Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A STUDY OF VARIANTS OF THE LAMBEK CALCULUS AND AN IMPLEMENTATION IN PROLOG

S.-T. Huang
D. S. Parker

June 1992
CSD-920028

A Study of Variants of The Lambek Calculus and An Implementation in Prolog [1]

Shen-Tzay Huang and D. Stott Parker

sthuang@cs.ucla.edu stott@cs.ucla.edu

Computer Science Department

University of California

Los Angeles, CA 90024-1596 USA

Abstract

The Lambek Calculus was proposed as a syntactic calculus of expressions of natural languages. A computation oriented analysis of one of its variants leads to the discovery of the property of local permutability; and the theorems and proofs suggest directions for useful extensions of the systems to meet concerns of specific applications, beyond linguistics. A couple of such possible variants are suggested and discussed. Some recently proposed linguistic extensions are shown to be achieved in those variants, from an algebraic consideration independent from the linguistic proposal. The study also pinpoints the similarity and differences between Categorial Grammars and Phrase Structure Grammars, from a computational perspective. An implementation in Prolog is discussed and linked with the modularity and compositionality features of the calculus, an instance of gain through interdisciplinary interplay.

[1] This research is supported by NSF grant IRI-8917907

Contents

1 Introduction

The Lambek Calculus [La 58] is a formalism, proposed for the mathematization of symbolic manipulation of expressions in natural languages. As it was first developed as a mathematical system, notions of calculus, algebraic structures and formal deductions were already incorporated in. With its decidability established as a result similar to the Gentzen Cut Elimination Theorem [La 58], the system serves as a very nice model with many interesting linguistic and computational variants.

Yet, historically, as it was primarily applied to formal linguistic studies, a proof of weak equivalence of The Lambek Calculus with context free grammar almost put a stop to its exploration in linguistic study, not to mention its much less explored computational aspects. Only after phrase structure was no longer treated as a deficit [PG 82] in linguistic study and paradigmatic diversifications were resurrected in linguistics, did the Lambek Calculus and the associated Categorial Grammars revive and find renewed perspectives of inquiry [OBW 88]. In particular, the previous identification with context free grammar is now known to be true only for a simple variant of the Lambek Calculus, the Ajdukeiwicz-Bar-Hillel Grammars.

In this article, we study a variant of the Lambek Calculus, called the Simple Lambek Calculus in [Ben 88], from a computational and formal perspective. Although we draw comparisons with and borrow ideas from Categorial Grammar-based linguistic studies, we are not really addressing linguistic applications. Rather, we treat the system more as a formal system, a calculus, and a set of computational mechanisms, as it is, abstracted from its linguistic significance, and try out variants toward possible computational applications. However, we do expect that ignoring linguistic relevance in the beginning will still maintain its potential for linguistic studies, as well as for symbol processing.

The article is organized as follows. In section 2, we first introduce the Simple Lambek Calculus and establish its capacity beyond the formalism of context free grammars. This serves as a basis for later exploration. Next, in section 3, we compare the Simple Lambek Calculus with categorial grammar, and prove the difference between these two, from a computational perspective. The key emphasis is on the roles of the M and G rules and their interaction, in this categorial formulation. Several theorems about the properties of this categorial version are established . Furthermore, the theorem and proof themselves shed light on the comparison with phrase structure based formalisms, and suggest directions for possible variations. In section 4, based upon the results established, we pinpoint about the difference between Categorial Grammars and Phrase Structure Grammars, whose similarity and potential research parallels have been observed and advocated, for example, in [Po 88]. In section 5, we describe an implementation in Prolog and discuss further features of Logic Programming, in particular, unification and logical variables. These give not only an edge in implementation, but also that in conceptual organizations and flexibility for variations. We also learn how the principle of compositionality and related notions embodied in linguistic studies help modular development of the implementation. It is a fruitful instance of interdisciplinary interplay. Then in section 6, we propose possible alternative Lambek variants, and point out in remarks how algebraic variants would cover some linguistically proposed extensions. In particularly, we can account for the Multiple Dependence in [St 88], which is not possible in the original Lambek Calculus, nor in the Simple Lambek Calculus. Finally, we summarize the article and offer conclusions and directions for further researches.

2 The Simple Lambek Calculus

2.1 Lambek Calculi

Recently there has been revived interest in categorial grammar and its variants. Much of this interest has come for its potential in linguistics, morphology, syntax and semantics and natural language processing and acquisition, and [OBW 88] is recommended as a collective survey. The Lambek Calculus, as the common

formal system, has then enjoyed more detailed scrutiny and extensions in many directions. Variants of the Lambek Calculus are being proposed, to extend its power, or to illustrate applications. Classical issues regarding its weak equivalence with context free languages are now being put in doubt, or simply left as unimportant issues as the paradigm and foci of research have evolved.

Among many others, a simple version, *The Simple Lambek Calculus*, a non-directional and non-associative variant of the original Lambek Calculus, was proposed in [Ben 88], as an introduction to its variants and framework for accounts of formal semantics and deductive power, with respect to intuitionistic propositional logic. This system differs slightly from the original Lambek Calculus, in its being non-directional and having no product categories. The former means the allowance of arguments on either sides in textual order of functors. The latter simplifies the set of possible categories and thus consideration of associativity of internal structures of type categories.

As a common generic component of any Lambek Calculus, type or category terms are constructed by inductive closure of the following rules :

- All basic types, usually, e for entity and t for truth values, are type terms.
- if a and b are type terms, then so is (a, b), called a compound type.

The compound type term, (a, b), has an intuitive interpretation as denoting the functions or mappings with domain of type a and codomain of type b.

Historically, in the formal categorial paradigm in linguistics, another function-argument terminology is used for compound types (a, b) where a is referred to as the argument type and b as the functor type. Furthermore, the distinction between types and categories as syntactic types and semantic categories, for strict formal usages of terminology, is not enforced here as it is irrelevant at the level of detail we pursue.

The following is the formal system for the Simple Lambek Calculus, a sequent calculus with three rules of inference, where a_i, b, c, are all metavariables ranging over type terms and ";" is an associative connectives for type terms.

Sequent Formula			
	$a_1; \cdots; a_m$	$\Rightarrow$	$b, \quad m \geq 1$
Axiom			
	a	$\Rightarrow$	a
Rule of Inference			
(,) elimination			
	$a_1; \cdots; a_i$	$\Rightarrow$	b
		and	
	$a_{i+1}; \cdots; a_n$	$\Rightarrow$	(b, c)
		$\vdash$	
	$a_1; \cdots; a_n$	$\Rightarrow$	c
Left (,) introduction			
	$a_1; \cdots; a_n; b$	$\Rightarrow$	c
		$\vdash$	
	$a_1; \cdots; a_n$	$\Rightarrow$	(b, c)
Right (,) introduction			
	$b; a_1; \cdots; a_n$	$\Rightarrow$	c
		$\vdash$	
	$a_1; \cdots; a_n$	$\Rightarrow$	(b, c)

2.2 Properties of The Simple Lambek Calculus

Important theoretical characteristics of this simple system are listed and compared with other formal systems such as the Lambda Calculus, Heyting Algebras [Ben 88], and biclosed monoidal categories [La 88]. For our discussion, the following properties are most related to our investigation and comparison (as the proof for Permutation Closure was not provided in [Ben 88], we do it for the sake of self assurance and of self-containment of the article) :

Lemma. 1 (M Rule Lemma)

$$\vdash \qquad a \Rightarrow ((a,b),b)$$

for arbitrary types a and b.

Proof. 1

1)	a	$\Rightarrow$	a	*Axiom*
2)	(a,b)	$\Rightarrow$	(a,b)	*Axiom*
3)	$a;(a,b)$	$\Rightarrow$	b	*(,) Elim*
4)	a	$\Rightarrow$	$((a,b),b)$	*Right (,) Intro*

□

Theorem. 1 (Permutation Closure [Ben 88]) *if $a_1;\cdots;a_n \Rightarrow b$ is provable in the simple Lambek Calculus, then so is $a_{\pi(1)};\cdots;a_{\pi(n)} \Rightarrow b$, for any permutation π of $1,\cdots,n$.*

Proof. 2 *We prove this by induction on the length of antecedents.*

- *Basis : $n = 2$.*

  Given $a_1;a_2 \Rightarrow b$, we have

1)	$a_2 \Rightarrow (a_1,b)$	*Left (,) Intro*
2)	$a_1 \Rightarrow ((a_1,b),b)$	*M Rule Lemma*
3)	$a_2;a_1 \Rightarrow b$	*1),2) (,) Elim*

- *Inductive step : $n = m + 1$.*

  We need to prove that a_1 can be inserted into any of the $m+1$ positions between the remaining permuted m elements. The first case is similar to the basis case. So, we only need to consider how the insert a_1 into an arbitrary place.

1)	$a_1;\cdots;a_{m+1} \Rightarrow b$	*Ind. Hypo.*
2)	$a_2;\cdots;a_{m+1} \Rightarrow (a_1,b)$	*Left (,) Intro*
3)	$a_{i+1};\cdots;a_{m+1};a_2;\cdots;a_i \Rightarrow (a_1,b)$	*Ind. Hypo.*
4)	$a_1 \Rightarrow ((a_1,b),b)$	*M Rule Lemma*
5)	$a_1;a_{i+1};\cdots;a_{m+1};a_2;\cdots;a_i \Rightarrow b$	*3),4) (,) Elim*
6)	$a_1;a_{i+1};\cdots;a_{m+1};a_2;\cdots;a_{i-1} \Rightarrow (a_i,b)$	*Right (,) Intro*
7)	$a_i \Rightarrow ((a_i,b),b)$	*M Rule Lemma*
8)	$a_i;a_1;a_{i+1}\cdots;a_{m+1};a_2;\cdots;a_{i-1} \Rightarrow b$	*6),7) (,) Elim*
9)	$\cdots$	*Repeat 6)-8)*
10)	$a_2;\cdots;a_i;a_1;a_{i+1};\cdots;a_{m+1} \Rightarrow b$	

This completes the the proof. □

With proper definition of **acceptance**, confirming the slogan of *parsing as deduction*, it is easy to establish the non-context freeness of this simple Lambek Calculus, as a corollary.

Definition. 1 (Acceptance) *A sequent of types, $a_1; \cdots; a_n$ is accepted by the simple Lambek Calculus iff $a_1; \cdots; a_n \Rightarrow s$ is a theorem of the calculus, where s is a special predetermined basic type symbol.*

In categorial analysis of natural languages, each expression, or sequences of words, is assigned finite number of types from its categorial grammar. Such type assignment is called lexical type assignment. Of course, the categorization of lexicons definitely bears a lot of linguistic analysis, however, we only need to note that it is a finite substitution as defined in [HU 79]. Given that, it is easy to have the corollary.

Corollary. 1 *The set of word sequences, after any lexical type assignment, acceptable by the simple Lambek Calculus, is not context free.*

Proof. 3 *Given that lexical type assignment is essentially an ϵ-free finite substitution $\mathcal{H}$, from standard results concerning closure properties of context free languages under substitution and intersection with regular sets, and closure properties of regular sets [HU 79], we see that :*

$$\begin{aligned} \mathcal{L} &= \{a^n b^n c^n\} \\ &= \{a^* b^* c^*\} \cap permutation(\mathcal{L}). \end{aligned}$$

and $\mathcal{L}$ is not context free, so permutation($\mathcal{L}$) is not context free. Yet, permutation($\mathcal{L}$) is accepted by the simple Lambek Calculus by the ϵ-free finite substitution $\mathcal{H}$, with the special type symbol t :

$$\begin{aligned} \mathcal{H}(a) &\mapsto e \\ \mathcal{H}(b) &\mapsto (e, s) \\ \mathcal{H}(c) &\mapsto (s, t) \\ \mathcal{H}(c) &\mapsto (s, (t, t)) \end{aligned}$$

Since $\mathcal{L}$ is not context free, neither is permutation($\mathcal{L}$), nor in turn the ϵ-free finite substitutions of $\mathcal{L}$. We have the result that Simple Lambek Calculus is more expressive than any CFG. □

We note that permutation of a regular set would give us a non-context free set too. For instance, in the above,

$$\mathcal{L} = permutation((abc)^*) \cap \{a^* b^* c^*\}.$$

3 Extended Categorial Grammar

While the Lambek Calculus is the fundamental theory for various categorial grammars, the formal calculus is not employed directly in those applications. Rather, most are described by a set of category formation rules and combination rules. We adopt a slightly notational change to retain the resemblance with the linguistic usage, namely, a/b as functor a and argument b, which is the same as (b, a) in the Simple Lambek Calculus, while **undirectionality and non-associativity** are kept intact. The following are the most widely used set of rules, with annotation for later references :

Function Application :			
	$F/X; X$	$\Rightarrow_{FA}$	F
	$X; F/X$	$\Rightarrow_{FA}$	F
Function Composition :			
	$F/X; X/Y$	$\Rightarrow_{FC}$	F/Y
	$X/Y; F/X$	$\Rightarrow_{FC}$	F/Y
M (Montague) Rule :			
	A	$\Rightarrow_{M}$	$X/(X/A)$
G (Geach) rule :			
	X/Y	$\Rightarrow_{G}$	$(X/Z)/(Y/Z)$

Variants and extensions correspond to different sets of combination rules. Among others, Steedman's Extended Categorial Grammar [St 88] relates it to and extends it with various combinators of combinatory logic.

While it is easy to establish that the above rules are theorems of the Simple Lambek Calculus, it is not clear whether they form an axiomatic basis of the Simple Lambek Calculus. Also it is difficult to quantify the expressive power and overall dynamic behavior of categorial grammars using these rules.

As a matter of fact, we can prove that the combination of these rules is strictly less powerful than the Simple Lambek Calculus. In particular, global permutability is not true in the system of those four rules.

3.1 Basic Properties

Lemma. 2 *Functional Composition is redundant (i.e., it is derivable from other rules).*

Proof. 4 *Each use of the rule of Functional Composition can be replaced by a step of the G rule plus a step of Function Application :*

$$\Rightarrow_{FC} \quad \cong \quad \Rightarrow_{FA} \circ \Rightarrow_{G}$$

$$\begin{array}{lll} F/X; X/Y & \Rightarrow_{G} & (F/Y)/(X/Y); (X/Y) \\ & \Rightarrow_{FA} & F/Y \end{array}$$

□

As a remark, we note that the **undirectionality** can be reconciled as the composition of one **M** step and one Function Application too, as :

$$\begin{array}{lll} X; F/X & \Rightarrow_{M} & F/(F/X); F/X \\ & \Rightarrow_{FA} & F \end{array}$$

But, for reference to the Simple Lambek Calculus and λ-terms representing different derivations, we maintain the assumption of **undirectionality** in the article instead.

Theorem. 2 (Local Permutability) *All permutations of a (n-1)-ary functor and its (n-1) arguments are acceptable sequences. Namely, if π is a permutation of $1, \cdots, n$:*

$$\begin{array}{llll} \textit{If} & a_1; \cdots; a_n & \Rightarrow^{*}_{FA} & a \\ \textit{then} & a_{\pi(1)}; \cdots; a_{\pi(n)} & \Rightarrow^{*}_{(FA,G,M)} & a \end{array}$$

Proof. 5 *We note that both M and G rules are only applied to single type terms, and thus so called type shifting rules, without reducing or combining terms. Without loss of generality, as only FA is used to operate on adjacent terms, we can simply assume all the arguments are of simple types, i.e. not functional types, and the functor is of type* $(((a/a_{n-1})/a_{n-2})\cdots/a_1)$.

We prove the theorem by induction :

- *Base case :* $n = 2$.

  This is obvious, since the four rule system is **undirectional** *for the terms in antecedents :*

$$\begin{array}{lll} a/a_1; a_1 & \Rightarrow_{FA} & a \\ a_1; a/a_1 & \Rightarrow_{FA} & a \end{array}$$

- *Inductive case :* $n = m + 1$.

  Using the notion of Currying, we can have a (m)-ary functor act as a (m-1)-ary functional. To be precise, we can treat the functor, $((a/a_m)\cdots/a_1)$ *as equal to* $((b/a_{m-1})\cdots/a_1)$, *with new pseudo type symbol* $b = (a/a_m)$. *By the induction hypothesis and undirectional Function Application, we only need to consider the case when* a_m *is immediately to the right of this (m)-ary functor and the other arguments are all to the right of this* a_m, *since other all other cases are reducible to cases for* $k \leq m-1$ *by treating the (m)-ary functor as (k)-ary functionals first, with* $k \leq m-1$, *and using the induction hypothesis. In other words,* $(((a/a_m)\cdots/a_k)\cdots/a_1); a_k$ *is the same as* $((b/a_k)\cdots\cdots/a_1); a_k$; *so the induction hypothesis can be applied.*

$$\begin{array}{ll} & ((a/a_m)\cdots/a_1); a_m \\ \Rightarrow_M & ((a/a_m)\cdots/a_1); (a/(a/a_m)) \\ \Rightarrow_G & ((a/a_m)\cdots/a_1); ((a/a_{m-1})/((a/a_m)/a_{m-1})) \\ \Rightarrow_G^{m-2} & ((a/a_m)\cdots/a_1); (((a/a_{m-1})\cdots/a_1)/(((a/a_m)/a_{m-1})\cdots/a_1)) \\ \Rightarrow_{FA} & ((a/a_{m-1})\cdots/a_1) \end{array}$$

  By the induction hypothesis, we have the result for $n = m+1$.

Thus, the induction is completed and we have the theorem. □

As an observation from the proof, we note that for each displaced argument, we need an M step and a proportional number of G steps to proceed. The worse case for an (n+1)-ary functor with n arguments, requires altogether an (n-1) extra M steps and $(n \times (n-1)/2)$ G steps, in addition to n normal FA steps, to recover the normal Function Application-only derivations.

$$\begin{array}{ll} & \text{Total number of steps} \\ = & \{\sum_{m=2}^{n}(1 \times M + (m-1) \times G + 1 \times FA)\} + 1 \times FA \\ = & (n-1) \times M + (n \times (n-1)/2) \times G + n \times FA \end{array}$$

The Geach (G) rule together with Montague (M) rule, as seen from the proof, allow a generalization of ordinary Function Application, where the temporal ordering of applications to arguments, implicitly dictated by the functor structure, is made free. Since Functional Composition is obtainable as one G step and one Function Application, while using multiple instances of G steps on an argument term is indispensable for the establishment of the theorem of Local Permutability, it is obvious that inclusion of the G rule is strictly more powerful than Functional Composition, as a corollary.

Corollary. 2

$$\mathcal{L}(\{FA, FC, M\}) \overset{\subset}{\neq} \mathcal{L}(\{FA, G, M\})$$

Note, the local permutability is a properties between the functor and its arguments in immediate lower layer. The property of course holds between any adjacent layers of functor and its arguments. So the notion of Immediate Dominance (ID) and Constituent is available. The former mean the functor dominates its arguments in terms of combination structure, and the latter refers to the recursively constructed terms according to such dominance and combination. The theorem also establishes that the temporal ordering, Linear Ordering, of arguments is irrelevant, as far as functor-argument relation, or now ID, is concerned. Of course, natural languages and specific linguistic analysis would suggest constraints on the ID and possible Linear Ordering.

Other linguistic paradigms, in particular GPSG, have same characteristic notions as Constituent, ID and Linear Ordering [Po 88]. Observation of such concepts from local permutability demonstrates the close relationship of Lambek Calculus and Categorial Grammars with GPSG. We will say more about such comparison in next section.

In addition, these observation of constituency and local permutability which leads to our invention of a counterexample to the global permutability allowed in the Simple Lambek Calculus, and thus the difference in expressive power.

Before stating that as a theorem and proving it, we need some more observations about the M and G steps.

3.2 Properties of the M and G rules

Since only Function Application can combine type terms, and combination is only done for adjacent terms, we have to appeal to the G and M rules to change, or more precisely, upgrade, term types to allow Function Application to succeed in reducing the number of type terms in the sequence. Because of the restriction of adjacency and purpose of reduction by Function Application, we only need to examine contexts of possible functor-argument structures of adjacent terms with respect to which G and M rules can be applied and then enable Function Application, and the effects on the structure of resultant type terms.

For M steps :

- Case 1 : no relation between the twe terms.

$$\begin{array}{lll} X/b;a & \Rightarrow_M & X/b;b/(b/a) \\ & \Rightarrow_G & (X/(b/a))/(b/(b/a));b/(b/a) \\ & \Rightarrow_{FA} & X/(b/a) \end{array}$$

  This step combines unrelated argument, a, into its argument type by raising the argument type. Note further application of M steps or G steps to the second terms will not get us further, since it will prevent the unification with b in the first term, X/b. Also note, the unrelated second term will be kept in the lower part of the right side of the argument type as a result.

- Case 2 : Second term as part of the internal functor structure.

$$\begin{array}{lll} (b/a)/X;a & \Rightarrow_M & (b/a)/X;b/(b/a) \\ & \Rightarrow_G & (b/a)/X;(b/X)/((b/a)/X) \\ & \Rightarrow_{FA} & b/X \end{array}$$

  This step consumes a displaced normal argument. It is a kind of local permutation of arguments, an extended form of Function Application; and the general case was established in the proof of the theorem of Local Permutability.

- Case 3 : Second term as argument of first term.

$$\begin{array}{lll} (b/a);a & \Rightarrow_M & (b/a);b/(b/a) \\ & \Rightarrow_{FA} & b \end{array}$$

This is just a variant of normal Function Application. In general, this includes cases when the first term's argument type can *exactly* be G- or M- raised from the second term. Otherwise, it is the same as totally unrelated as case 1, e.g. :

$$\begin{array}{lll} b/(c/(c/a));a & \Rightarrow_M & (b/(c/(c/a));c/(c/a) \\ & \Rightarrow_{FA} & b \end{array}$$

Though it has effects on semantic aspect of changing the internal functor-argument meaning structure of constituents, there is no difference as far as the reduction of type terms and its resultant syntactic type is concerned.

As the formation of resultant type after a M step, namely, two instances of arbitrary type are stacked over the atomically treated original type, these cases exhaust effectiveness of applying the M rule.

Similarly for G steps, but notice now the two instances of arbitrary type are distributed into functor and argument part of a functional type term :

- Case 1 : Functor part of second term appears in the argument part of the first term

$$\begin{array}{lll} X/(a/c);a/b & \Rightarrow_G & X/(a/c);((a/c)/(b/c)) \\ & \Rightarrow_G & (X/(b/c))/((a/c)/(b/c));((a/c)/(b/c)) \\ & \Rightarrow_{FA} & X/(b/c) \end{array}$$

This step applies the argument a into its raised argument type by replacing the internally outermost type in the compound argument type.

Note the case when the argument of second term appears in the argument of the first term, i.e. $X/(b/c);a/b$, has to be treated as a totally unrelated case, since neither G nor M steps on second term can raise the argument term (b) to the functor position in the argument of first term. Furthermore, the functor term a in the second term must be exactly the same as the outermost functor term of the argument term of the first term; otherwise, e.g. $X/(z/a)/c;(a/b)$, neither term G nor M steps would keep the whole second term as atomic and so could not exactly match the functor term in the argument of the first term.

Note also, as a result of above derivation, the position of unrelated argument (c) is intact.

In the degenerate case where $c = b$, then Function Application, instead of above sequence of derivation, should have been applied. Otherwise, divergent results would occur, namely X/(b/b) vs. X, the former being produced by above derivation whereas the latter by direct Function Application. In general, the degenerate case includes cases when arguments can be G- or M-raised from the second term :

$$\begin{array}{lll} X/((a/c)/(b/c));a/b & \Rightarrow_G & X/((a/c)/(b/c));((a/c)/(b/c)) \\ & \Rightarrow_{FA} & X \end{array}$$

- Case 2 : argument of the second term appears in the functor of first term

$$\begin{array}{lll} (b/z)/X;a/b & \Rightarrow_G & (b/z)/X;(a/z)/(b/z) \\ & \Rightarrow_G & (b/z)/X;((a/z)/X)/((b/z)/X) \\ & \Rightarrow_{FA} & (a/z)/X \end{array}$$

This step changes the original functor type by replacing its internally outermost type when used as an argument type by higher order functors. Following the same reasoning as in case 1, it must be the

outermost functor in the first term that the argument of second appears, and the case for the functor of second term is again totally unrelated.

It is in effect an variant of Functional Composition, in turn an extended form of Functional Application. A degenerate case is when X or z is nil, which then corresponds to the derivation of Functional Composition from G step and Function Application.

- Case 3 :

$$\begin{array}{lll} (b/a);a/b & \Rightarrow_G & (b/a);(a/a)/(b/a) \\ & \Rightarrow_{FA} & a/a \end{array}$$

This is actually another degenerate case 2, when z = nil and $X = a$. Or, it is a case satisfying both case 1 and 2. Once again, divergent results can occur, depending upon which type term, (b/a) or a/b, is first G-raised and then combined by Function Application, we get b/b or a/a, respectively. This is different from the degenerate case in case 1, though. Because, as far as the Calculus is concerned, they both are valid, since they are all in form of axioms. Yet, $X/(b/b)$ is not an axiom.

To summarize the major observations, we note that although unrelated arguments can be kept in the compound argument type, they must be kept in stack-like ordering. Furthermore, the stack-ordering compound type prohibits the normal consumption of arguments occurring later. This is due to a constraint of outmostness only is imposed as a result of stacking in the compound argument type, and consequently, iterative G steps can only used to raise simple types to replace the compound type as whole, rather than simple types of the compound. On the other hand, internal functor type structure is free to later combination with its atomic arguments, as long as they are indeed genuine atomic arguments. This discrepancy confirms the tree-like constituent structure, and also leads to the invention of counterexamples where normal arguments of two adjacent constituents overlaps their normal arguments and consequently mutually block each other by raising normal arguments into compound types, and leads to divergent derivations from normal or intended one.

We also note that, totally unrelated adjacent type terms can be combined, with the cost of extra component types (Y's) which could not be removed later.

Example. 1 *Suppose the final goal is to take d as a sub-argument of c :*

$$\begin{array}{lll} & d & ; \ c \\ \Rightarrow_M^2 & X/(X/d) & ; \ Y/(Y/c) \\ \Rightarrow_G & X/(X/d) & ; \ (Y/Z)/((Y/c)/Z) \\ \textit{with unifier } \delta: & \{X \cong Y/c, & \quad Z \cong X/d\} \\ & (Y/c)/(Y/c)/d & ; \ (y/(y/c))/((y/c)/((y/c)/d)) \\ \Rightarrow_{FA} & & \quad y/((y/c)/d) \end{array}$$

Note that the $\Rightarrow_M^2$ and $\Rightarrow_G$ are the only rules to apply before the final FA. The former is due to the fact that both c and d are atomic types and Y/c fails to unify with d. The latter is due to the failure of unification with $X/(X/d)$, if another $\Rightarrow_M$ is applied to $Y/(Y/c)$, or useless complexity if applied to $X/(X/d)$.

Dually, if the final goal is to take c as a sub-argument of d, we shall have the result as $Y/((Y/d)/c)$. □

If c is in the form of a/b, then by simplifying the same derivation, we have the results $a/(b/d)$ or $X/((X/d)/(a/b))$.

3.3 Non Global Permutability

By previous examples and analysis of all applicable G and M steps, it is always possible to combine arbitrary sequences of type terms into a single term, though the resultant term may be loaded with excessive copies

of component types not occurring in the original sequence if not following normal composing ordering in the derivation.

Granting that, it would be reasonable to further limit the application of G or M rules, if overloaded control is to be prevented. Indeed, as commonly practiced, G rule applications are replaced by Functional Composition, which, by the above analysis, corresponds to one G step plus one Function Application step. On the other hand, $X/((X/\bullet)/\bullet)$ can be thought as a pairing constructor as in the situation when c and d are both arguments to a two-place functor with $X/((X/c)/d)$ as ordered argument. For instance,

$$\begin{array}{lllll} & (a/b)/c & ; & c & ; \ b \\ \text{Pairing} & (a/b)/c & ; & X/((X/b)/c) & \\ \text{With unifier}\delta: & X \cong a & & & \\ & (a/b)/c & ; & a/((a/b)/c) & \\ \Rightarrow_{FA} & & & a & \end{array}$$

However, such usage is actually a mixture of meta level with object level, as X was originally a meta variable ranging over type terms. A more detailed discussion would be provided in the context of the implementation in Prolog. It is good enough for readers to treat the X as a in the very beginning.

Alternatively, as a hint for generalization, since the combined term introduces undesirable excessive internal structure, we may like to have c/d, in stead of $Y/((Y/c)/d)$. This can be achieved by allowing :

$$\begin{array}{lllll} & Y/(Y/c) & \Rightarrow_{M^{-1}} & c & \\ \text{Or} & & & & \\ & Y/((Y/c)/d) & \Rightarrow_{SubM^{-1}} & y/(y/(c/d)) & \\ & & \Rightarrow_{M^{-1}} & c/d & \end{array}$$

As the notation indicates, this is an inverse of M rule and is applicable to internal substructure of a type term too, rather than atomic whole in the original system. Generalization of this line is open for further investigation.

Example. 2 *The following sequence of types is an example for the non global permutability of the system with Function Application, Functional Composition, and G and M rules.*

$$\begin{array}{lllll} & (s/a)/b; & c; & b; & a/c \\ \Rightarrow^* & (s/a)/(b/c); & & b; & c/(d/b) \\ & (or\ (X/((X/c)/((s/a)/b)); & & & \\ Or & & & & \\ \Rightarrow^* & (s/a)/b; & X/((X/b)/c); & & a/c \\ & & (or\ X/((X/c)/b)); & & a/c \\ Or & & & & \\ \Rightarrow^* & (s/a)/b; & c; & & a/(c/b) \\ & & & & (or\ X/((X/b)/(a/c))) \end{array}$$

All these lead to some complicated compound terms, in which adjacent pairs are unrelated, which are not the desired simple term s. As they all have acquired non-reducible internal structures which cannot be cancelled within those four steps of Function Applications. This is due to the fact that, applications of the M or G rules introduce copies of new internal types, one in the argument part and the other in the functor part and there is no rule to convert between argument and functor, while Function Application can cancel only one of them.

On the other hand, there is indeed an derivation of the simple term s, from the above overlapped sequence

$$
\begin{array}{lll}
 & ((s/a)/b; & \underline{(a/c; \qquad\qquad c);}\ (b)) \\
\Rightarrow_{FA} & ((s/a)/b; & \underline{(a);}\ (b)) \\
\Rightarrow_{M} & ((s/a)/b; & \underline{(s/(s/a));}\ (b)) \\
\Rightarrow_{G} & \underline{((s/a)/b;} & \underline{((s/b)/((s/a)/b));}\ (b)) \\
\Rightarrow_{FA} & & \underline{((s/b);}\ \underline{(b))} \\
\Rightarrow_{FA} & & s
\end{array}
$$

In this counterexample of global permutability, the constituent structures is indicated by pairs of parentheses. Note how adjacent sub-constituents, (a/c ; c) and (b), are destroyed by overlappings in the previous case. □

With counterexamples of this kind, namely, overlapping adjacent constituents, we have conclude our comparison of the Simple Lambek Calculus with the commonly-used system of Function Application, Functional Composition, G and M rules.

Theorem. 3

$$\mathcal{L}(\{FA, FC, M, G\}) \supsetneq \mathcal{L}(\textit{Simple Lambek Calculus})$$

Remark On Context-Freeness

It is still left open so far in our study if

$$CF \stackrel{?}{=} \mathcal{L}(\{FA, FC, M, G\})$$

Since FC is redundant, the key to the answer seems to lie upon the effects of the G and M rules. Essentially they are type change rules. So if there are only finite number of possible types, where G and M rules are only principles guiding their relatedness, then the system weakly equivalent with CF, i.e. accepting same set of strings. If there are indeed infinite number of types, then by the well-known Konig's Tree Lemma, there is an infinite branch of applications of G or M rules, granting G and M rules are the type generating mechanisms. In the case, the final answer is up to empirical judgement if natural languages have infinite types, or infinite layers of intentionality as M rule was first employed. Of course, the number of natural types corresponds to complexity of processing. This may be an account for some seemingly non-context-freeness of natural language.

4 Categorial Grammar and GPSG

4.1 The Similarity

Categorial Grammar usually only employs the notion of type raising, by the M rule and Functional Composition, possibly with some language-specific contextual restrictions. Type raising systematically adds flexibility to type combination, by unilaterally changing type terms or indirectly reassigning types to lexicons. A direct result is the non-constituent, or pseudo-constituents, as named in [Do 88], in addition to standard right branching constituents in deep structures. As we have shown that the G rule is more powerful than FC, such categorial grammars are still less powerful than the system studied here, despite their linguistic reality and potential as a model of competence. On the other hand, extension of classical Categorial Grammar, in particular, correspondence with Combinators as in [St 88], and new combination modes, such as Functional Substitution, were proposed. Currently it is being subjected to further scrutiny for their properties and expressive power.

The approach of categorial grammars is described as lexicalism [Mo 88], referring to its highly condensed grammatical information into lexicon. In comparison with transformation approaches, GPSG sticks to phrase structures, and replaces transformational rules by incorporating feature-value attributes into non terminals and employing a meta rule schema to describe constraints between subtrees. Categorial grammars go one step further by pushing the phrase structures, coded as function-argument structures, into the lexicon. Our study, or the formal calculus itself, focuses only on the combinatorial aspects of syntax. Other linguistic aspects, like morphological agreement and government, though not covered here, can actually be accounted for as feature theories, by adding feature value sets to decompose or refine the classification of basic categories. As long as these feature value sets has only finite combinatorial expansions, they have no theoretical advantage in expressiveness over ther system we have described here. Logic Programming can encode those features easily by terms and unification. And variants proposed in later section can further show the flexibility of the studied system.

As we remarked previously on the notion of constituents in the proof of the theorem of local permutability, we see a close resemblance between Extended Categorial Grammars with the four combination rules, and the slash feature for subcategorization in Generalized Phrase Structure Grammar (GPSG) and its variants, such as the Head-driven Phrase Structure Grammar (HPSG) [Po 88]. The similarity is not only declaratively, in terms of notions of Immediate Dominance (ID), Constituency and major functors as Heads; but also operationally as can be seen from the proofs of theorems. In these proofs, an argument type is iteratively built up, by the shifting up of G or M steps, to resemble the internal structure of the major functor type. This computational process is the same as the matching processing of slash feature in GPSG. In brief, the result obtained not only confirms the trend toward merging those two paradigms, but actually pinpoints to the sites of agreement and disparity.

4.2 The Differences

In spite of the similarity, we note the system studied here is in fact a little more flexible than the ID relationship and the slash feature in GPSG. While such flexibility may be useful, we may like to restrict it by imposing some controls over application ordering of G, M and Function Application steps.

Example. 3 *While we have a constituent structure in following sequence,*

$$\begin{array}{lll} & (a/b; & \underline{(b/c; \quad c))} \\ \Rightarrow_{FA} & \underline{(a/b; \quad (b))} & \\ \Rightarrow_{FA} & a & \end{array}$$

the six permutations are in effect all acceptable in the system, for example :

$$\begin{array}{lll} & (b/c; & \underline{(a/b; \qquad\qquad\qquad c))} \\ \Rightarrow_{M,G} & (b/c; & \underline{((a/(b/c))/(b/(b/c)); \quad b/(b/c))} \\ \Rightarrow_{FA} & \underline{(b/c; \quad a/(b/c))} & \\ \Rightarrow_{FA} & (a) & \end{array}$$

Note the structure of parenthesis and order of Function Application is the same, namely, left branching. In general, a circular shift of a linearly structured constituents still has the above preservation property. □

5 An Implementation of FA + FC + G + M

5.1 Overview

Based upon the theoretical investigation of the system of FA + FC + G + M, and the operational characteristics, we implement a version as a universal machine in Prolog.

Several features of this implementation include modularity with respect to compositional meaning, and parameterization such as structure-building parse tree constructions. Examples from [St 88, Do 88] are used for tests, using λ−expression for representation of meaning and terms for parse trees. However, not all linguistic examples can be accepted, for instance, Multiple Dependency in [St 88], which essentially due to its combinatoric nature is out of the scope of Lambek Calculus. By that, we mean the multiset properties of occurrences are no longer true in such linguistic phenomena. On the other hand, such algebraic features can be added as variants, as we do this in the next section.

Features of Prolog, in particular, logical variables and unification, are helpful both for implementation and suggestions of extensions. In G steps or M steps, we already see uses of a kind of category variables ranging over possible types. But those meta variables denoting arbitrary types can also be used directly as object variables implemented by logical variables. Unification can then be employed to instantiate them in accordance with contextual information of neighboring types. In general, first order unification and term structure can be used for extending the space of possible types, in the Lambek Calculus. This is an advantage immediately available in Logic Programming environments like Prolog. In addition, unification and term structure is obviously relevant for incorporating feature theory to make the Extended Categorial Grammars compatible even with phrase structure approaches.

5.2 Features of The Implementation

There are experiences and observations learned from this implementation task in a logic programming environment, and with regards to linguistic and programming principles. And we would like to discuss it in this general setting, rather than specific codings. For the detailed code, please see the appendix.

- Universal Grammar, Completeness and green cuts in the implementation. Essentially, Lambek Calculus or Categorial Grammars are universal grammars. It is to account for universal linguistic phenomenon, even though English is the major focus. As a result, any implementation would serve as an universal engine. On the other hand, since there are spaces for control, as the phrase **"Algorithm = Logic + Control"** is applicable to any formal deductive system admitting computability interpretations. Lambek Calculus is surely such a system. The declarative aspect, both linguistic and algebraic knowledge, of the theory, are the properties, theorems and logical notions of derivability. The procedural aspect, without reference to its linguistic significance, corresponds to how a theorem is established in this system, namely, the process of deriving theorem. Our implementation bears such distinctions, more than another instance of programming practice. Nonetheless, as the combinatorial complexity is huge, we use cuts to add controls over the derivation. Indeed, it is less powerful theoretically, yet it is still powerful enought to cover examples of linguistic interests in point. The reason is this implementation corresponds to succeed once in the execution of Prolog program. And those linguistic examples indeed need one instance rule application only for each step. That is, there is no need to apply two rules on the same terms. Furthermore, it is possible to relax those controls to achieve more completeness as needed. The issue in point is complexity, rather than decidability or correctness. And what is been excluded is redundant derivations, though correct ones.
- Lexicalism of Categorial Grammars and Logic Programming Environment. Categorial grammars are described as Lexicalism in linguistic, as it pushes all phrase structure relations and properties into

lexicons. This is also why the Lambek Calculus can be a universal Grammars. From a programming point of view, it allow a high degree of modularity and parameterization of program codes. For instance, codes and design of the dictionary can be fully factored out from codes of the universal engine. While this is a general principle of most programming environments, Logic Programming environments as Prolog fit even better. The term structures, use of logical variables and unification mechanism match the independence of the engine and lexicon dictionary, as they are fully separated as different sets of clauses. Furthermore, it allows incremental design and coding, easy modifications and reuse of codes, as witnessed from the similarity and changes of three versions of specifications. Also such modularity suggests new variants, out of the perspective of Logic Programming. For instance, unification is factored as a predicate eq/2, which suggests extensions into any useful equational theories, or in general, any Prolog definable theories.

- Linguistic version of the Principle of Compositionality, namely, each expression is determined by the constituent parts and the mode of combination, fits the counterpart of formal semantics of programming language. Especially it shows the pragmatics of the principle for a computational implementation. Herbrand term models, as semantics of logic programs, further blur the distinction of syntax and semantics, and consequently distinguish logic paradigms by such close compatibility. In particular, the part of modes of combination allows a modular replacement of set of combination rules, without affecting other parts of the codes. This can be seen from the case analysis of M and G rules and their ramifications for as modularized in implementations.

- The notion of proof as type, or formula as type, known as Curry-Howard isomorphism, also contributes to the modularity of the universal engine. In other words, one derivation step corresponds to a step of proof and also a step of processing the proof, regardless if it is meaning composition or parse tree construction.

- Logic programming environment, in particular, logical variables and unification, facilitates the implementation, as explained above. Furthermore, it provides an easy extension with feature theory into the Categorial Grammars, withnessed the trend of merges with phrase structure based approaches. However, it is not without potential disparity, for instance, the universal scoping and untypedness of logical variables is different from the quantified variables in the semantic aspect of Categorial Grammars. It is solved in this instance of implementation, by careful implementation of improper substitutions. Care is needed for coping with other quantification and binding phenomena in other linguistic discourses.

In brief, the implementation is not only a computational version of the Lambek Calculus. More importantly, it demonstrates a close relationship between this linguistic paradigm and computer science. Further interdisciplinary interactions definitely would benefit both, beyond a computation version of a linguistic performance theory, or even beyond the performance and competence distinction itself.

In [Bu 88], the classical version, so-called Ajdukiewicz–Bar-Hillel Categorial Grammar, a forward directional version with Function Application as only rule of combination, is proved to be of same expressive power as context free language. As a corollary, this implementation, as more general than the classical version, is at least as powerful as context free languages. In [PG 82], phrase structure languages are convenient and comprehensive for describing and explaining phenomena of natural languages Furthermore, traditional arguments and examples for non-context freeness of natural languages are refuted as disguised context freeness. As a result, context freeness is no longer thought as an inadequacy for the description of weak generative capacity of natural languages per se, at least within this GPSG paradigm. Moreover, phrase structure is an advantageous framework of the study and explanation of other linguistic phenomena, like acquisition, meaning and processing, as advocated in [Po 88, PG 82]. Revival of Categorial Grammars as methodological alternative benefits from the same line of arguments, more than its generative capacity. Furthermore, the framework and extensions are even argued to be more naturally in accords with human processing, in particular, on the aspects of non-constituent and incremental understanding [St 88]. And as we will show, some variants of our system, using **Sets** as aggregate domain, would be able to cover cases in [St 88], Multiple

Dependence. Other aggregate domains certainly would be able to incorporate more features not available in the Lambek Calculus. This is not yet the case in current version and its implementation.

6 Other Possible Variants

6.1 Variant A

As shown in the example where unrelated type terms, $a; b$, can be *paired* into $X/((X/a)/b)$. In fact, this can be generalized into unrelated functor-argument terms, so, $a_1; \cdots; a_n$, are paired into $X/((((X/a_1)/a_2)/\cdots/a_n)$. In this case, $X/(X/\cdots)$ functions as an aggregation constructor. There are many instances of domains with aggregation operators of different algebraic properties, like,**Sets, Bags, String, Lists, Numbers** etc. So this extension actually suggests a powerful scheme for many an application domains. Along with the property of Local Permutability, and the fact that most of aggregation domains are also permutation closed for all aggregated elements, we propose an alternative restricted set of combination rules, nondirectional and non-associative still, other than the commonly used set of $\{FA, FC, M\}$:

Definitions :		
$X/aggr(a_1, \cdots a_n)$	$\equiv$	$(((X/a_1)/a_2)/\cdots a_n)$
Function Application I :		
$F/A; A$	$\Rightarrow$	F
Function Application II :		
$F/aggr(a_1, \cdots, a_n); (a_1; \cdots; a_n)$	$\Rightarrow$	F
Function Composition :		
$F/A; A/B$	$\Rightarrow$	F/B

F, A, B are all non-aggr basic type terms, where slight relaxation to compound type may be considered as an another variant. Basically, we replace all (n)-ary functor-argument structures as a unary function with a n-aggr argument. Furthermore, we prevent it from being Functionally Composed. For other cases, like, F, A, B, it is the same as conventional functor-argument terms, which in strict sense are restricted to be unary with possible higher order arguments.

While G and M rules are explicitly eliminated, we note they are actually implicitly encoded in Functional Composition, as G plus Function Application, and aggregate arguments, as M plus some G steps. Intuitively, this set of combination rule would enforce constituents as arguments and phrase structure as functor-argument relationship, as dictated by the aggregate argument and restricted Functional Composition, while allowing for some unbounded dependency enabled by Function Composition.

To illustrate the point, let's see an example of derivation :

Example. 4

$$
\begin{array}{lcccccc}
 & \underline{a/b;} & \underline{b/c;} & d; & c/aggr(d,e); & \underline{e/f;} & f \\
\Rightarrow_{FC} & & a/c; & & & & \\
\Rightarrow_{FA.I} & & & & \underline{\qquad\qquad\qquad} & e; & \\
\Rightarrow_{FA.II} & & \underline{\qquad} & & c; & & \\
\Rightarrow_{FA.I} & & a & & & &
\end{array}
$$

Note how the constituent is kept and eager functional composition previously available between $(c/d)/e$ *and* e/f *is blocked out.* □

6.2 Variant B

To impose further controls over the possible derivations, to allow a little more freedom of functional composition in last example, and to prevent the non-confluent derivations mostly due to undirectedness, the following forward directional variant would be of more practical interests :

Definitions :		
$X/aggr(a_1, \cdots a_n)$	$\equiv$	$(((X/a_1)/a_2)/\cdots a_n)$
Identity :		
$F/aggr()$	$\Rightarrow$	F
Function Application I :		
$F/A; A$	$\Rightarrow$	F
Function Application II :		
$F/aggr(a_1, \cdots, a_i, \cdots, a_n); (a_i)$	$\Rightarrow$	$F/aggr(a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n)$
Function Composition I :		
$F/A; A/B$	$\Rightarrow$	F/B
Function Composition II :		
$F/(aggr \cdots, a_i, \cdots); a_i/B$	$\Rightarrow$	$F/aggr(\cdots, B, \cdots)$

To simplify the case, we keep the constraint that F, A, B are all non-aggr type terms only.

Example. 5

	$a/b;$	$b/c;$	$c/aggr(d,e);$	$e/f;$	$d/aggr(h);$	$h;$	f
$\Rightarrow_{FC.I}$		$a/c;$					
$\Rightarrow_{FC.II}$			$c/aggr(d,f);$				
$\Rightarrow_{FA.II}$					$d/aggr();$		
$\Rightarrow_{Id.}$					$d;$		
$\Rightarrow_{FA.II}$			$c/aggr(f);$				
$\Rightarrow_{FA.II}$			$c/aggr();$				
$\Rightarrow_{Id.}$			$c;$				
$\Rightarrow_{FA.I}$		a					

Note how only adjacent levels of constituent is collapsed and how eager functional composition is restored in better controlled style. □

This version, though very much deviated from standard linguistic variants of Lambek Calculus, is most interested and promising for the study of exchange and pattern processing in the pursuit of both authors.

6.3 Remarks

Remark 1.

The diversity of variants as illustrated above is majorly contributed by the local permutability. It is the richest and most comprehensive, in the sense of combinatorial complexity of factorial explosion of allowed patterns from finite objects. It is indeed this properties which underlies those diversity and powerful expressiveness. In addition, the directedness and controlled use of Functional Composition are another source of introducing powerful variations. On the other hand, the complexity need be harnessed to fit complexity requirement and special structural simplicity of specific applications. Of course, this is more an advantage. For instance, in the application to set aggregations, we only need consider normalized sets, namely, eliminating the occurrence and ordering in permutations. And a notion of failure to be joinable can be characterized.

To be precise :

$$
\begin{array}{lll}
F/set(X); x_i & \Rightarrow & F/set(X - x_i) \\
F/set() & \Rightarrow & F \\
\text{Or} & & \\
F/set(X); x_i & \Rightarrow & F \\
 & & \text{iff } x_i \in X \\
\text{failure} & \stackrel{\text{def}}{=} & x_i \notin X. \\
 & & \text{in } (F/X; x_i) \\
 & \text{or} & \\
 & \stackrel{\text{def}}{=} & Y \not\subseteq X \\
 & & \text{in } (F/X; Y)
\end{array}
$$

Those different clauses correspond to different modes of acceptance in pattern processing, or equality theory in general. And notion of failure would allow us to model backtracking or deadlock, as complementary to acceptance.

Remark 2.

It is important to note that, by using the set aggregate in the second variant, we are able to encompass the combinatory approach in [St 88], where Functional Substitution, a non Lambek Calculus feature, is used to deal with the Multiple Dependency. We show how this can be done :

$$
\begin{array}{lll}
\text{Functional Substitution :} & & \\
(X/Y)/Z; Y/Z & \Rightarrow_{FS} & (X/Z) \\
\text{Set Aggregate :} & & \\
X/set(Y, Z); Y/Z & \Rightarrow_{FC.II} & X/set(Z, Z) \\
 & \Rightarrow_{Set} & X/set(Z)
\end{array}
$$

It is even clear from this correspondence, that the nature of multiple dependence is to identify the two referenced arguments as a single expression.

This coverage also demonstrates the potential bearing of linguistic significance, even it is pure out of general algebraic or computational perspective.

Remark 3.

In effect, other than controls over local permutability, there is another line of generalization, which is even power. Notice that, in all reduction rule sets proposed, there is always a notion of unification involved, for instance, $F/X; X$, between functor and arguments. This could be in general equational theories, or arbitrary theories, where decidability is desired, and perhaps reasonable complexity, since this is supposedly a primitive mechanism in the system. This, along with the controlled permutability, can incorporate in a rich set of algebraic properties. This is very useful for search of canonically represented value arguments. In terms of implementation, this demonstrates another aspect of benefit of modularity by logic programming and formal linguistic principles.

7 Conclusion and Further Researches

In this article, we have reported a computation oriented study of a formalism from linguistics. The Categorial version of Simple Lambek Calculus is examined and its computational properties are established. Major properties are established and used later for comparisons, implementations and suggestions of extensions.

Among others, the property of local permutability in this categorial version plays the key role, as it embodies the full combinatorial versatility of factorial varieties. Meanwhile options are left open for controls of delicate refinement according to specific applications and needs. Through this examination, not only are we able to draw comparisons with other linguistic paradigms, but also able to exploit the observations in a computationally precise style. The experience of the implementations and the insights gained show the benefit of interdisciplinary interplay, by a correspondence of notions of respective fields of studies. And it demonstrates an instance of how knowledge and ideas in one field can be transformed into the other, and vice versa.

The analysis and implementation altogether helps suggest useful and powerful variants of linguistic formalism, beyond original linguistically motivated Lambek Calculus and Categorial Grammars. Those variants and potentially many others, not only bear interests in non-linguistic applications, but also exhibits how some linguistic proposal can be covered by such extensions.

While the linguistic aspect is less emphasized or even neglected, it is a fact that linguistic intuitions help shed light on the symbol-oriented perspective, along the line of computational, deductive and algebraic aspects of the system and its applications.

As for further research, drawing from those suggested variants, focuses and directions of efforts would be to exploit specific aggregate domains of interest, as [St 92, SVP 92]. Also the research here forms a relevant foundation for the study of patterns and stream processing, which is currently under intensive study by the authors. As it is proposed, the algebraic properties of aggregate domains and their defining characteristics can be formalized and translated into rules compatible with or refining the set of core rules. As far as the implementation is concerned, it would be fruitful to implement those specialized variants. As long as the aggregate domain bear computational representation, it is very likely adaptable to the framework. As this study suggests, Logic Programming environment is believed to be very helpful for implementations of those possible extensions. Furthermore, we would like to see how those variants related to studies in natural languages or to the aggregate domain per se, too.

Acknowledgement

References

[Ben 83] Johan van Benthem, *The Semantics of Variety in Categorial Grammar*, in *Categorial Grammars*, Buzskowski et al. (eds.), 1986.

[Ben 84] Johan van Benthem, *The Logics of Semantics*, in *Varieties of Formal Semantics*, F. Landman and F. Vletman (eds.), 1984.

[Ben 88] Johan van Benthem, *The Lambek Calculus*, in *Categorial Grammars and Natural Language Structures*, Richard T. Oehrle, Emmon Bach and Deirdre Wheeler (eds.), D.Reidel Publishing Company, 1988.

[Bu 78] Wojciech Buszkowski, *Undecidability of Some Logical Extensions of Ajdukiewicz-Lambek Calculus*, Studia Logica, 1978, 37(1), page 59-64.

[Bu 88] Wojciech Buszkowski, *Generative Power of Categorial Grammars*, in *Categorial Grammars and Natural Language Structures*, Richard T. Oehrle, Emmon Bach and Deirdre Wheeler (eds.), D.Reidel Publishing Company, 1988.

[Co 78] A. Colmerauer, *Metamorphosis Grammars*, in *Natural Language Communication with Computers*, LNCS 63, 1978.

[Do 88] David Dowty, *Type Raising, Functional Composition, And Non-Constituent Conjunction*, in *Categorial Grammars and Natural Language Structures*, Richard T. Oehrle, Emmon Bach and Deirdre Wheeler (eds.), D.Reidel Publishing Company, 1988.

[La 58] Joachim Lambek., *The Mathematics of Sentence Structure*, American Mathematical Monthly, 1958,65(3), 154-170.

[La 61] Joachim Lambek, *On the Calculus of Syntactic Types*, in *Structure of Language and Its Mathematical Aspects*, Proc. of Symposia in Applied Mathematics, Vol. 12, 1961.

[La 88] Joachim Lambek, *Categorial and Categorical Grammars*, in *Categorial Grammars and Natural Language Structures*, Richard T. Oehrle, Emmon Bach and Deirdre Wheeler (eds.), D.Reidel Publishing Company, 1988.

[HU 79] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 1979.

[Mo 88] Michael Moortgat, *Categorial Investigations - Logical and Linguistic Aspects of The Lambek Calculus*, Foris Publication, 1988.

[OBW 88] Richard T. Oehrle, Emmon Bach and Deirdre Wheeler (eds.), *Categorial Grammars and Natural Language Structures*, D.Reidel Publishing Company, 1988.

[Po 88] Carl J. Pollard, *Categorial Grammar and Phrase Structure Grammar : An Excursion on the Syntax-Semantics Frontier*, in *Categorial Grammars and Natural Language Structures*, Richard T. Oehrle, Emmon Bach and Deirdre Wheeler (eds.), D.Reidel Publishing Company, 1988.

[PG 82] Geoffrey K. Pullum and Gerald Gazdar, *Natural Languages and Context-Free Languages*, Linguistics and Philosophy, Vol.4, No.4, 1982.

[St 88] Mark Steedman, *Combinators and Grammars*, in *Categorial Grammars and Natural Language Structures*, Richard T. Oehrle, Emmon Bach and Deirdre Wheeler (eds.), D.Reidel Publishing Company, 1988.

[St 92] D. Stott Parker, Shen-Tzay Huang and Xin Wang , *Exchange Calculus and Their Applications*, (working paper), CSD UCLA, 1992.

[SVP 92] D. Stott Parker, Eric Simon and Patrick Valduries, *SVP - A Model Capturing Sets, Streams and Parallelism*, Technical Report CSSD-920020, UCLA Computer Science Department, April 1992. To appear, *Proc. Intnl. Conf. on VLDB*, 1992.

A Appendix : Prolog Implementations

Note all implementation is done in Sicstus Prolog 2.0.

A.1 Pure Calculus

```
%  Version 1 :
%    Undirectional + FA + FC.
:- op(400, xfy, ':').

%  Main program.
lambek(Lexicon, Final) :-
        lca(Lexicon, Clist),
        calculus(Clist,Final).

calculus([X],[X]).
calculus([X,Y|Xs],Z) :-
        lc(X,Y,XY),!,
        calculus([XY|Xs],Z).
calculus([X,Y|Xs],[Z]) :-
        calculus([Y|Xs],[YXs]),
        lc(X,YXs,Z), !.
calculus([X,Y|Xs],Z) :-
        geachable(X,Y,XY),!,
        calculus([XY|Xs],Z).
calculus([X,Y|Xs],Z) :-
        mable(X,Y,XY),!,
        calculus([XY|Xs],Z).

%  Function Application
lc(F:X, Y, F)    :- eq(X,Y).
lc(X, F:Y, F)    :- eq(X,Y).

%  Functional Composition
lc(F:X, Y : Z, F:Z) :- eq(X,Y).
lc(Y:Z, F : X, F:Z) :- eq(X,Y).

%  Type Raising
%  a. Geach Rule
%      a:b  -> (a:c):(b:c)
%      Lower middle case.
%       W:R -> ((W:Z):(R:Z)).
geachable(X:(Y:Z), W:R, X:(R:Z)) :-
        eq(W,Y).
geachable(W:R, X:(Y:Z), X:(R:Z)) :-
        eq(W,Y).
%  Upper middle case.
geachable((Y:Z):X, R:W, (R:Z):X) :-
        eq(W,Y).
geachable(R:W, (Y:Z):X, (R:Z):X) :-
        eq(W,Y).

%  b. M rule.
%      a -> b/(b/a).
%      lower case : X/Y ; Y/(Y/W).
mable(X:Y,W,X:(Y:W)) :- \+ eq(W,Y).
mable(W,X:Y,X:(Y:W)) :- \+ eq(W,Y).
%      upper case : (X/W)/Y ; X/(X/W).
mable((X:U):Y,W,X:Y) :-
        \+ eq(W,Y),eq(U,W).
mable(W,(X:U):Y,X:Y) :-
        \+ eq(W,Y),eq(U,W).

%  Lexical Category Assignment.
lca([],[]).
lca([X|Xs], [Y|Ys]) :-
        dict(X,Y), lca(Xs,Ys).

%  dictionary :
%  for phrase :
%  a cake which i believe that she ate.
dict(a, np:n).
dict(cake, n:r).
dict(which, r:(s:np)).
dict(i,s:fvp).
dict(believe, fvp:s1).
dict(that, s1:s).
dict(she,s:fvp).
dict(ate, fvp:np).

%  for phrase :
%  i believe that she ate those cakes.
dict(those, np:n).
dict(cakes,n).

% for phrase : john walks.
dict(john, n).
dict(walks, t:n).

% for phrase : everyone loves someone.
dict(everyone,s:fvp).
dict(loves,fvp:np).
dict(someone,s:(s:np)).

%  Underlying equational theory for
%  defining matching. Currently, it's
%  just first order unification.
eq(X,X).
```

A.2 With λ-Terms as Meaning Component

```
%  Version 2 :
%    Undirectional + FA + FC + M + G,
%    together with Semantic Data Structure
%    along with the parsing processing.
%
:-  op(400, xfy, ':').
:-  op(500, xfy, 'of').

%  Main program.
lambek(Lexicon, Final) :-
        lca(Lexicon, Clist),
        calculus(Clist,Final).

calculus([X],[X]).
calculus([X,Y|Xs],Z) :-
        lc(X,Y,XY),!,
        calculus([XY|Xs],Z).
calculus([X,Y|Xs],[Z]) :-
        calculus([Y|Xs],[YXs]),
        lc(X,YXs,Z), !.
calculus([X,Y|Xs],Z) :-
        geachable(X,Y,XY),!,
        calculus([XY|Xs],Z).
calculus([X,Y|Xs],Z) :-
        mable(X,Y,XY),!,
        calculus([XY|Xs],Z).

%  Function Application
lc(V1 of F:X,V2 of Y, V3 of F) :-
        eq(X,Y), apply(V1,V2,V3).
lc(V1 of X, V2 of F:Y, V3 of F) :-
        eq(X,Y), apply(V2,V1,V3).

%  Functional Composition
lc(V1 of F:X, V2 of Y : Z, V3 of F:Z) :-
        eq(X,Y),compose(V1,V2,V3).
lc(V1 of Y:Z, V2 of F : X, V3 of F:Z) :-
        eq(X,Y),compose(V2,V1,V3).

%  Type Raising
%  a. Geach Rule
%     a:b -> (a:c):(b:c)
%     Lower middle case.
geachable(V1 of X:(Y:Z), V2 of W:R,
          V3 of X:(R:Z))
      :- eq(W,Y), simple_graise(V2,V1,V3).
geachable(V1 of W:R, V2 of X:(Y:Z),
          V3 of X:(R:Z))
      :- eq(W,Y),simple_graise(V1,V2,V3).

%  Upper middle case.
geachable(V1 of (Y:Z):X, V2 of R:W,
          V3 of (R:Z):X)
      :- eq(W,Y), graise(V2,V1,V3).
geachable(V1 of R:W, V2 of (Y:Z):X,
          V3 of (R:Z):X)
      :- eq(W,Y), graise(V1,V2,V3).

%  b. M rule.
%     a -> b/(b/a).
%     Lower case :   X/Y ; Y/(Y/W).
mable(V1 of X:Y, V2 of W, V3 of X:(Y:W)) :-
      \+ eq(W,Y),mraise(V2,V1,V3).
mable(V1 of W, V2 of X:Y, V3 of X:(Y:W)) :-
      \+ eq(W,Y),mraise(V1,V2,V3).
%  Upper case : (X/W)/Y ; (X/(X/W)).
mable(V1 of (X:U):Y, V2 of W, V3 of X : Y)
      :- \+ eq(W,Y), eq(U,W),
         mraise2(V2,V1,V3).
mable(V1 of W, V2 of (X:U):Y, V3 of X : Y)
      :- \+ eq(W,Y), eq(U,W),
         mraise2(V1,V2,V3).

%  Lexical Category Assignment.
lca([],[]).
lca([X|Xs], [Y|Ys]) :-
      dict(X,Y), lca(Xs,Ys).

%  dictionary :
%  for phrase : john walks.
dict(john, john of n).
dict(walks, fun(X, walks(X)) of t:n).

% for phrase : everyone loves someone.
dict(everyone,fun(X, all(X)) of s:fvp).
dict(loves,fun(X, love(X)) of fvp:np).
dict(someone,fun(F,F:somebody) of s:(s:np)).

%  Underlying equational theory for
%  defining matching. Currently, it's just
%  first order unification.
eq(X,X).

%  Supporting Lambda Calculus routines.
apply(fun(X,F),D,F)  :- eq(X,D).

compose(fun(X,F), fun(Y,G), fun(Y,F)) :-
       eq(X,G).

%  M rules : first raise W to fun(Z,Z:W),
%  and then compose with fun(Y,XY) using
%  the above rule compose/3.
```

```
mraise(W,fun(Y,X), fun(Z,X)) :-
       eq(Y,Z:W).
mraise2(W, fun(Y, fun(U,X)), fun(Y, X)) :-
       eq(U,W).

%  G rules :    This is tricky in terms of
%   parameter passing by lambda variables.
%   Improper substitution by the use of
%   logical variables is carefully coded to
%   deal with type varibles and renamed
%   substitution.
graise(fun(Y,X), fun(W,fun(Z,F)),
      fun(W,fun(Z,XF))) :- eq(Y,F),XF = X.

simple_graise(fun(R,W), fun(YZ,X),
      fun(R,fun(Z,XW))) :- eq(W,YZ), XW = X.
```

A.3 With Parse Tree

```
%  Version 3 :
%    Undirectional + FA + FC + M + G,
%    together with only parse structure
%    along with the parsing processing.
:-  op(400, xfy, ':').
:-  op(500, xfy, 'of').

%  Main program.
lambek(Lexicon, Final) :-
        lca(Lexicon, Clist),
        calculus(Clist,Final).

calculus([X],[X]).
calculus([X,Y|Xs],Z) :-
        lc(X,Y,XY),!,
        calculus([XY|Xs],Z).
calculus([X,Y|Xs],[Z]) :-
        calculus([Y|Xs],[YXs]),
        lc(X,YXs,Z), !.
calculus([X,Y|Xs],Z) :-
        geachable(X,Y,XY),!,
        calculus([XY|Xs],Z).
calculus([X,Y|Xs],Z) :-
        mable(X,Y,XY),!,
        calculus([XY|Xs],Z).

%  Function Application
lc(V1 of F:X,V2 of Y, V3 of F) :-
        eq(X,Y), apply(V1,V2,V3).
lc(V1 of X, V2 of F:Y, V3 of F) :-
        eq(X,Y), apply(V2,V1,V3).

%  Functional Composition
lc(V1 of F:X, V2 of Y : Z, V3 of F:Z) :-
        eq(X,Y),compose(V1,V2,V3).
lc(V1 of Y:Z, V2 of F : X, V3 of F:Z) :-
        eq(X,Y),compose(V2,V1,V3).

%  Type Raising
%  a. Geach Rule
%     a/b ->  (a/c)/(b/c).
%     Lower middle case.
geachable(V1 of X:(Y:Z), V2 of W:R,
          V3 of X:(R:Z))
       :- eq(W,Y), simple_graise(V2,V1,V3).
geachable(V1 of W:R, V2 of X:(Y:Z),
          V3 of X:(R:Z))
        :- eq(W,Y),simple_graise(V1,V2,V3).
%  Upper middle case.
geachable(V1 of (Y:Z):X, V2 of R:W,
          V3 of (R:Z):X)
       :- eq(W,Y), graise(V2,V1,V3).
geachable(V1 of R:W, V2 of (Y:Z):X,
         V3 of (R:Z):X)
        :- eq(W,Y), graise(V1,V2,V3).

%  b. M rule.
%     a ->  b/(b/a).
mable(V1 of X:Y, V2 of W, V3 of X:(Y:W)) :-
      \+ eq(W,Y),mraise(V2,V1,V3).
mable(V1 of W, V2 of X:Y, V3 of X:(Y:W)) :-
      \+ eq(W,Y),mraise(V1,V2,V3).
mable(V1 of (X:U):Y, V2 of W, V3 of X : Y)
           :- \+ eq(W,Y), eq(U,W),
              mraise2(V2,V1,V3).
mable(V1 of W, V2 of (X:U):Y, V3 of X:Y)
           :- \+ eq(W,Y), eq(U,W),
              mraise2(V1,V2,V3).

%  Lexical Category Assignment.
lca([],[]).
lca([X|Xs], [Y|Ys]) :-
       dict(X,Y), lca(Xs,Ys).

%  dictionary :
%  for phrase : john walks.
dict(john, john of n).
dict(walks, walks of t:n).

% for phrase : everyone loves someone.
dict(everyone,everyone of s:fvp).
dict(loves,loves of fvp:np).
dict(someone,someone of s:(s:np)).
```

```
%  Underlying equational theory for
%  defining matching. Currently, it's just
%  first order unification.
eq(X,X).

%  Parse Tree Construction Routines.
apply(F,D,beta(F,D)).

compose(F,G, comp(F,G)).

%  M rules :  Two Cases.
mraise(W,X, msubcomp(W,X)).

mraise2(W,X, msubcomp(W,X)).

%  G rules :  Two Cases.
graise(YX, WZF, g1subcomp(YX,WZF)).

simple_graise(RW, ZYX, g2subcomp(ZYX,RW)).
```

A.4 Sample Run Sessions

A.4.1 For Version 1 : Pure Calculus

```
[maui,2/1] prolog
SICStus 0.7 #0.1: Fri Oct 25 09:12:35 PDT 1991
{consulting /u/gs4/sthuang/prolog.ini...}
{/u/gs4/sthuang/prolog.ini consulted, 20 msec 356 bytes}
| ?- [lc1].
{consulting /amnt/hilo/gs4/sthuang/thesis/paper_TR/Examples/lc1.pl...}
{/amnt/hilo/gs4/sthuang/thesis/paper_TR/Examples/lc1.pl consulted,
 230 msec 9078 bytes}

yes
| ?- lambek[([everyone, loves, someone], X).

X = [s] ? ;

no
| ?- lambek([everyone,someone,loves],X).

X = [s] ? ;

no
| ?- lambek([loves,someone,everyone],X).

X = [s] ? ;

no
| ?- lambek([loves,everyone,someone]),X).

X = [s] ? ;

no
| ?- lambek([someone,everyone,loves],X).

X = [s] ? ;

no
| ?- lambek([someone,loves,everyone],X).

X = [s] ? ;

no
| ?- lambek([a,cake,which,i,believe,that,she,ate],X).

X = [np] ? ;

X = [n:((((((s:np):sa):s:fvp):fvp:s1):s1:s):s:fvp):fvp:np] ? ;

no
| ?- ^D
```

```
{ End of SICStus execution, user time 0.410 }
```

A.4.2 For Version 2 : With λ-Term as Meaning Component

```
[maui,2/2] prolog
SICStus 0.7 #0.1: Fri Oct 25 09:12:35 PDT 1991
{consulting /u/gs4/sthuang/prolog.ini...}
{/u/gs4/sthuang/prolog.ini consulted, 20 msec 356 bytes}
| ?- [lc2].
{consulting /amnt/hilo/gs4/sthuang/thesis/paper_TR/Examples/lc2.pl...}
{/amnt/hilo/gs4/sthuang/thesis/paper_TR/Examples/lc2.pl consulted,
 350 msec 10798 bytes}

yes
| ?- lambek([john,walks],X).

X = [walks(john)of t] ? ;

no
| ?- lambek([everyone,loves,someone],X).

X = [fun(_178,all(love(_178))):somebody of s] ? ;

no
| ?- lambek([everyone,someone,loves],X).

X = [fun(_256,all(fun(_206,love(_206))):somebody)of s] ? ;

no
| ?- lambek([loves,everyone,someone],X).

X = [fun(_154,all(love(_154))):somebody of s] ? ;

no
| ?- lambek([loves,someone,everyone],X).

X = [fun(_276,all(fun(_154,love(_154))):somebody)of s] ? ;

no
| ?- lambek([someone,loves,everyone],X).

X = [fun(_182,all(love(_182))):somebody of s] ? ;

no
| ?- lambek([someone,everyone,loves],X).

X = [fun(_206,all(love(_206))):somebody of s] ? ;

no
| ?- ^D
{ End of SICStus execution, user time 0.530 }
```

A.4.3 For Version 3 : With Parse Tree

```
[maui,2/3] prolog
SICStus 0.7 No.1: Fri Oct 25 09:12:35 PDT 1991
{consulting /u/gs4/sthuang/prolog.ini...}
{/u/gs4/sthuang/prolog.ini consulted, 20 msec 356 bytes}
| ?- [lc3].
{consulting /amnt/hilo/gs4/sthuang/thesis/paper_TR/Examples/lc3.pl...}
{/amnt/hilo/gs4/sthuang/thesis/paper_TR/Examples/lc3.pl consulted,
 280 msec 10288 bytes}

yes
| ?- lambek([john,walks],X).

X = [beta(walks,john)of t] ? ;

no
| ?- lambek([everyone,loves,someone],X).

X = [beta(someone,comp(everyone,loves))of s] ? ;

no
| ?- lambek([everyone,someone,loves],X).

X = [beta(g2subcomp(someone,everyone),loves)of s] ? ;

no
| ?- lambek([loves,someone,everyone],X).

X = [beta(g2subcomp(someone,everyone),loves)of s] ? ;

no
| ?- lambek([someone,loves,everyone],X).

X = [beta(someone,comp(everyone,loves))of s] ? ;

no
| ?- lambek([someone,everyone,loves],X).

X = [beta(someone,comp(everyone,loves))of s] ? ;

no
| ?- lambek([loves,everyone,someone],X).

X = [beta(someone,comp(everyone,loves))of s] ? ;

no
| ?- lambek([john,walks],X).

X = [beta(walks,john)of t] ? ;

no
```

```
| ?- lambek([walks,john],X).

X = [beta(walks,john)of t] ? ;

no
| ?- ^D
{ End of SICStus execution, user time 0.600 }
```