework*, in: G. Hyet and G. Plotkin, editors, *Logical Frameworks*, Cambridge University Press, 1991 .

[13] Puite, Q. and R. Moot, *Proof nets for the multimodal lambek calculus* (1999), unpublished ms., Rijksuniversiteit Utrecht.

[14] Roorda, D., "Resource Logics: Proof-theoretical Investigations," Ph.D. thesis, Universiteit van Amsterdam (1991).