*Electronic Notes in Theoretical Computer Science 53 (2003)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume53.html*](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 pre- sented, and some complexity issues related to parsing with Lambek Categorial Grammars are discussed.

# 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 gram- mars (CFG), the most relevant formal construal of this parsing question is still open, namely, ”Is the sequent *A1 ::: An ▶ B*, derivable in the Lambek Calculus?,” where *A1 ;::: ; An;B* are (possibly complex) categories. Given an LCG, *G*, and a string *w1 ::: wn* , with unique lexical entries, *A1 ::: An* , in *G*, this amounts to string recognition when *B = s*, the distinguished cate- gory of *G*.

A simple graph-theoretic construction for representing the well-form- edness 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 prob- lem, but it is hoped that it will contribute to an eventual solution to the

*1* Email: [*gpenn@cs.toronto.edu*](mailto:gpenn@cs.toronto.edu)

*◯*c 2003 Published by Elsevier Science B. V. *Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

sequent derivability problem. A few related parsing problems are also dis- cussed.

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 combin- ing these charts into a single “multi-dimensional” chart. In this chart, lex- ical 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 ofintervals that it was added to. Morrill [8] used a chart-like rep- resentation 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 proofnets.

Proof nets have the advantage that they abstract away from all of the spurious ambiguities that arise from proof search techniques based di- rectly on natural deduction or sequent presentations of the Lambek Cal- culus. As a result, they expose the essential sources of non-determinism that a worst-case complexity analysis ofLCG 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 the- ory 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 algorith- mic 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 embed- ding ofa known NP-hard problem into LCG recognition.

The use ofgraph theory in the context ofproofsearch in substructural logics is also not a new one. LC-graphs and their well-formedness con- straints are certainly reminiscent ofGirard’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 ofvarious LCG parsing problems is part ofa broad- er theoretical picture that is extremely interesting in its own right. The Lambek Calculus is only one ofa large number ofsubstructural logics that

have been studied to date, all of which can be related to each other by the relative presence or absence ofmodal operators along with structural rules ofinference that control the behaviour ofthese operators [6]. What is not completely understood is how the presence or absence ofthese opera- tors and rules affect the complexity of proof search. Just as with each of the better-known members ofthe Chomsky Hierarchy of formal languages we have a characterisation ofthat class of languages in terms ofthe automata and stacks required to compute string membership, an operational char- acterisation 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 in- teresting potential applications within computational linguistics and com- putational 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 frag- ment 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 prop- erties of LC-graphs, and shows their connection to the correctness crite- ria for proof nets. Section 5 shows how to combine these graphs with a simple parsing algorithm that resembles a context-free chart-parser. Sec- tion 6 then discusses the worst-case parsing complexity of a few related LCG parsing problems.

# Proof Nets

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

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

1. create a sequence of *terminal formulae* from a candidate sequent,
2. *lexically unfold* the sequence of terminal formulae to a sequence of

*axiomatic formulae*,

1. build an *axiomatic linkage* on the axiomatic formulae, and then
2. apply the *variable substitution* rules derived from the lexical unfold-

ing and axiomatic linkage.

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

* 1. *Terminal Formulae*

Given a sequent of potentially complex categories, *A1 ::: An ▶ B*, we first write them as the following sequence of polarised formulae:

*A A*

*::: A B+;*

*1 2 n*

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\A))=AA A\A A\A ▶ A;*

where *A* is a basic category in the grammar, we obtain the following se- quence ofterminal formulae:

*((A=(A\A))=A) A (A\A) (A\A) A+:*

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

*+*

*((A=(A\A))=A) : b A : a (A\A) : h (A\A) : l A : m*

* 1. *Lexical Unfolding*

The next step is to transform the sequence of terminal formulae by substi- tuting 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\B* means, ”looking for an *A* on the left to yield a *B*”):

*(A\B) −→ A+ B (A\B)+ −→ B+ A (A=B) −→ A B+ (A=B)+ −→ B A+*

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

*((A=(A\A))=A) A (A\A) (A\A) A+ −→ (A=(A\A)) A+ A (A\A) (A\A) A+ −→ A (A\A)+ A+ A (A\A) (A\A) A+ −→*

*A A+ A A+ A (A\A) (A\A) A+ −→ A A+ A A+ A A+ A (A\A) A+ −→ A A+ A A+ A A+ A A+ A A+*

The categories in these formulae become simpler with each rule applica- tion, so this trivially terminates, and no matter in which order the redexes ofthe 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 lamb- da calculus. We assigned variables to all formulae, positive and negative, when we created the sequence of terminal formulae. During lexical un- folding, 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 un- folded 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:

*+*

*(A\B) : t −→ A : uB : tu*

*+ +*

*(A\ −→ 0 0*

*B) : v*

*B v*

*A : u [v := u:v ]*

*+*

*(A=B) : t −→ A : tu B : u*

*+ + 0 0*

*(A=B) : v −→ B : uA : v [v := u:v ]*

The above labelled sequence ofterminal formulae then unfolds like this:

*+*

*((A=(A\A))=A) : b A : a (A\A) : h (A\A) : l A : m −→*

*+ +*

*(A=(A\A)) : bc A : c A : a (A\A) : h (A\A) : l A : m −→*

*+ + +*

*A : bcd (A\A) : dA : c A : a (A\A) : h (A\A) : l A : m −→ (∗)*

*+ + +*

*A : bcd A : e A : f A : c A : a (A\A) : h (A\A) : l A : m −→*

*+ + + +*

*A : bcd A : e A : f A : c A : aA : g A : hg (A\A) : l A : m −→*

*+ + + + +*

*A : bcd A : e A : f A : c A : aA : g A : hg A : k A : lk A : m*

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

* 1. *Axiomatic Linkage*

After lexical unfolding, we link matching pairs of axiomatic polar formulae together (*X+* and *X* ,for a basic category *X*). A subsequence ofaxiomatic formulae plus a complete matching by these links is called an **(axiomatic) linkage**. If the subsequence is the entire sequence obtained from unfold- ing a sequence ofterminal 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 span- ning linkages for the above example:

*(1)*

*+ + + + +*

*A : bcd A : e A : f A : c A : a A : g A : hg A : k A : lk A : m*

*+*

*(A\A) : d (A\A) : h (A\A) : l*

*(A=(A\A)) : bc*

*((A=(A\A))=A) : b*

*(2)*

*+ + + + +*

*A : bcd A : e A : f A : c A : a A : g A : hg A : k A : lk A : m*

*+*

*(A\A) : d (A\A) : h (A\A) : l*

*(A=(A\A)) : bc*

*((A=(A\A))=A) : b*

Usually, we write the lexical unfolding underneath the axiomatic sequence, and the linkage above. The first spanning linkage contains a sublinkage

*+*

between *A : f* and *A : g*. The second one does not, although there is one

*+*

between *A : f* and *A : k*.

* 1. *Variable Substitution*

Each axiomatic link:

*+*

*X : v X : t*

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

*(1) d := f:e; m := bcd; e := lk; g := f; k := hg; c := a*

*(2) d := f:e; m := bcd; e := lk; k := f; c := hg; g := a*

The first substitution comes from the side condition acquired during lex- ical unfolding.

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

*(1) m → bcd → bc( f:e) → ba( f:e) → ba( f:lk) → ba( f:lhg)*

*→ ba( f:lhf )*

*(2) m → bcd → bc( f:e) → bhg( f:e) → bha( f:e) → bha( f:lk)*

*→ bha( f:lf )*

* 1. *Correctness Criteria for Proof Nets*

A **(Lambek) proof net** consists of a lexically unfolded sequence of termi- nal 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 *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.

# 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 ofour knowledge about correctness and incrementally and com- pactly combining that knowledge too.

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

Given a sequence of terminal formulae and a lexical unfolding, a vari- able, *v*, that labels a formula *X : 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 labelsaformula

*X 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-var- iable, *v*, of the form *v := u:w*. In this case, *u* and *w* are called the **daughter variables** of *v*.

Given a sequence ofaxiomatic formulae,, *V ( )* is the set ofvariables occurring as subterms oflabels ofthe axiomatic formulae in.

Given a linkage, *M* , over a sequence of axiomatic formulae,, its **LC- graph** is a directed graph *G = ⟨V; E⟩*, such that *V* is the smallest set for which:

*V ( ) ⊆ 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 ∈ V* , if *v* is a lambda-variable and *u* is one of the daughter variables of *v*, then *(v; u) ∈ E*, and

for every axiomatic link in *M* :

*+*

*X : v X : t*

and for every variable *u* in *t*, *(v; u) ∈ E*.

When there is a path from some *u* to *v* in *G*, we say *u ; v*, or *u ; v*, when it is clear which *G* is meant. If *u = v*, then trivially *u ; v*.

*G*

In the context ofLC-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\B) : t −→ A : uB : tu*

*+*

*(A=B) : t −→ A : tu B : u;*

where *u* labels a positive formula, and is thus a plus-node, but also oc- curs in the term, *tu*, which labels a negative formula. A minus-node corre- sponds to a variable that occurs *by itself* as the label ofa negative formula,

*A : v*. Minus-nodes never have positive occurrences.

We say that a linkage, *M* , with LC-graph, *G = ⟨V; E⟩*, 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 ∈ V* , there is a path from its plus-daughter,

*u*, to its minus-daughter, *w*, and

**I(CT)** for every lambda-node *v ∈ V* , there is a path in *G*, *v ; x*, where x is a terminal node and there is no lambda-node *v0 ∈ V* such that *v ; v0 → x*.

Both ofthe spanning linkages presented for the example above are in- tegral. They have the following LC-graphs:

*(1)*

*h*



*b*

*m+ c+ d*

*g+*

*a l*

*e+ k+*

*f*

*(2)*

*m+*

*h*

*g+ * *a*



*b*

*c+ d*

*e+ l*

*f k+*

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 ax- iomatic linkage — it derives solely from the terminal sequence and its lex- ical unfolding.

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

# Basic Properties of LC-Graphs

LC-graphs represent the substitutions that must be carried out in the final step ofproof 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.

* 1. *Degree*

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

* + 1. *labels a positive terminal formula, in which case it has an in-degree of*

*0, or*

* + 1. *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 occur- rence (which receives one arc from an axiom link) or through a lambda- node parent. A plus-node labelling a positive terminal formula has neither ofthese, and therefore has an in-degree of0.

By inspection ofthe lexical unfolding rules, it can be observed that ev- ery other plus-node in a spanning linkage has either one negative occur- rence or is the plus-daughter of one lambda-node, but not both. So the others have an in-degree of1.

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. *2*

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

1. *is a minus-daughter of a lambda-node, in which case it has an in- degree of 2, or*
2. *has an in-degree of 1.*

**Proof.** By inspection ofthe 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. *2*

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. Ifit labels a positive terminal formula, then in deference to Propo- sition 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 ofterminal formulae arising from sequent premises and to bound variables of lambda-terms. These are never redexes for substitution. *2*

**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 ofits daughters are in it. *2*

**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 ofaxiomatic formulae, every positive ax- iomatic 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. *2*

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 ofthe variables that appear in the term labelling a negative ax- iomatic 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 pos- itive unfolding rule are minus-daughters, and therefore consist of a single minus-node. The ones that were created by a negative unfolding rule con- sist of a term with a lower number of lexical unfolding steps applied to a new plus-node. *2*

* 1. *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 ofProposition 4.3. *2*

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

**Proof.** If *v* is not a minus-daughter of a lambda-node, then by Proposi- tion 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 ; v* and *w ; v* is a plus-node, and by Proposition 4.1, has an in-degree of1. *2*

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

**Proof.** This is the label ofthe axiomatic formula obtained by following the lexical unfolding of the positive terminal formula along its positive daugh- ters. *2*

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

* 1. *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 de- creases at each step of variable substitution, and thus variable substitu- tion terminates. *2*

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

**Proof.** Given I(3), there is likewise an occurrence of *v* in *s*. If *v* also oc- curred outside *s*, then there would be a path from some plus-node *w ; v*, such that neither *u ; w* (and thus expands to a subterm of *s*) nor *w ; u* (with *v:s* being a subterm ofthe 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 ; v* were via *l*, then there would be a path *w ; l → u* as well.

If the path *u ; v* passed through *l*, then there would be a cycle *u ; l → u*, which is excluded by I(2). Thus the path *u ; 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 ; w ; x → v* or a path *w ; u ; x → v*, which contradicts our assumption. *2*

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

**Proof.** Ifthere 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 vari- able substitution begins with this proper root node, and since, by Propo- sition 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 ofthe labels of negative terminal formulae. *2*

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

**Proof.** Given PN(3), each subterm *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 ; v*. *2*

**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 substi- tution, which begins at that label, would not terminate, since the redex corresponding to *n* would expand to a term containing *v*. *2*

**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 termi- nal 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 ofthe proper root node, and all ofthe 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 proofthat these remaining nodes are path-accessible from the proper root node is by induction on the number oflexical 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 termi- nal 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 formu- lae are not daughters ofa 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 ; 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 ofthe nodes on this path have negative occurrences.

Suppose the last unfolding step is a positive unfolding:

*(A\*

*+*

*B) : v −→*

*+ B : v0*

*A : u*

*+ + 0*

*(A=B) : v −→ B : uA : v*

By the inductive hypothesis, there is a path to the lower lambda-node, *v* and thus a path to the plus-daughter, *v0* . By I(3), there is also a path from *v0* to *u*. By Proposition 4.2, the in-degree of *u* is 2, but if the last step of the path *v0 ; u* were via *v → u*, then there would be a cycle *v0 ; v → v0* . 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 *v0 ; u* arises from an axiom link.

Otherwise, the last unfolding step is a negative unfolding:

*+*

*(A\B) : t −→ A : uB : tu*

*+*

*(A=B) : t −→ A : tu B : 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. *2*

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

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

**Proof.** Given I(CT), it is claimed that no lambda-node expands under vari- able substitution to a closed term. Given a lambda-node *v*, with minus- daughter *w*, there is a path *v ; x*, where *x* is some terminal node for which there is no lambda-node *v0 ∈ V* such that *v ; v0 → 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 ex- pand 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 *v0* be its lambda-node, and *u0* be the plus-daughter of *v0* . *x /= w*, and *v0 /= v*, or else trivially *v ; v0 → x*. By I(3), there is a path *v0 → u0 ; y → x*, where *y /= v0* , or else there is a cycle, which contradicts I(2).

By Proposition 4.2, *x* has an in-degree of2, so either *v ; v0 → x*, which

contradicts our choice of *x*, or *v ; y → x*. In the latter case, by Proposi- tion 4.8, *v0 ; 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. *2*

PN(L) will be discussed in Section 5.

* 1. *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 ofa 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 ofeither 1 or 2, depending on how many oftheir 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 ofacyclicity and the fact that LC-graphs have finitely many nodes. *2*

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

**Proof.** A trivial consequence ofProposition 4.8. *2*

We call a terminal formula, *T* , **peripheral** in a sublinkage iff the right- most or leftmost axiomatic formula in the sublinkage’s axiomatic sequence derives from T’s lexical unfolding. We call a node **peripheral** in a sublink- age iff it derives from the lexical unfolding of a peripheral terminal for- mula. 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 ter- minal plus-nodes for the purposes of sequent derivability:

**Proposition 4.20** *If a sequence of terminal formulae has exactly one pos- itive 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 nega- tive occurrences of a node, when both exist, necessarily derive from the same lexical unfolding, i.e., the lexical unfolding of the same terminal for- mula. 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 occur- rences anywhere. By assumption, however, the proper root node labels the rightmost terminal formula, so when they occur in a sublinkage, they are necessarily part ofthe rightmost unfolding. *2*

# Building Spanning Linkages

The definition of LC-graphs themselves does not shed any light on pars- ing 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 proofnet correct- ness criterion, PN(L), which requires that axiomatic linkages can be drawn as a planar graph above the sequence of axiomatic formulae. This is ex- actly 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:

*1) L → B L*

*2) B → X L X+;* for every basic category X

*3) B → X+ L X ;* for every basic category X

*4) B → X X+;* for every basic category X

*5) B → X+ X ;* for every basic category X

*6) L → B*

*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 ofthe edges on its right-hand side.

* 1. *Base Cases: Rules (4) and (5)*

For these rules, the resulting linkage consists ofa 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 ofarcs, 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.

* 1. *Bracketing: Rules (2) and (3)*

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

the LC-graph ofthe right-hand-side category, *L*.

* 1. *Adjunction: Rule (1)*

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

the LC-graph ofthe right-hand-side category, *B*,

the LC-graph ofthe right-hand-side category, *L*, and

for each daughter node in the LC-graph, an arc from its parent lambda- node to itself.

* 1. *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*.

* 1. *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 regard- less of how they were derived. Here, either through two different adjunc- tions 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 qua- dratic in the length of the sequence of axiomatic formulae (which in turn grows with the length ofan 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 expo- nentially as a function of the interval covered by an edge. So this algorithm is not polynomial-time in the worst case.

* 1. *Incremental Enforcement of Integrity Criteria*

In spite ofits exponential worst-case complexity, this algorithm does per- mit some degree ofincrementality in the enforcement ofthe integrity cri- teria for LC-graphs.

Among the three integrity criteria, I(2) stands out because it demands that a particular kind ofpath, namely a cycle, does not exist, whereas I(3) and I(CT) demand that a particular kind ofpath 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 itselfcan 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 sat- isfies 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. *2*

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

# Related Parsing Problems

It is important to realise that the sequent derivability decision problem is only one aspect ofparsing with LCGs. There are other sources ofcomplex- ity, and other restrictions that can be made, which can affect the complex- ity ofthe overall problem.

* 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 result- ing 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 ofparsing problems.

On the other hand, if we fix a particular LCG, *G*, apply Pentus’s con- struction 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 be- cause CFG recognition can also be performed in polynomial-time. This is described in detail by Finkel and Tellier [2].

* 1. *Lexical Ambiguity*

Given a string ofwords, *w*, and a grammar, *G* (unfixed), ifwe 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 deriv- ability question. It is often the case, however, that there is more than one category associated with the words of *w*. Even if the sequent derivabil- ity problem should turn out to be solvable in polynomial time, iteratively choosing categories and performing a sequent derivability check could lead to *2n* possible category selections given a string of *n* words. At present, no method is known for somehow combining multiple lexical categories in the course ofa single proofsearch, although it is likely that one exists.

* 1. *Parsing vs. Recognition*

Sequent derivability, just as context-free chart parsing, asks a yes-or-no question. In the context ofCFG parsing, this is known as the string recog- nition 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 enumer- ating each tree contained in it can take exponentially long.

The analogue in the case of sequent derivability is to provide an ac- tual 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 in- herently 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 ofthe candidate sequent.

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

*{(A=(A\A)):bi }i A:a {A\A:hi A\A:li}i ▶ A:m*

For example, *S2* is the sequent:

*(A=(A\A)):b1 (A=(A\A)):b2 A:a A\A:h1 A\A:l1 A\A:h2 A\A:l2 ▶ A:m*

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

*b1x1 ::: bixia( fi:liyifi) ::: ( f1:l1y1f1)*

*where for all 1 ≤ j ≤ i, either (xi = and yi = hi), or (xi = hi and yi = ).*

**Proof.** Let *Gi* be the graph:

*1*

*b*

*i*

*+*

*c*

*l*

*i i*

*d*

*i*

*e*

*+*

*i*

*k*

*+*

*i*

*g+*

*i*

*f*

*i*

*h*

*i*

and *Gi* be the graph:

*2*

*b*

*h*

*i i*

*+ +*

*c*

*g*

*i i*

*+*

*di*

*e*

*l*

*i i*

*+*

*f*

*k*

*i i*

and let tail*1(i) = ci* and tail*2(i) = gi*. It can be seen that *Gi* and *Gi* internally

*1 2*

possess the paths required by I(3) and I(CT) for the lambda-nodes that they contain, and that both are acyclic. Let *j1 ::: ji* be a sequence of1s and 2s, and let *G* be the smallest graph containing the following subgraphs and arcs:

*G1 ; G2 ;::: ; Gi ;*

*j1 j2 ji*

*m → b1;m → c1;m → d1;* tail*ji (i) → a;*

and for all *1 ≤ k ≤ i − 1;*

tail*jk (k) → bk+1;* tail*jk (k) → ck+1;* tail*jk (k) → dk+1:*

Regardless of the choice of *j1 ::: ji*, *G* is acyclic because all of the *Gk*

*jk*

are

acyclic for *1 ≤ k ≤ i*, and no arc traverses from a *k + 1*-indexed node to a lesser-indexed node for any *1 ≤ k ≤ i − 1*. Also, any node from which all three of *bk*, *ck* and *dk* are path-accessible has access to every node of *Gk* , including tail*jk (k)*, so every node of *G* is path-accessible from *m*. Planar linkages exist for *G* for any choice of *j1 ::: ji*, roughly similar to those given for the running example in Section 2 of this paper, which is actually *S1*. When *jk = 1*, *yk = hk* and *xk =* . When *jk = 2*, *xk = hk* and *yk =* . *2*

*jk*

The choices of *jk* are mutually independent, so there are *2i* possible readings for *Si*. Notice that this is stated purely in terms of sequents, so the exponential blow-up is independent oflexical 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 iftwo 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 ofthe 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 ofcomplement 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 .

1. 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.

1. 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.
2. Morrill, G., *Memoisation of categorial proof nets: parallelism in categorial processing*, Technical Report LSI-96-24-R, Dept. de Llenguatges i Sistemes Informa`tics, Universitat Polite`cnica de Catalunya (1996).
3. 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).
4. Pentus, M., *Lambek grammars are context free*, Technical report, Dept. of Mathematical Logic, Moscow University (1992).
5. Pfenning, F., *Logic programming in the lf logical framework*, in: G. Hyet and

G. Plotkin, editors, *Logical Frameworks*, Cambridge University Press, 1991 .

1. Puite, Q. and R. Moot, *Proof nets for the multimodal lambek calculus* (1999), unpublished ms., Rijksuniversiteit Utrecht.
2. Roorda, D., “Resource Logics: Proof-theoretical Investigations,” Ph.D. thesis, Universiteit van Amsterdam (1991).