Efficient Parsing with the Product-Free Lambek Calculus

Timothy A. D. Fowler
Department of Computer Science
University of Toronto
10 King's College Road, Toronto, ON, M5S 3G4, Canada
tfowler@cs.toronto.edu

Abstract

This paper provides a parsing algorithm for the Lambek calculus which is polynomial time for a more general fragment of the Lambek calculus than any previously known algorithm. The algorithm runs in worst-case time $O(n^5)$ when restricted to a certain fragment of the Lambek calculus which is motivated by empirical analysis. In addition, a set of parameterized inputs are given, showing why the algorithm has exponential worst-case running time for the Lambek calculus in general.

1 Introduction

A wide variety of grammar formalisms have been explored in the past for parsing natural language sentences. The most prominent of these formalisms has been context free grammars (CFGs) but a collection of formalisms known as categorial grammar (CG) (Ajdukiewicz, 1935; Dowty et al., 1981; Steedman, 2000) has received interest because of some significant advantages over CFGs.

First, CG is inherently lexicalized due to the fact that all of the variation between grammars is captured by the lexicon. This is a result of the rich categories which CG uses in its lexicon to specify the functor-argument relationships between lexical items. A distinct advantage of this lexicalization is that the processing of sentences depends upon only those categories contained in the string and not some global set of rules. Second, CG has the advantage that it centrally adopts the principle of compositionality, as outlined in Montague grammar (Montague, 1974), allowing the semantic derivation to exactly parallel the syntactic derivation. This leads to a semantical form which is easily extractable from the syntactic parse.

A large number of CG formalisms have been introduced including, among others, the Lambek calculus (Lambek, 1958) and Combinatory Categorial Grammar (CCG) (Steedman, 2000). Of these, CCG has received the most zealous computational attention. Impressive results have been achieved culminating in the state-of-the-art parser of Clark and Curran (2004) which has been used as the parser for the Pascal Rich Textual Entailment Challenge entry of Bos and Markert (2005). The appeal of CCG can be attributed to the existence of efficient parsing algorithms for it and the fact that it recognizes a mildly context-sensitive language class (Joshi et al., 1989), a language class more powerful than the context free languages (CFLs) that has been argued to be necessary for natural language syntax. The Lambek calculus provides an ideal contrast between CCG and CFGs by being a CG formalism like CCG but by recognizing the CFLs like CFGs (Pentus, 1997).

The primary goal of this paper is to provide an algorithm for parsing with the Lambek calculus and to sketch its correctness. Furthermore, a time bound of $O(n^5)$ will be shown for this algorithm when restricted to product-free categories of bounded order (see section 2 for a definition). The restriction to bounded order is not a significant restriction, due to the fact that categories in CCGbank[1] (Hockenmaier, 2003), a CCG corpus, have a maximum order of 5 and an average order of 0.78 by token. In addition to the presentation of the algorithm, we will provide a parameterized set of in-

[1] Although CCGbank was built for CCG, we believe that transforming it into a Lambek calculus bank is feasible.

puts (of unbounded order) on which the algorithm has exponential running time.

The variant of the Lambek calculus considered here is the product-free Lambek calculus chosen for three reasons. First, it is the foundation of all other non-associative variants of the Lambek calculus including the original Lambek calculus (Lambek, 1958) and the multi-modal Lambek calculus (Moortgat, 1996). Second, the calculus with product is NP-complete (Pentus, 2006), while the sequent derivability in the product-free fragment is still unknown. Finally, the only connectives included are $/$ and $\backslash$, which are the same connectives as in CCG, providing a corpus for future work such as building a probabilistic Lambek calculus parser.

2 Problem specification

Parsing with the Lambek calculus is treated as a logical derivation problem. First, the words of a sentence are assigned *categories* which are built from *basic categories* (e.g. NP and S) and the connectives $\backslash$ and $/$. For example, the category for transitive verbs is $(NP\backslash S)/NP$ and the category for adverbs is $(S/NP)\backslash(S/NP)$[2]. Intuitively, the $\backslash$ and $/$ operators specify the arguments of a word and the direction in which those arguments need to be found. Next, the *sequent* is built by combining the sequence of the categories for the words with the $\vdash$ symbol and the sentence category (e.g. S).

Strictly speaking, this paper only considers the parsing of categories without considering multiple lexical entries per word. However, using techniques such as supertagging, the results presented here yield an efficient method for the broader problem of parsing sentences. Therefore, we can take the size of the input n to be the number of *basic* categories in the sequent.

A parse tree for the sentence corresponds to a proof of its sequent and is restricted to rules following the templates in figure 1. In figure 1, lowercase Greek letters represent categories and uppercase Greek letters represent sequences of categories. A proof for the sentence ”Who loves him?” is given in figure 2.

The version of the Lambek calculus presented above is known as the product-free Lambek calculus allowing empty premises and will be denoted by L. In addition, we will consider the fragment L^k, obtained by restricting L to categories of order bounded by k. The *order* of a category, which can

[2]We use Ajdukiewicz notation, not Steedman notation.

$$\frac{}{\alpha \vdash \alpha}$$

$$\frac{\Gamma \vdash \alpha \qquad \Delta\beta\Theta \vdash \gamma}{\Delta\Gamma\alpha\backslash\beta\Theta \vdash \gamma} \qquad \frac{\alpha\Gamma \vdash \beta}{\Gamma \vdash \alpha\backslash\beta}$$

$$\frac{\Gamma \vdash \alpha \qquad \Delta\beta\Theta \vdash \gamma}{\Delta\beta/\alpha\Gamma\Theta \vdash \gamma} \qquad \frac{\Gamma\alpha \vdash \beta}{\Gamma \vdash \beta/\alpha}$$

Figure 1: The sequent presentation of L.

$$\dfrac{NP \vdash NP \qquad \dfrac{S \vdash S \qquad \dfrac{\dfrac{NP \vdash NP \qquad S \vdash S}{NP\ \ NP\backslash S \vdash S}}{NP\backslash S \vdash NP\backslash S}}{S/(NP\backslash S)\ \ NP\backslash S \vdash S}}{\underset{\text{Who}}{S/(NP\backslash S)}\ \ \underset{\text{loves}}{(NP\backslash S)/NP}\ \ \underset{\text{him}}{NP} \vdash S}$$

Figure 2: A derivation for “Who loves him?”.

be viewed as the depth of the nesting of argument implications, is defined as:

$$o(\alpha) = 0 \text{ for } \alpha \text{ a basic category}$$
$$o(\alpha/\beta) = o(\beta\backslash\alpha) = \max(o(\alpha), o(\beta) + 1)$$

For example, $o((NP\backslash S)/NP) = 1$ and $o((S/NP)\backslash(S/NP)) = 2$.

3 Related work

Two other papers have provided algorithms similar to the one presented here.

Carpenter and Morrill (2005) provided a graph representation and a dynamic programming algorithm for parsing in the Lambek calculus with product. However, due to there use of the Lambek calculus with product and to their choice of correctness conditions, they did not obtain a polynomial time algorithm for any significant fragment of the calculus.

Aarts (1994) provided an algorithm for L^2 which is not correct for L. Ours is polynomial time for L^k, for any constant k, and is correct for L, albeit in exponential running time.

A number of authors have provided polynomial time algorithms for parsing with CCG which gives some insight into how good our bound of $O(n^5)$ is. In particular, Vijay-Shanker and Weir (1994) provided a chart parsing algorithm for CCG with a time bound of $O(n^6)$.

4 An algorithm for parsing with L

This section presents a chart parsing algorithm similar to CYK where entries in the chart are arcs annotated with graphs. The graphs will be referred to as abstract term graphs (ATGs) since they are graph representations of abstractions over semantic terms. ATGs will be presented in this section by construction. See section 5 for their connection to the proof structures of Roorda (1991).

The algorithm consists of two steps. First, the base case is computed by building the *base ATG* B and determining the set of *surface variables* by using the proof frames of Roorda (1991). Second, the chart is filled in iteratively according to the algorithms specified in the appendix. The details for these two steps can be found in sections 4.1 and 4.2, respectively. Section 4.3 introduces a procedure for culling extraneous ATGs which is necessary for the polynomial time proof and section 4.4 discusses recovery of proofs from the packed chart. An example of the algorithm is given in figure 3.

For parsing with L, the input is a sequent and for parsing with L^k, the input is a sequent with categories whose order is bounded by k. Upon completion, the algorithm outputs "YES" if there is an arc from 0 to $n-1$ and "NO" otherwise.

4.1 Computing the base case

Computing the base case consists of building the proof frame and then translating it into a graph, the base ATG B.

4.1.1 Building the proof frame

Proof frames are the part of the theory of proof nets which we need to build the base ATG. The proof frame for a sequent is a structure built on top of the categories of the sentence. To build the proof frame, all categories in the sequent are assigned a polarity and labelled by a fresh variable. Categories to the left of $\vdash$ are assigned negative polarity and the category to the right of $\vdash$ is assigned positive polarity. Then, the four decomposition rules shown in table 1 are used to build a tree-like structure (see figure 3). The decomposition rules are read from bottom to top and show how to decompose a category based on its main connective and polarity. In table 1, d is the label of the category being decomposed, f, g and h are fresh variables and order of premises is important.

The bottom of the proof frame consists of the original sequent's categories with labels and polarities. These are called *terminal formulae*. The top of the proof frame consists of basic categories with labels and polarities. These are called the *axiomatic formulae*. In addition, we will distinguish

$$\frac{\alpha \overset{+}{:} f \qquad \beta \overset{-}{:} df}{\alpha\backslash\beta \overset{-}{:} d} \qquad \frac{\beta \overset{+}{:} h \qquad \alpha \overset{-}{:} g}{\alpha\backslash\beta \overset{+}{:} d}$$

$$\frac{\alpha \overset{-}{:} df \qquad \beta \overset{+}{:} f}{\alpha/\beta \overset{-}{:} d} \qquad \frac{\beta \overset{-}{:} g \qquad \alpha \overset{+}{:} h}{\alpha/\beta \overset{+}{:} d}$$

Table 1: The proof frame decomposition rules.

the leftmost variable in the label of each axiomatic formula as its *surface variable*. See figure 3 for an example.

4.1.2 Building the Base ATG

The base ATG B is built from the proof frame in the following way. The vertices of the base ATG are the surface variables plus a new special vertex τ. The edges of ATGs come in two forms: Labelled and unlabeled, specified as $\langle s, d, l\rangle$ and $\langle s, d\rangle$, respectively, where s is the source, d is the destination and l, where present, is the label.

To define the edge set of B, we need the following:

Definition. For a variable u that labels a positive category in a proof frame, the *axiomatic reflection*, $\rho(u)$, is the unique surface variable v such that on the upward path from u and v in the proof frame, there is no formula of negative polarity. For example, in figure 3, $\rho(b) = c$.

The edgeset E of the base ATG is as follows:

1. $\langle m, \rho(p_i)\rangle \in E$ for $1 \leq i \leq k$ where $mp_1 \ldots p_k$ appears as the label of some negative axiomatic formula

2. $\langle \tau, \rho(t)\rangle \in E$ where t is the label of the positive terminal formula

3. For each rule with a positive conclusion, negative premise labelled by g and positive premise labelled by h, $\langle \rho(h), g, g\rangle \in E$

A labeled edge in an ATG specifies that its source must eventually connect to its destination to complete a path corresponding to its label. For example, G_1 contains the edge $\langle c, e, d\rangle$ which indicates that to complete the path from c to d, we must connect c to e. In contrast, an unlabeled edge in an ATG specifies that its source is already connected to its destination. For example, in figure 3, G_3 contains the edge $\langle a, f\rangle$ which indicates that there is some path, over previously deleted nodes, which connects a to f.

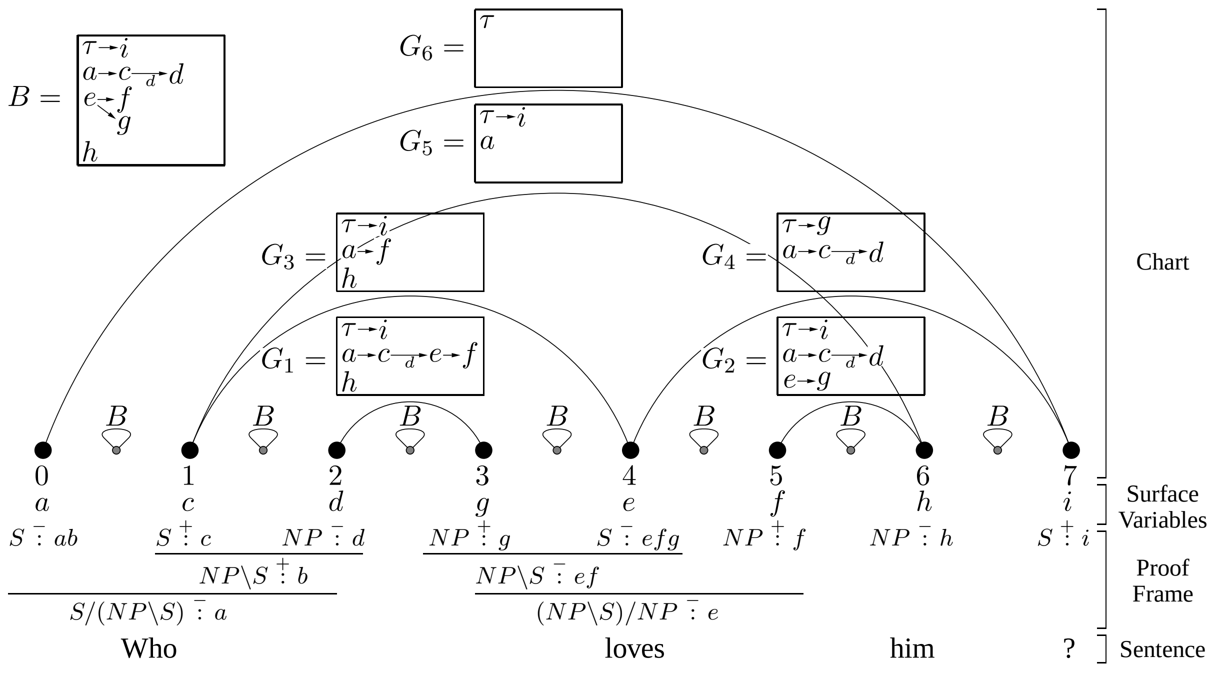

Figure 3: The algorithm's final state on the sequent $S/(NP\backslash S)\ \ (NP\backslash S)/NP\ \ NP \vdash S$.

Note that all nodes in an ATG have unlabeled in-degree of either 0 or 1 and that the vertices of an ATG are the surface variables found outside its arc.

4.2 Filling in the chart

Once the base ATG and the sequence of surface variables is determined, we can begin filling in the chart. The term *entry* refers to the collection of arcs beginning and ending at the same nodes of the chart. An arc's *length* is the difference between its beginning and end points, which is always odd. Note that each entry in the example in figure 3 contains only one arc. We will iterate across the entries of the chart and at each entry, we will attempt a *Bracketing* and a number of *Adjoinings*. If an attempt results in a *violation*, no new ATG is inserted into the chart. Otherwise, a new ATG is computed and inserted at an appropriate entry.

Bracketing is an operation on a single ATG where we attempt to extend its arc by connecting two nodes with the same basic category and opposite polarity. For example, G_3 is the result of bracketing G_1. Adjoining, on the other hand, is an operation on two adjacent ATGs where we attempt to unify their ATGs into one larger ATG. For example, G_5 is the result of adjoining G_3 and G_2.

The chart filling process is described by algorithm 1 in the appendix. The chart in figure 3 is filled by the graphs $G_1, \ldots, G_6$, in that order. A walk through of the example is given in the remainder of this section. Arcs of length 1 are treated specially, since they are derived directly from the base ATG. To show this, the base ATG is shown at pseudo-nodes, labeled by Bs.

4.2.1 Inserting arcs of length 1

This section corresponds to lines 1-2 of algorithm 1 in the appendix. For each arc from i to $i+1$, we will attempt to bracket the base ATG from axiomatic formula i to axiomatic formula $i+1$.

To follow our example, the first step is to consider inserting an arc from 0 to 1 by bracketing B. Bracketing causes a positive surface variable to be connected to a negative surface variable and in this case, a cycle from a to c and back to a is formed resulting in the violation on line 12 of algorithm 2. Therefore, no arc is inserted.

Then, the second step considers inserting an arc from 1 to 2. However, axiomatic formula 1 has category S and axiomatic formula 2 has category NP which results in the violation on line 3 of algorithm 2 since they are not the same.

Next, we attempt to insert an arc from 2 to 3. In this case, no violations occur meaning that we can insert the arc. The intuition is that the ATG for this arc is obtained by connecting g to d in the base ATG. Since c must eventually connect to d ($c \rightarrow_d d$), and now g connects to d, the in-degree constraint on ATG nodes requires that the path connecting c to d pass through g. Further-

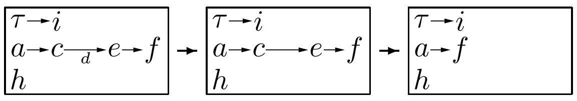

Figure 4: The intuition for bracketing from c to e.

more, the only way to connect c to g is through e. So $c \rightarrow_d e$. Then, we delete d and g.

This procedure continues until we have considered all possible arcs of length 1.

4.2.2 Inserting arcs of length 3 and greater

Next, we iterate across graphs in the chart and for each, consider whether its ATG can be bracketed with the axiomatic formulae on either side of it and whether it can be adjoined with any of the other graphs in the chart. This process closely resembles CYK parsing as described on lines 3-10 of algorithm 1. The choice of shortest to longest is important because part of the invariant of our dynamic program is that all derivable ATGs on shorter arcs have already been added.

Following our example, the first graph to be considered is G_1. First, we attempt to bracket it from axiomatic formulae 1 to 4. As before, this intuitively involves connecting c to e in the ATG for this arc. This is allowed because no cycles are formed and no labelled edges are prohibited from eventually being connected. Then, as before, we delete the vertices c and e and as a result connect a to f, resulting in G_3. The bracketing process is illustrated in figure 4.

Next, we consider all graphs to which G_1 could adjoin and there are none, since such graphs would need to annotate arcs which either end at 1 or begin at 4. After processing G_1, we process G_2, which has a successful bracketing resulting in G_4 and no successful adjoinings.

Next, we process G_3. Bracketing it is prohibited, as it would result in a cycle from a to f and back to a. However, it is possible to adjoin G_3 with G_2, since they are adjacent.

The adjoining of two graphs can be viewed as a kind of intersection of the two ATGs, in the sense that we are combining the information in both graphs to yield a single more concise graph. Attempting an adjoining involves traversing the two graphs being adjoined and the base ATG in both a forward and a backward direction as specified in algorithms 4 and 5 in the appendix.

The intuition behind these traversals is to generate a picture of what the combination of the two graphs must look like as illustrated in figure 5. In general, we can only reconstruct those parts of the graph which are necessary for determining the resultant ATG and no more. The dotted edges indicate uncertainty about the edges present at this stage of the algorithm. Adjoining G_2 and G_3 does not fail and the resultant graph is G_5.

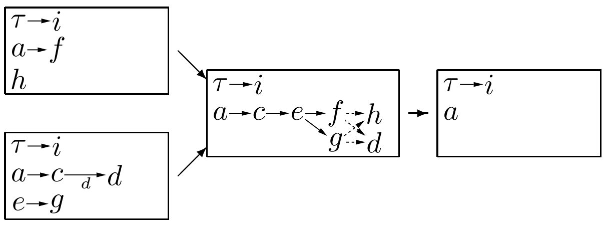

Figure 5: The intuition for adjoining two ATGs.

Note that this example does not contain any instances of two identical ATGs being inserted multiple times into the chart which occurs often in large examples yielding significant savings of computation.

4.3 Culling of extraneous ATGs

It often happens that an entry in the chart contains two ATGs such that if one of them is extendable to a complete proof then the other necessarily is as well. In this case, the former can be discarded. We will outline such a method here that is important for the polynomial time proof.

Definition. ATGs G_1 and G_2 are *equivalent* if some surjection of edge labels to edge labels applied to the those of G_1 yields those of G_2.

Then, if two ATGs in a chart are equivalent, one can be discarded.

4.4 Recovering proofs from a packed chart

The algorithm as described above is a method for answering the decision problem for sequent derivability in the Lambek calculus. However, we can annotate the ATGs with the ATGs they are derived from so that a complete set of Roorda-style proof nets, and thus the proofs themselves, can be recovered.

5 Correctness

Correctness of the algorithm is obtained by using structural induction to prove the equivalence of the constructive definition of ATGs outlined in section 4 and a definition based on semantic terms given in this section:

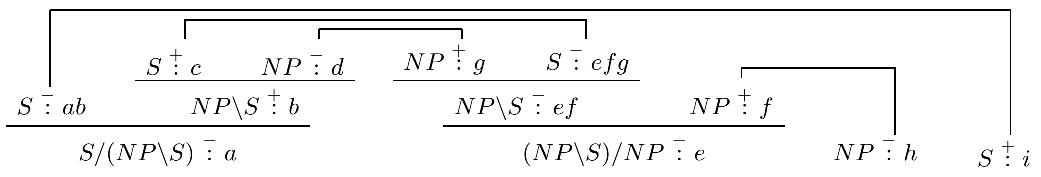

Figure 6: A proof structure for “Who loves him?”.

Definition. A *partial proof structure* is a proof frame together with a matching of the axiomatic formulae. A *proof structure* is a partial proof structure whose matching is complete.

An example is given in figure 6. Proof structures correspond to proofs under certain conditions and our conditions will be based on the semantic term of the proof given to us by the Curry-Howard isomorphism for the Lambek calculus (Roorda, 1991). To do this, we interpret left rules as functional application and right rules as functional abstraction of lambda terms. Under this interpretation, the semantic term obtained from the proof structure in figure 6 is $a\lambda d.ehd$.

As in Roorda (1991), proof structures correspond to a proof if the semantic term assigned to the sentence category is a well formed lambda term which includes all the terms assigned to the words of the sentence. Then, ATGs are graph representations of abstractions of the undetermined portion of semantic terms of partial proof structures. Unlabeled edges correspond to functional applications whose arguments must still be determined and labelled edges correspond to functional abstractions whose body does not yet contain an instance of the abstracted variable. The violations which occur during the execution of the algorithm correspond to the various ways in which a lambda term can be ill formed.

6 Asymptotic Running Time Complexity

In this section we provide proof sketches for the runtime of the algorithm. Let $f(n)$ be a bound on the number of arcs occurring in an entry in the chart where n is the number of axiomatic formulae. Then, observe that the number of edges within an ATG is $O(n^2)$ and the number of edges adjacent to a vertex is $O(n)$, due to basic properties of ATGs.

Then, it is not hard to prove that the worst case running time of Bracketing is $O(n^2)$, which is dominated by the for loops of lines 20-23 of algorithm 2.

Next, with some effort, we can see that the worst case running time of Adjoining is dominated by the execution of the procedures Fore and Back. But, since there are at most a linear number of labels l and for each label l we need to visit each vertex in G_1 and G_2 at most a constant number of times, the worst case running time is $O(n^2)$.

Then, for each ATG, we attempt at most one bracketing and adjoinings with at most $2n+1$ other entries for which there can be $(2n+1)f(n)$ ATGs. Therefore, each entry can be processed in worst case time $O(n^3 f(n)^2)$.

Finally, there are $O(n^2)$ entries in the chart, which means that the entire algorithm takes time $O(n^5 f(n)^2)$ in the worst case. Sections 6.1 and 6.2 discuss the function $f(n)$.

6.1 Runtime for L^k

By structural induction on the proof frame decomposition rules and the base ATG building algorithm, it can be proven that in L^k the length of the longest path in the base ATG is bounded by k.

Next, consider a partition of the surface variables into a pair of sets such that the axiomatic formulae corresponding to the surface variables within each set are contiguous. For the example in figure 3, one such pair of sets is $S_1 = \{a, c, d, g\}$ and $S_2 = \{e, f, h, i\}$. Then, given such a partition, it can be proven that there is at least one maximal path P in the base ATG such that all vertices in one set that are adjacent to a vertex in the other set are either in P or adjacent to some vertex in P. For example, a maximal path for S_1 and S_2 is $P = e \to g$.

An entry in the chart induces two such partitions, one at the left edge of the entry and one at the right edge. Therefore, we obtain two such maximal paths and for any ATG G in this entry and any vertex v not in or adjacent to one of these paths, either v is not in G or v has the same neighbourhood in G as it has in the base ATG. Then, the number of vertices adjacent to vertices in these paths can be as many as n. However, if we put these vertices into sets such that vertices in a set have identical neighbourhoods, the number of sets is dependant only on k.

In the worst case, the out-neighbourhood of one of these sets can be any set of these sets. So, we get a bound for $f(n)$ to be $O(k^2 4^k)$. Therefore, because k is constant in L^k, $f(n)$ is constant and the running time of the algorithm for L^k is $O(n^5)$ in the worst case.

6.2 Runtime for L

Despite the results of section 6.1, this algorithm has an exponential running time for L. We demonstrate this with the following set of parameterized sequents:

$$\begin{aligned} F(1) &= ((A/A)\backslash A)\backslash A \\ F(i) &= ((A/(A/F_{i-1}))\backslash A)\backslash A \text{ for } i > 1 \\ U(n) &= F_n \ F_n \vdash A\backslash A \end{aligned}$$

Theorem. *There are* $\frac{(2n-1)!}{n!(n-1)!} \in \Theta(4^n)$ *distinct arcs in the entry from* n *to* $3n-1$ *in the chart for* $U(n)$.

Proof. By induction and a mapping from the possible matchings to the possible permutations of a sequence of length $2n-1$ such that two subsequences of length n and $n-1$ are in order. $\square$

7 Conclusions and Future Work

We have presented a novel algorithm for parsing in the Lambek calculus, sketched its correctness and shown that it is polynomial time in the bounded-order case. Furthermore, we presented a set of parameterized sequents proving that the algorithm is exponential time in the general case, which aids future research in finding either a polynomial time algorithm or an NP-completeness proof for L.

In addition, this algorithm provides another step toward evaluating the Lambek calculus against both CFGs (to evaluate the importance of Categorial Grammar) and CCG (to evaluate the importance of the mildly context-sensitive languages).

In the future, we plan on determining the running time of this algorithm on an actual corpus, such as a modified version of CCGbank, and then to empirically evaluate the Lambek calculus for natural language processing. In addition, we would like to investigate extending this algorithm to more complex variants of the Lambek calculus such as the multi-modal calculus using the proof nets of Moot and Puite (2002).

Acknowledgments

Many thanks to Gerald Penn, for his insightful comments and for guiding this research.

References

Aarts, Erik. 1994. Proving Theorems of the Second Order Lambek Calculus in Polynomial Time. *Studia Logica*, 53:373–387.

Ajdukiewicz, Kazimierz. 1935. Die syntaktische Konnexitat. *Studia Philosophica*, 1(1-27).

Bos, Johan and Katja Markert. 2005. Recognising textual entailment with logical inference. *Proceedings of HLT and EMNLP*, pages 628–635.

Carpenter, Bob and Glyn Morrill. 2005. Switch Graphs for Parsing Type Logical Grammars. *Proceedings of IWPT '05, Vancouver*.

Clark, Steven and James R. Curran. 2004. Parsing the WSJ using CCG and log-linear models. *Proceedings of ACL '04*, pages 104–111.

Dowty, David R., Robert E. Wall, and Stanley Peters. 1981. *Introduction to Montague Semantics*. Reidel.

Hockenmaier, Julia. 2003. *Data and Models for Statistical Parsing with Combinatory Categorial Grammar*. Ph.D. thesis, University of Edinburgh.

Joshi, Aravind K., K. Vijay-Shanker, and David J. Weir. 1989. *The Convergence of Mildly Context-sensitive Grammar Formalisms*. University of Pennsylvania.

Lambek, Joachim. 1958. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170.

Montague, Richard. 1974. *Formal philosophy: selected papers of Richard Montague*. Yale University Press New Haven.

Moortgat, Michael. 1996. Multimodal linguistic inference. *Journal of Logic, Language and Information*, 5(3):349–385.

Moot, Richard and Quintijn Puite. 2002. Proof Nets for the Multimodal Lambek Calculus. *Studia Logica*, 71(3):415–442.

Pentus, Mati. 1997. Product-Free Lambek Calculus and Context-Free Grammars. *The Journal of Symbolic Logic*, 62(2):648–660.

Pentus, Mati. 2006. Lambek calculus is NP-complete. *Theoretical Computer Science*, 357(1-3):186–201.

Roorda, Dirk. 1991. *Resource Logics: Proof-theoretical Investigations*. Ph.D. thesis, Universiteit van Amsterdam.

Steedman, Mark. 2000. *The Syntactic Process*. Bradford Books.

Vijay-Shanker, K. and David J. Weir. 1994. Parsing Some Constrained Grammar Formalisms. *Computational Linguistics*, 19(4):591–636.

Appendix. Algorithm Pseudocode

The term *source set* refers to the out-neighbourhood of τ. The term *minus variable* refers to surface variables obtained from negative axiomatic formulae plus τ. X_i refers to the i^{th} axiomatic formula.

Algorithm 1 Chart Iteration

```
1: for $i = 0$ to $n-1$ do
2:    Bracketing($B$, $X_i$, $X_{i+1}$)
3: for $l = 1, 3, 5, \ldots$ to $n-1$ do
4:    for $e = 0$ to $n-l-1$ do
5:       for each arc from $e$ to $e+l$ with ATG $G$ do
6:          Bracketing($G$, $X_{e-1}$ to $X_{e+l+1}$)
7:          Adjoin $G$ to ATGs from $e-l-1$ to $e-1$
8:          for $al = 1, 3, ..., l-2$ do
9:             Adjoin $G$ to ATGs from $e-al-1$ to $e-1$
10:            Adjoin $G$ to ATGs from $e+l+1$ to $e+l+al+1$
```

Algorithm 2 Bracketing(G, X_i, X_j)

```
1: $C_i \overset{p_i}{:} l_i = X_i$ and $C_j \overset{p_j}{:} l_j = X_j$
2: if $C_i \neq C_j$ then
3:    $Violation:$ Mismatched Basic Categories
4: if $p_i = p_j$ then
5:    $Violation:$ Mismatched Polarities
6: Let $m, p \in \{i, j\}$ such that $p_m$ is negative and $p_p$ is positive
7: if $G$ is not from 1 to $n-1$ and the source set of $G$ is the singleton $l_p$ and $l_m$ has out-degree 0 in $G$ then
8:    $Violation:$ Empty Source Set
9: if the edge $\langle l_m, l_p\rangle \in G$ then
10:   $Violation:$ Cycle Exists
11: if $l_p$ is in the source set of $G$ and there exists an in-edge of $m$ with label $l$ such that no edge from $p$ to $m$ has label $l$ and no edge from a vertex other than $p$ to a vertex other than $m$ has label $l$ then
12:   $Violation:$ Path Completion Impossible
13: if $m$ has out-degree 0 and and there exists an out-edge of $p$ with label $l$ such that no edge from $p$ to $m$ has label $l$ and no edge from a vertex other than $p$ to a vertex other than $m$ has label $l$ then
14:   $Violation:$ Path Completion Impossible
15: Copy $G$ to yield $H$
16: for each edge $\langle l_p, l_m, l\rangle \in G$ do
17:   Delete all edges from $H$ with label $l$
18: Delete $l_m$, $l_p$ and all their incident edges from $H$
19: Let $in_p$ be the in-neighbour of $l_p$ in $G$
20: for each $q$ in the out-neighbourhood of $l_m$ in $G$ do
21:   Insert $\langle in_p, q\rangle$ into $H$
22:   for each edge $\langle p, d, l\rangle$ in $G$ do
23:      Insert $\langle q, d, l\rangle$ into $H$
24: for each edge $\langle q, m, l\rangle$ in $G$ do
25:   Insert $\langle q, in_p, l\rangle$ into $H$
26: if $H$ contains a cycle then
27:   $Violation:$ Future Cycle Required
28: return $H$
```

Algorithm 3 Adjoining(G_1, G_2)

```
1: Let $V_H$ be the intersection of the vertices in $G_1$ and $G_2$
2: if $V_H \neq \tau$ and $Fore(\tau, G_1, G_2) \cap V_H = \emptyset$ then
3:    $Violation:$ Empty Source Set
4: for each $l$ such that $l$ labels an edge in $G_1$ and $G_2$ do
5:    Let $\langle p, m, l\rangle$ be the unique edge labelled $l$ in $B$
6:    if $Fore(p, G_1, G_2, l) \cap Back(m, G_1, G_2) = \emptyset$ then
7:       if $Fore(p) \cap V_H = \emptyset$ then
8:          $Violation:$ Path Completion Impossible
9:       if $Back(m) \cap V_H = \tau$ then
10:         $Violation:$ Path Completion Impossible
11: Let $H$ be the graph with vertex set $V_H$ and no edges
12: for each minus variable $m \in V_H$ do
13:   for each $p \in Fore(m, G_1, G_2, \emptyset)$ do
14:      Insert $\langle m, p\rangle$ into $H$
15: for each $l$ such that $l$ labels an edge in $G_1$ and $G_2$ do
16:   Let $\langle p, m, l\rangle$ be the unique edge labelled $l$ in $B$
17:   if $Fore(p, G_1, G_2, l) \cap Back(m, G_1, G_2) = \emptyset$ then
18:      for each $q \in Fore(p, G_1, G_2, l) \cap V_H$ do
19:         Insert $\langle q, Back(m, G_1, G_2) \cap V_H, l\rangle$ into $H$
20: return H
```

Algorithm 4 Fore(v, G_1, G_2, l)

```
1: if $v \in G_1$ and $v \in G_2$ then
2:    return $\{v\}$
3: else
4:    if $v$ is a minus vertex then
5:       S = $\cup_{i \in \{1,2\}}$Out-neighbourhood$_{G_i} v$
6:    else if $v$ is a plus vertex then
7:       Let $j$ be such that $v \in G_j$
8:       S = $\cup_{e \in \text{Edges labelled by} l}$Source of e
   F = S
9:    while $S$ is not empty do
10:      Remove any element $u$ from $S$
11:      Let $m$ be the in-neighbour of $u$ in $B$
12:      if $u$ does not appear in one of $G_1, G_2$ and $m$ does not appear in the other then
13:         Let $i$ be such that $m \in G_i$
14:         Let $O$ be the out-neighbourhood of $m$ in $G_i$
15:         $S = S \cup O$
16:         $F = F \cup O$
17:      $F = F \cup \{m\}$
18:   return F
```

Algorithm 5 Back(m, G_1, G_2)

```
1: if $m \in G_1$ and $m \in G_2$ then
2:    return $\{m\}$
3: else
4:    Let $i, j \in \{1, 2\}$ be such that $m \in G_i$ and $m \notin G_j$
5:    Let $m'$ be the destination of the edges labelled by $m$ in $G_j$
6:    $M = \{m, m'\}$
7:    while $m' \notin G_1$ and $m' \notin G_2$ do
8:       Let $i', j' \in \{1, 2\}$ be such that $m' \in G_{i'}$ and $m' \notin G_{j'}$
9:       Let $p \in G_{j'}$ be an out-neighbour of $m'$ in $B$
10:      Let $m''$ be the in-neighbour of $p$ in $G_{j'}$
11:      $m' = m''$
12:      $M = M \cup \{m''\}$
13:   return M
```