Abstract

We study the computational complexity of the parsing problem of a variant of Lambek Categorial Grammar that we call semidirectional. In semidirectional Lambek calculus SDL there is an additional non-directional abstraction rule allowing the formula abstracted over to appear anywhere in the premise sequent's left-hand side, thus permitting non-peripheral extraction. SDL grammars are able to generate each context-free language and more than that. We show that the parsing problem for semidirectional Lambek Grammar is NP-complete by a reduction of the 3-Partition problem.

Key words: computational complexity, Lambek Categorial Grammar

1 Introduction

Categorial Grammar (CG) and in particular Lambek Categorial Grammar (LCG) have their well-known benefits for the formal treatment of natural language syntax and semantics. The most outstanding of these benefits is probably the fact that the specific way, how the complete grammar is encoded, namely in terms of 'combinatory potentials' of its words, gives us at the same time recipes for the construction of meanings, once the words have been combined with others to form larger linguistic entities. Although both frameworks are equivalent in weak generative capacity — both derive exactly the context-free languages —, LCG is superior to CG in that it can cope in a natural way with extraction and unbounded dependency phenomena. For instance, no special category assignments need to be stipulated to handle a relative clause containing a trace, because it is analyzed, via hypothetical reasoning, like a traceless clause with the trace being the hypothesis to be discharged when combined with the relative pronoun.

Figure 1 illustrates this proof-logical behaviour. Notice that this natural-deduction-style proof in the type logic corresponds very closely to the phrase-structure tree one would like to adopt in an analysis with traces. We thus can derive Bill misses ε as an s from the hypothesis that there is a "phantom" np in the place of the trace. Discharging the hypothesis, indicated by index 1, results in Bill misses being analyzed as an s/np from zero hypotheses. Observe, however, that such a bottom-up synthesis of a new unsaturated type is only required, if that type is to be consumed (as the antecedent of an implication) by another type. Otherwise there would be a simpler proof without this abstraction. In our example the relative pronoun has such a complex type triggering an extraction.

A drawback of the pure Lambek Calculus L is that it only allows for so-called 'peripheral extraction', i.e., in our example the trace should better be initial or final in the relative clause.

This inflexibility of Lambek Calculus is one of the reasons why many researchers study richer systems today. For instance, the recent work by Moortgat (Moortgat 94) gives a systematic in-depth study of mixed Lambek systems, which integrate the systems L, NL, NLP, and LP. These ingredient systems are obtained by varying the Lambek calculus along two dimensions: adding the permutation rule (P) and/or dropping the assumption that the type combinator (which forms the sequences the systems talk about) is associative (N for non-associative).

Taken for themselves these variants of L are of little use in linguistic descriptions. But in Moortgat's mixed system all the different resource management modes of the different systems are left intact in the combination and can be exploited in different parts of the grammar. The relative pronoun which would, for instance, receive category (np\np)/(np → s) with → being implication in LP,1 i.e., it requires

1The Lambek calculus with permutation LP is also called the "nondirectional Lambek calculus" (Benthem 88). In it the leftward and rightward implication
as an argument “an s lacking an np somewhere”. The present paper studies the computational complexity of a variant of the Lambek Calculus that lies between L and LP, the Semidirectional Lambek Calculus SDL. Since LP derivability is known to be NP-complete, it is interesting to study restrictions on the use of the LP operator –o. A restriction that leaves its proposed linguistic applications intact is to admit a type B →o A only as the argument type in functional applications, but never as the functor. Stated prove-theoretically for Gentzen-style systems, this amounts to disallowing the left rule for –o. Surprisingly, the resulting system SDL can be stated without the need for structural rules, i.e., as a monolithic system with just one structural connective, because the ability of the abstracted-over formula to permute can be directly encoded in the right rule for –o. Note that our purpose for studying SDL is not that it might be in any sense better suited for a theory of grammar (except perhaps, because of its simplicity), but rather, because it exhibits a core of logical behaviour that any richer system also needs to include, at least if it should allow for non-peripheral extraction. The sources of complexity uncovered here are thus a fortiori present in all these richer systems as well.

2 Semidirectional Lambek Grammar

2.1 Lambek calculus

The semidirectional Lambek calculus (henceforth SDL) is a variant of J. Lambek’s original (Lambek 58) calculus of syntactic types. We start by defining the Lambek calculus and extend it to obtain SDL.

Formulae (also called “syntactic types”) are built from a set of propositional variables (or “primitive types”) B = {b1, b2, …} and the three binary connectives •, \, /, called product, left implication, and right implication. We use generally capital letters A, B, C, … to denote formulae and capitals towards the end of the alphabet T, U, V, … to denote sequences of formulae. The concatenation of sequences U and V is denoted by (U, V).

In Semidirectional Lambek Calculus we add as additional connective the LP implication –o, but equip it only with a right rule.

\[
\frac{U, B, V \Rightarrow A}{T \Rightarrow B \rightarrow A} (-o A) \text{ if } T = (U, V) \text{ nonempty.}
\]

Note that our purpose for studying SDL is not that it might be in any sense better suited for a theory of grammar (except perhaps, because of its simplicity), but rather, because it exhibits a core of logical behaviour that any richer system also needs to include, at least if it should allow for non-peripheral extraction. The sources of complexity uncovered here are thus a fortiori present in all these richer systems as well.

\[\text{Figure 1: Extraction as resource-conscious hypothetical reasoning}\]

\[\text{Figure 2: Lambek's original calculus}\]

\[\text{Formulae (also called “syntactic types”) are built from a set of propositional variables (or “primitive types”) B = \{b1, b2, …\} and the three binary connectives •, \, /, called product, left implication, and right implication. We use generally capital letters A, B, C, … to denote formulae and capitals towards the end of the alphabet T, U, V, … to denote sequences of formulae. The concatenation of sequences U and V is denoted by (U, V).}\]

\[\text{First of all, since we don't need products to obtain our results and since they only complicate matters, we eliminate products from consideration in the sequel.}\]

\[\text{In Semidirectional Lambek Calculus we add as additional connective the LP implication –o, but equip it only with a right rule.}\]

\[\frac{U, B, V \Rightarrow A}{T \Rightarrow B \rightarrow A} (-o A) \text{ if } T = (U, V) \text{ nonempty.}\]

\[\text{Note that our purpose for studying SDL is not that it might be in any sense better suited for a theory of grammar (except perhaps, because of its simplicity), but rather, because it exhibits a core of logical behaviour that any richer system also needs to include, at least if it should allow for non-peripheral extraction. The sources of complexity uncovered here are thus a fortiori present in all these richer systems as well.}\]
Let us define the polarity of a subformula of a sequent $A_1, \ldots, A_n \Rightarrow A$ as follows: $A$ has positive polarity, each of $A_i$ have negative polarity and if $B/C$ or $C/B$ has polarity $p$, then $B$ also has polarity $p$ and $C$ has the opposite polarity of $p$ in the sequent.

A consequence of only allowing the ($\rightarrow R$) rule, which is easily proved by induction, is that in any derivable sequent $\rightarrow$ may only appear in positive polarity. Hence, $\rightarrow$ may not occur in the (cut) formula $A$ of a $(Cut)$ application and any subformula $B \rightarrow A$ which occurs somewhere in the prove must also occur in the final sequent. When we assume the final sequent's RHS to be primitive (or $\rightarrow$-less), then $\rightarrow$ may not occur in the $(Cut)$ formula $A$ of a $(Cut)$ application and any subformula $B \rightarrow A$ which occurs somewhere in the prove will be.

We can show Cut Elimination for this calculus by a straightforward adaptation of the Cut elimination proof for $L$. We omit the proof for reasons of space.

**Proposition 1 (Cut Elimination)** Each SDL-derivable sequent has a cut-free proof.

The cut-free system enjoys, as usual for Lambek-like logics, the Subformula Property: in any proof only subformulae of the goal sequent may appear.

In our considerations below we will make heavy use of the well-known count invariant for Lambek systems (Benthem 88), which is an expression of the resource-consciousness of these logics. Define $\#_b(A)$ (the $b$-count of $A$), a function counting positive and negative occurrences of primitive type $b$ in an arbitrary type $A$, to be

$$
\#_b(A) = \begin{cases} 
1 & \text{if } A = b \\
0 & \text{if } A \text{ primitive and } A \neq b \\
\#_b(B) - \#_b(C) & \text{if } A = B/C \text{ or } A = C \rightarrow B \\
\#_b(B) + \#_b(C) & \text{if } A = B \bullet C 
\end{cases}
$$

The invariant now states that for any primitive $b$, the $b$-count of the RHS and the LHS of any derivable sequent are the same. By noticing that this invariant is true for $(Ax)$ and is preserved by the rules, we immediately can state:

**Proposition 2 (Count Invariant)** If $\vdash_{SDL} U \Rightarrow A$, then $\#_b(U) = \#_b(A)$ for any $b \in B$.

Let us in parallel to SDL consider the fragment of it in which $(/R)$ and $(\setminus R)$ are disallowed. We call this fragment SDL-. Remarkable about this fragment is that any positive occurrence of an implication must be $\rightarrow$ and any negative one must be $/ \text{ or } \setminus$.

### 2.2 Lambek Grammar

**Definition 3** We define a Lambek grammar to be a quadruple $(E, \mathcal{F}, b_s, l)$ consisting of the finite alphabet of terminals $\Sigma$, the set $\mathcal{F}$ of all Lambek formulae generated from some set of propositional variables which includes the distinguished variable $s$, and the lexical map $l : \Sigma \rightarrow 2^\mathcal{F}$ which maps each terminal to a finite subset of $\mathcal{F}$.

We extend the lexical map $l$ to nonempty strings of terminals by setting $l(w_1 w_2 \ldots w_n) := l(w_1) \times l(w_2) \times \ldots \times l(w_n)$ for $w_1 w_2 \ldots w_n \in \Sigma^+$.

The language generated by a Lambek grammar $G = (\Sigma, \mathcal{F}, b_s, l)$ is defined as the set of all strings $w_1 w_2 \ldots w_n \in \Sigma^+$ for which there exists a sequence
of types $U \in l(w_1 w_2 \ldots w_n)$ and $\vdash_L U \Rightarrow b_S$. We denote this language by $L(G)$.

An SDL-grammar is defined exactly like a Lambek grammar, except that $\Rightarrow_{SDL}$ replaces $\Rightarrow_L$.

Given a grammar $G$ and a string $w = w_1 w_2 \ldots w_n$, the parsing (or recognition) problem asks the question, whether $w$ is in $L(G)$.

It is not immediately obvious, how the generative capacity of SDL-grammars relate to Lambek grammars or nondirectional Lambek grammars (based on calculus LP). Whereas Lambek grammars generate exactly the context-free languages (modulo the missing empty word) (Pentus 93), the latter generate all permutation closures of context-free languages (Benthem 88). This excludes many context-free or even regular languages, but includes some context-sensitive ones, e.g., the permutation closure of $a^n b^n c^n$.

Concerning SDL, it is straightforward to show that all context-free languages can be generated by SDL-grammars.

**Proposition 4** Every context-free language is generated by some SDL-grammar.

**Proof.** We can use a the standard transformation of an arbitrary cfr. grammar $G = (N, T, P, S)$ to a categorial grammar $G'$. Since $\Rightarrow$ does not appear in $G'$ each SDL-proof of a lexical assignment must be also an $\Rightarrow_L$ proof, i.e. exactly the same strings are judged grammatical by SDL as are judged by $L$. □

Note that since the $\{(A x), (/ L), (\langle L)\}$ subset of $L$ already accounts for the cfr. languages, this observation extends to SDL$^\perp$.

Moreover, some languages which are not context-free can also be generated.

**Example.** Consider the following grammar $G$ for the language $a^n b^n c^n$. We use primitive types $B = \{b, c, x, y, z\}$ and define the lexical map for $\Sigma = \{a, b, c\}$ as follows:

$$
\begin{align*}
l(a) &= \{ x/(c \Rightarrow (b \Rightarrow x)), x/(c \Rightarrow (b \Rightarrow y)) \} \\
l(b) &= \{ (y/b)/y, (y/b)/z \} \\
l(c) &= \{ (x/c)/z, z/c \}.
\end{align*}
$$

The distinguished primitive type is $x$. To simplify the argumentation, we abbreviate types as indicated above.

Now, observe that a sequent $U \Rightarrow x$, where $U$ is the image of some string over $\Sigma$, only then may have balanced primitive counts, if $U$ contains exactly one occurrence of each of $A_2$, $B_2$ and $C_2$ (accounting for the one supernumerary $x$ and balanced $y$ and $z$ counts) and for some number $n \geq 0$, $n$ occurrences of each of $A_1$, $B_1$, and $C_1$ (because, resource-oriented speaking, each $B_i$ and $C_i$ “consume” a $b$ and $c$, resp., and each $A_i$ “provides” a pair $b, c$). Hence, only strings containing the same number of $a$'s, $b$'s and $c$'s may be produced. Furthermore, due to the Subformula Property we know that in a cut-free proof of $U \Rightarrow x$, the main formula in abstractions (right rules) may only be either $c \Rightarrow (b \Rightarrow X)$ or $b \Rightarrow X$, where $X \in \{x, y\}$, since all other implication types have primitive antecedents. Hence, the LHS of any sequent in the proof must be a subsequence of $U$, with some additional $b$-types and $c$-types interspersed. But then it is easy to show that $U$ can only be of the form

$$
A_1^n, A_2, B_1^n, B_2, C_1^n, C_2,
$$

since any $/$ connective in $U$ needs to be introduced via $(/L)$.

It remains to be shown, that there is actually a proof for such a sequent. It is given in Figure 3.

The sequent marked with * is easily seen to be derivable without abstractions.

A remarkable point about SDL’s ability to cover this language is that neither L nor LP can generate it. Hence, this example substantiates the claim made in
(Moortgat 94) that the inferential capacity of mixed
Lambek systems may be greater than the sum of
its component parts. Moreover, the attentive reader
will have noticed that our encoding also extends to
languages having more groups of n symbols, i.e., to
languages of the form \(a_1^n a_2^n \ldots a_k^n\).

Finally, we note in passing that for this grammar the
rules \(\frac{R}{R}\) and \(\frac{L}{L}\) are irrelevant, i.e. that it is at
the same time an SDL- grammar.

3 NP-Completeness of the Parsing
Problem

We show that the Parsing Problem for SDL-
grammars is NP-complete by a reduction of the
3-Partition Problem to it.\(^6\) This well-known NP-
complete problem is cited in (GareyJohnson 79) as
follows.

Instance: Set \(\mathcal{A} = \{A_1, A_2, \ldots, A_m\}\) of 3m elements, a bound \(N \in \mathbb{N}\), and a size \(s(a) \in \mathbb{N}\)
for each \(a \in \mathcal{A}\) such that \(\frac{N}{3} < s(a) < \frac{N}{2}\) and
\(\sum_{a \in \mathcal{A}} s(a) = mN\).

Question: Can \(\mathcal{A}\) be partitioned into \(m\) disjoint
sets \(A_1, \ldots, A_m\) such that, for

\[1 \leq i \leq m, \quad \sum_{a \in A_i} s(a) = N\]  

(note that each \(A_i\) must therefore contain
exactly 3 elements from \(\mathcal{A}\)?)

Comment: NP-complete in the strong sense.

Here is our reduction. Let \(\Gamma = (\mathcal{A}, m, N, s)\)
be a given 3-Partition instance. For notational conve-
nience we abbreviate \((\ldots ((A/B_1)/B_2)/\ldots)/B_n\) by
\(A/B_n \ldots B_2 \cdot B_1\) and similarly \(B_n \rightarrow (\ldots (B_1 \rightarrow A)\ldots)\) by
\(B_n \cdot B_{n-1} \cdot B_1 \cdot A\), but note that this
is just an abbreviation in the product-free fragment.
Moreover the notation \(A^k\) stands for

\[A \cdot A \cdot \ldots \cdot A\]  
k times

We then define the SDL-grammar \(G_\Gamma = (\Sigma, F, b_S, I)\)
as follows:

\[
\Sigma := \{v, w_1, \ldots, w_{3m}\} \\
F := \text{all formulae over primitive types} \\
b_S := \{a, d\} \cup \bigcup_{i=1}^{m} \{b_i, c_i\} \\
l(v) := a/(b_1^3 \cdot b_2^3 \cdot \ldots \cdot b_m^3 \cdot c_1^N \cdot c_2^N \cdot \ldots \cdot c_m^N \rightarrow d) \\
l(w_i) := \bigcup_{1 \leq j \leq m} d/d \cdot b_j \cdot c_j^{s(a_i)} \\
l(w_{3m}) := \bigcup_{1 \leq j \leq m} d/d \cdot b_j \cdot c_j^{s(a_{3m})}
\]

\[\text{The word we are interested in is } v w_1 w_2 \ldots w_{3m}.\]
We do not care about other words that might be
generated by \(G_\Gamma\). Our claim now is that a given
3-Partition problem \(\Gamma\) has a solution if and only if
\(v w_1 \ldots w_{3m}\) is in \(L(G_\Gamma)\). We consider each direction
in turn.

Lemma 5 (Soundness) If a 3-Partition problem
\(\Gamma = (A, m, N, s)\) has a solution, then \(v w_1 \ldots w_{3m}\)
is in \(L(G_\Gamma)\).

Proof. We have to show, when given a solution to \(\Gamma\),
how to choose a type sequence \(U \in l(v w_1 \ldots w_{3m})\)
and construct an SDL proof for \(U \Rightarrow a\). Suppose
\(A = \{a_1, a_2, \ldots, a_{3m}\}\). From a given solution (set of
triples \(A_1, A_2, \ldots, A_m\) we can compute in poly-
nomial time a mapping \(k\) that sends the index of
an element to the index of its solution triple, i.e.,
k(i) = j iff \(a_i \in A_j\). To obtain the required sequence
\(U\), we simply choose for the \(w_i\) terminals the type
d/d \cdot b_{k(i)}(s(a_i)) \cdot c_{k(i)}^{s(a_i)} (resp. d/b_{k(3m)}(s(a_{3m})) \cdot c_{k(3m)}^{s(a_{3m})}) for \(w_{3m}\). Hence
the complete sequent to solve is:

\[
a/(b_1^3 \cdot b_2^3 \cdot \ldots \cdot b_m^3 \cdot c_1^N \cdot c_2^N \cdot \ldots \cdot c_m^N \rightarrow d) \\
d/d \cdot b_{k(1)}(s(a_1)) \\
\vdots \\
(*) \quad d/d \cdot b_{k(3m-1)}(s(a_{3m-1})) \cdot c_{k(3m-1)}^{s(a_{3m-1})} \\
d/b_{k(3m)}(s(a_{3m-1})) \cdot c_{k(3m)}^{s(a_{3m})} \\
\Rightarrow a
\]

Let \(a/B_0, B_1, \ldots B_{3m} \Rightarrow a\) be a shorthand for \((*)\),
and let \(X\) stand for the sequence of primitive types
\(b_{k(3m)}(s(a_{3m})) \cdot c_{k(3m-1)}^{s(a_{3m-1})} \cdot k(3m) \cdot b_{k(3m-1)}(s(a_{3m-1})) \cdot \ldots \cdot b_{k(1)}(s(a_1)) \cdot c_{k(1)}^{s(a_1)} \cdot k(1)\).

Using rule \((/L)\) only, we can obviously prove
\(B_1, \ldots B_{3m}, X \Rightarrow d\). Now, applying \((- \rightarrow R)\) \(3m + N\) times we can obtain \(B_1, \ldots B_{3m} \Rightarrow B_0\), since there
are in total, for each \(i\), \(3\) \(b_i\) and \(N\) \(c_i\) in \(X\). As final
step we have

\[
B_1, \ldots B_{3m} \Rightarrow B_0 \Rightarrow a \Rightarrow a \Rightarrow a \\
\]

which completes the proof. \(\square\)

Lemma 6 (Completeness) Let \(\Gamma = (A, m, N, s)\)
be an arbitrary 3-Partition problem and \(G_\Gamma\) the
corresponding SDL-grammar as defined above. Then \(\Gamma\)
has a solution, if \(v w_1 \ldots w_{3m}\) is in \(L(G_\Gamma)\).

Proof. Let \(v w_1 \ldots w_{3m} \in L(G_\Gamma)\) and

\[a/(b_1^3 \cdot \ldots \cdot b_m^3 \cdot c_1^N \cdot \ldots \cdot c_m^N \rightarrow d), B_1, \ldots B_{3m} \Rightarrow a\]

be a witnessing derivable sequent, i.e., for \(1 \leq i \leq 3m\), \(B_i \in l(w_i)\). Now, since the counts of this
sequent must be balanced, the sequence \(B_1, \ldots B_{3m}\)

99
must contain for each $1 \leq j \leq m$ exactly 3 $b_j$ and exactly $N$ $c_j$ as subformulae. Therefore we can read off the solution to $\Gamma$ from this sequent by including in $A_j$ (for $1 \leq j \leq m$) those three $a_i$ for which $B_i$ has an occurrence of $b_j$, say these are $a_{j(1)}$, $a_{j(2)}$ and $a_{j(3)}$. We verify, again via balancedness of the primitive counts, that $s(a_{j(1)}) + s(a_{j(2)}) + s(a_{j(3)}) = N$ holds, because these are the numbers of positive and negative occurrences of $c_j$ in the sequent. This completes the proof. □

The reduction above proves NP-hardness of the parsing problem. We need strong NP-completeness of 3-Partition here, since our reduction uses a unary encoding. Moreover, the parsing problem also lies within NP, since for a given grammar $G$ proofs are linearly bound by the length of the string and hence, we can simply guess a proof and check it in polynomial time. Therefore we can state the following:

**Theorem 7** The parsing problem for SDL is NP-complete.

Finally, we observe that for this reduction the rules $(/R)$ and $(\backslash R)$ are again irrelevant and that we can extend this result to SDL$^\sim$.

### 4 Conclusion

We have defined a variant of Lambek’s original calculus of types that allows abstracted-over categories to freely permute. Grammars based on SDL can generate any context-free language and more than that. The parsing problem for SDL, however, we have shown to be NP-complete. This result indicates that efficient parsing for grammars that allow for large numbers of unbounded dependencies from within one node may be problematic, even in the categorial framework. Note that the fact, that this problematic case doesn’t show up in the correct analysis of normal NL sentences, doesn’t mean that a parser wouldn’t have to try it, unless some arbitrary bound to that number is assumed. For practical grammar engineering one can devise the motto avoid accumulation of unbounded dependencies by whatever means.

On the theoretical side we think that this result for SDL is also of some importance, since SDL exhibits a core of logical behaviour that any (Lambek-based) logic must have which accounts for non-peripheral extraction by some form of permutation. And hence, this result increases our understanding of the necessary computational properties of such richer systems. To our knowledge the question, whether the Lambek calculus itself or its associated parsing problem are NP-hard, are still open.

### References

J. van Benthem. The Lambek Calculus. In R. T. O. et al. (Ed.), *Categorial Grammars and Natural Language Structures*, pp. 35–68. Reidel, 1988.

M. R. Garey and D. S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, Cal., 1979.

J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

E. König. LexGram - a practical categorial grammar formalism. In *Proceedings of the Workshop on Computational Logic for Natural Language Processing. A Joint COMPULOGNET/ELSNET/EAGLES Workshop*, Edinburgh, Scotland, April 1995.

J. Lambek. The Mathematics of Sentence Structure. *American Mathematical Monthly*, 65(3):154–170, 1958.

P. Lincoln and T. Winkler. Constant-Only Multiplicative Linear Logic is NP-Complete. *Theoretical Computer Science*, 135(1):155–169, Dec. 1994.

M. Moortgat. Residuation in Mixed Lambek Systems. In M. Moortgat (Ed.), *Lambek Calculus. Multimodal and Polymorphic Extensions, DYANA-2 deliverable R1.1.1.B*. ESPRIT, Basic Research Project 0852, Sept. 1994.

G. Morrill. *Type Logical Grammar: Categorial Logic of Signs*. Kluwer, 1994.

M. Pentus. Lambek grammars are context free. In *Proceedings of Logic in Computer Science*, Montreal, 1993.