Learning Rigid Lambek Grammars and Minimalist Grammars from Structured Sentences[⋆]

Roberto Bonato[1] and Christian Retoré[2]

[1] IRISA-INRIA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
rbonato@irisa.fr
[2] IRIN, Université de Nantes, BP 92208, 44322 Nantes Cedex 03, France
retore@irisa.fr

Abstract. We present an extension of Buszkowski's learning algorithm for categorial grammars to rigid Lambek grammars and then for minimalist categorial grammars. The Kanazawa proof of the convergence in the Gold sense is simplified and extended to these new algorithms. We thus show that this technique based on principal type algorithm and type unification is quite general and applies to learning issues for different type logical grammars, which are larger, linguistically more accurate and closer to semantics.

1 Presentation

Learning lexicalized grammar This papers deals with automated syntax learning, also known as grammatical inference. The grammars we consider are lexicalized: their rules are universal, do not depend on the language, hence a grammar is completely defined by a map called the lexicon which associates to each word a finite set of objects, here types, which define the word syntactic behavior. In such grammar formalisms, acquiring a grammar consists in finding which types should be associated with a word in order that the lexicon generates the examples (computational linguistics does not usually consider counter examples). It should be observed that grammar formalisms that allow for a learning algorithm are rare.

Gold model There are various models for grammatical inference. Here we follow the model proposed by Gold [8]: *identification in the limit from positive examples only*. According to this model, each time the learner meets a new example, she generalizes her current grammar hypothesis to a grammar in the class which generates all the examples met so far. The learning process is said to be convergent if, whenever the set of examples is an enumeration of a language generated by a grammar in the class, the learner finds the correct grammar after a finite number of examples.

[⋆] This work is part of the INRIA *Action de Recherche Coopérative* GRACQ, Categorial Grammar Acquisition

Psycholinguistic aspects If one thinks, but this can be discussed, that a grammatical learning algorithm should roughly follow what is known of human language acquisition, this model is rather accurate: indeed, as claimed by Gleitman, Lieberman or Pinker in [7], it seems that only positive examples are used in the process of children's first language acquisition. Another fact that we do take into account is that learning proceed on structures, and it is known that children use structures to learn (they are provided by intonation, semantic contents etc.).

Nevertheless our kind of algorithm relying on generalization by unification departs from the actual process of learning: children do not go through an increasing sequence of languages till they reach the correct one, they rather go through languages which hardly intersect the language to be learnt.

Tools for automated grammar construction We are presently implementing such algorithms, using Objective CaML with a Tcl/Tk interface, and proceeding with XML files for representing structured data — directed acyclic graphs which embed all the various structures we are using. The learning algorithms are not too difficult to implement, and rather efficient.

A positive point is that we are able to learn classes of languages relevant for natural language syntax, which as far as we know, have not yet been addressed with respect to learning issues. The algorithms are quite fast. Nevertheless this is due to the fact that we learn from structures, but from structures which are too rich to be available from corpora, even tagged corpora. However it is necessary to learn from structures; when learning from plain strings of words nothing would prevent the grammar learnt from generating sentences like *Adam ate an apple* with the structure *(Adam (ate an)) apple*, which is not what we are looking for.

Another positive point is that we deal with lexicalized grammar. The grammatical words, which are the ones associated with the most sophisticated syntactic behavior are in a finite, fixed number: pronouns, prepositions and such are not invented by speakers, hence in a real application they should be already present in the lexicon, and the words which remain to be learnt are always simple words like nouns, adjectives, verbs. This advantage is minimized by the fact these grammars are not robust, and consequently have a small empirical coverage.

Previous work This work mainly relies on the first learning algorithm for categorial grammars of Buszkowski and Penn [4] which was shown to be convergent in the Gold sense by Kanazawa [10] — we use the simplified version due to the first author [2].

In order to learn a class of lexicalized grammars with a hope to be convergent, one has to bound the maximal number of types that the lexicon associates to a given word: otherwise each time a new word is seen the algorithm could provide it with a new type perfectly ad hoc for this particular sentence, and nothing would be learnt. The base case obviously is when the lexicon maps each word to one type: such grammars are called *rigid grammars* — and this limits their generative capacity.

The pioneering paper [4] provides an algorithm called RG for learning efficiently rigid AB grammars from structures — where AB grammars is a class of categorial grammars which only contains residuation rules. RG relies on a typing algorithm (an easy adaptation of the principal type algorithm for simply typed λ-calculus) and unification (with was shown to be a general tool for grammatical inference by Nicolas [16]).

Kanazawa [10] proved the convergence of RG, and extended the result in two directions that can be combined. One direction was to learn k-valued AB grammars by testing all the possibilities of unifying a type with one of the k types, and the other was to learn from strings by trying all possible parse structures on a given string. The convergence of these compatible extensions was shown, but the complexity of the learning algorithms becomes hardly tractable.

Aims of the paper Compared to Kanazawa, we extend the RG algorithm in an orthogonal direction: we moved to Lambek categorial grammars which are obtained by adding two rules to the two rules of AB grammars, and, in the same style we moved to the richer deductive system and grammar defined with Lecomte [13] which enables a representation of Stabler's minimalist grammars [18] a formalization of Chomsky's minimalist program [5]. In both cases we wish to learn more realistic classes of languages without losing the efficiency of the original algorithm. Furthermore the methods of Kanazawa for extending our results to k valued grammars or for learning from strings probably apply here as well.

Little is known regarding the class of rigid Lambek languages: all its languages are context-free and some are not regular, but it does not include all regular languages — learnable classes are always transversal to the Chomsky hierarchy [8]. We do think, from the examples we tried, e.g. example 1 in the paper, that rigid Lambek grammars are more expressive than rigid AB grammars — although when unrestricted both AB grammars and Lambek grammars exactly describe context free languages. Regarding minimalist grammars, we know that rigid minimalist grammars do contain non context free languages like the language $a^nb^nc^nd^ne^n$ (Stabler, private communication). Furthermore minimalist grammars take into account sophisticated syntactic constructions, and enjoy polynomial parsing.

As it well-known, learning is a cornerstone in Chomskyan linguistics. Language acquisition is viewed as parameter setting in the universal grammar — hence few examples are needed to acquire the syntax. These parameters, like feature strength, are explicit in minimalist grammars, and this could yield a more efficient learning strategy.

Another motivation is semantics. If the lexicon includes simply-typed λ-terms depicting word semantics, there is a straightforward mapping from syntactic analyses to semantics [15]. As real learning makes use of some semantic information, the possibility to learn Lambek grammars should improve syntax learning from sentences enriched with semantic information as suggested for instance in [19]. Although the syntax/semantics interface is less simple for minimalist grammars [13], it is also computable, and this interface could be exploited as well.

Regarding learning theory we think that the RG algorithm and the simple proof of convergence of [10, 2] could lead to a general result on learnability which can be applied to most type logical grammars.

2 Categorial grammars: BCG, LCG, MCG

In categorial grammars, the objects that the lexicon associates to words to rule their syntactic behavior are *types* (also called *categories* or *formulae*):

$$Tp \quad ::= \quad Pr \quad | \quad Tp/Tp \quad | \quad Tp\backslash Tp$$

The set Pr is the set of base categories which must contain a distinguished category s (sentence) and usually contains categories np (noun phrase) and n (noun), and possibly vp (verb phrase), pp (prepositional phrase). Stated informally, an expression e is of type B/A (resp. $A\backslash B$) when it needs to be followed (resp. preceded) by an expression e' of type A to obtain an expression of type B.

Given $a \in \Sigma$ (the set of words) and $A \in Tp$, we write $G : a \mapsto A$ (*G assigns A to a*), whenever A is one of the types associated to the word a. If for all $a \in \Sigma$ G assigns at most one type to a, G is said to be **rigid**.

Given a grammar/lexicon G, a sequence[1] of words $w_1 \dots w_n \in \Sigma^+$ is said to be of type $T \in Tp$ whenever there exists for every i in $[1, n]$ a type $T_i \in Tp$ such that $T_1 \cdots T_n \vdash T$, where $\vdash$ is the following relation between finite sequences of types and types:

Definition 1. *The binary relation $\vdash$ between Tp^+ and Tp is the smallest relation such that, for all $A, B \in Tp$ and all $\Gamma, \Delta \in Tp^+$:*

$[ID]$ $A \vdash A$
$[/E]$ *if $\Gamma \vdash A$ and $\Delta \vdash A\backslash B$, then $\Gamma, \Delta \vdash B$ (Forward application or modus ponens);*
$[\backslash E]$ *if $\Gamma \vdash B/A$ and $\Delta \vdash A$, then $\Gamma, \Delta \vdash B$ (Backward application, modus ponens);*
$[\backslash I]$ *if $A, \Gamma \vdash B$ then $\Gamma \vdash A\backslash B$ ($\backslash$ hypothetical reasoning, introduction);*
$[/I]$ *if $\Gamma, A \vdash B$ then $\Gamma \vdash B/A$ ($/$ hypothetical reasoning, introduction);*

As the $\vdash$ symbol suggests it is a deduction relation which can be depicted by natural deduction trees. [2]

$$A\ [ID] \qquad \dfrac{\begin{matrix}\vdots & \vdots\\ A/B & B\end{matrix}}{A}[/E] \qquad \dfrac{\begin{matrix}\vdots & \vdots\\ B & B\backslash A\end{matrix}}{A}[\backslash E] \qquad \dfrac{\begin{matrix}[B]\\ \vdots\\ A\end{matrix}}{A/B}[/I] \qquad \dfrac{\begin{matrix}[B]\\ \vdots\\ A\end{matrix}}{B\backslash A}[\backslash I]$$

Note: in [/I] and [\I] rules the cancelled or discharged hypothesis is always the rightmost and the leftmost uncanceled hypothesis, respectively, and there must remain at least one other uncanceled hypothesis.

The grammars defined using this deductive system are Lambek grammars (**LCG**) from [12] while the ones which only use $[\backslash R]$ and $[/E]$ are called AB (Ajdukiewicz Bar-Hillel) grammars or basic categorial grammars (**BCG**) from [1] (see [3] for a survey). The learning algorithms defined so far [4, 10] only handles BCGs.

[1] As usual, given a set U, U^* stands for the finite sequences of elements in U and U^+ for the finite non empty sequences of elements in U

[2] The only difference with usual natural deduction is that the order of premises matters, and that exactly one hypothesis is cancelled in an introduction rule. This can be observed from the fact that, in the previous definition there are no rule *if $\Gamma, A, B, \Delta \vdash C$ then $\Gamma, B, A, \Delta \vdash C$* and no rule *if $\Gamma, A, B, \Delta \vdash C$ then $\Gamma, A, B, \Delta \vdash C$*.

Example 1.

If one consider the rigid grammar/lexicon besides, then the sentences *Guarda passare il treno* (*(s)he is looking the train passing by*) and *Cosa guarda passare* (*What is (s)he looking passing by?*) belong to the generated language of the LCG. With BCG, only the first one is generated.

$cosa$	$\mapsto S/(S/np)$
$guarda$	$\mapsto S/np$
$passare$	$\mapsto vp/np$
il	$\mapsto np/n$
$treno$	$\mapsto n$

The languages generated by AB categorial grammars exactly are the context-free languages, and the languages generated by Lambek categorial grammars also are exactly context-free languages. Nevertheless the LCG generate much richer structure languages than BCG, see below.

Minimalist categorial grammars (MCG) In [13] the minimalist grammars of [18] have defined similarly, by using a lexicon which ranges over more sophisticated types. There is no room to present there intuitively, but formally they can be defined a mere variation of categorial grammars. The set of primitive categories is larger: it includes primitive categories but also movement features, like case, wh etc. which can be strong or weak[3] Types are extended with a commutative and associative product:

$$Tp_m \quad ::= \quad Pr \quad | \quad Tp_m/Tp_m \quad | \quad Tp_m\backslash Tp_m \quad | \quad Tp_m \otimes Tp_m$$

Instead of sequences of types, contexts are sets of types and the deductive system is replaced with the following one, where every formula is labeled by a sequence of words:[4]

$[ID]$ $x : A \vdash x : A$
$[AX]$ $\vdash w : A$ whenever $w \mapsto A$ is in the lexicon.
$[/E]$ if $\Gamma \vdash x : A$ and $\Delta \vdash y : A\backslash B$, then $\{\Gamma, \Delta\} \vdash xy : B$
$[\backslash E]$ if $\Gamma \vdash y : B/A$ and $\Delta \vdash x : A$, then $\{\Gamma, \Delta\} \vdash yx : B$
$[\otimes E]$ if $\Delta \vdash u : A \otimes B$ and $\Gamma[\{x : A, y : B\}] \vdash t(x, y) : C$ then $\Gamma, \Delta \vdash t(x := u_1, y := u_2) : C$ with, depending on A either $u_1 = u_2 = u$ or $u_1 = (u)$ and $u_2 = u$.

The labels can be computed from the proof. The way to read them is a bit puzzling, but agrees with the copy theory of minimalism. Every word sequence w has a phonological part $/w/$ and a semantic part (w) and when they are superimposed, this is simply denoted by w. The reason for this double sequence (semantic and syntactic) is to obtain correct semantic representation in a parameterized view of language variation.

To read such a sentence, one should only keep the phonological part and forget items which are met several times, the repetition being traces: for instance *Petrus Mariam amat Mariam* is phonologically read as */Petrus/ /Mariam/ /amat/* and semantically read as *(Petrus) (Mariam) (amat)* while *Peter (Mary) loves Mary* is phonologically read as

[3] For instance in VSO languages the accusative case provided by the verb is weak, while it strong in SOV languages.

[4] In fact if one wants a real logical system, then it is more complex. It involves structured contexts with a commutative comma and a non commutative comma, and the possibility to replace non commutative commas by commutative ones. However a large part of this formalism can be handled by the little system we give here.

/Peter/ /loves/ /Mary/ and semantically read as *(Peter) (Mary) (loves)*. If we consider the assignments $Mary \mapsto d \otimes k$ and $loves \mapsto K\backslash vp/d$ we obtain that *(Mary) loves Mary* is a vp, but if case k was strong we would obtain that *Mary loves Mary* is a vp as in SOV languages.

$$\dfrac{\vdash Mary : d \otimes k \qquad \dfrac{\dfrac{y : K \vdash y : K \qquad \dfrac{\vdash loves : K\backslash vp/d \quad x : d \vdash x : d}{x : d \vdash loves\ x : k \backslash vp}\,[/E]}{y : K; x : d \vdash y\ loves\ x : vp}\,[\backslash E]}{y : K, x : d \vdash y\ loves\ x : vp}}{\vdash (Mary)\ loves\ Mary : vp}\,[\otimes E]$$

The grammar is said to generate a sentence $w_1 \ldots w_n$ whenever the lexicon contains assignments $w_i \mapsto T_i$ such that $T_1, \ldots, T_n \vdash /w_1/ \ldots /w_n/ : s$. These grammars have been shown to generate exactly the same languages as Multiple Context-Free grammars, which go beyond context free grammars and even TAG languages. [9, 14].

2.1 Proofs as Parse Structures

As said above, parse structures are essential to a grammar system. As Tiede [20], we define them as *normal* proof trees that are proof trees in which an introduction rule yielding $A\backslash B$ or B/A is never followed by an elimination rule whose argument is A. Every proof tree can be normalized in linear time to a normal one, and there are finitely many normal proof tree with the same hypotheses and conclusion.

Thus the structure underlying a sentence $a_1 \cdots a_n \in \Sigma^+$ generated by a Lambek grammar G is one of the finitely many normal proof trees of the deduction $A_1, \ldots, A_n \vdash s$, with $G : a_i \mapsto A_i$. Figure 1 displays the proof tree of the sentence 'he likes her' in a grammar G such that G : he $\mapsto s/(np\backslash s)$, him $\mapsto (s/np)\backslash s$, likes $\mapsto (np\backslash s)/np$.

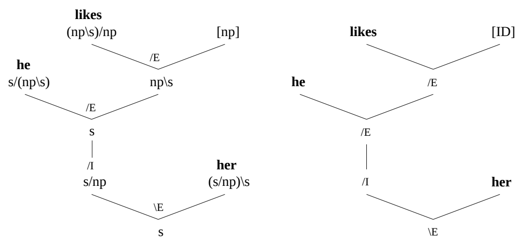

Fig. 1. Proof tree for a sentence and the corresponding proof-tree structure.

Given a Lambek grammar G, a *proof-tree structure* over its alphabet Σ is a unary-binary branching tree whose leaf nodes are labeled by either $[ID]$ ("discharged" leaf nodes) or symbols of Σ and whose internal nodes are labeled by either $[\backslash E]$, $[/E]$, $[\backslash I]$, or $[/I]$. Thus, a proof tree structure of a sentence is obtained from a proof tree by removing the types which label the nodes. We write Σ^P for the *structure language* that is the set of proof-tree structures over Σ, and $\mathrm{PL}(G)$ for the *(proof-tree) structure language* of G that is the set of structures generated by G. In order to distinguish $\mathrm{L}(G)$, the language of G, from $\mathrm{PL}(G)$, its structure language, the former is called the *string language* of G. The *yield* of a proof-tree structure T is the string of symbols $a_1, \ldots, a_n \in \Sigma^+$ labelling the undischarged leaf nodes of T, from left to right in this order. The yield of T is denoted $yield(T)$. Note that $\mathrm{L}(G) = \{yield(T) \mid T \in \mathrm{PL}(G)\}$.

By taking a proof tree as the structure of a sentences generated by Lambek grammars, Tiede in [20] proved some important results about their strong generative capacity. Firstly, a proof in the Lambek calculus is really a tree — as opposed to a proof in intuitionistic logic. Indeed, as he observed there is no need to specify which introduction rule $[I/]$ or $[\backslash I]$ did cancel a leaf in the course of a derivation: this information can be reconstructed from the leaves to the root, since it is always the left the leftmost or right most uncanceled hypothesis. Secondly, as opposed to a previous notion of parse structures for Lambek grammars [3], not every bracketing is produced by the finitely many normal proof trees which parse a given sentence. Thirdly, even though Lambek grammars and BCGs both exactly generate context-free languages, the former can generate proper context-free tree languages while the latter, as context free grammars only generate regular tree languages.

Parse structures for minimalist categorial grammars For the same reason we take proof trees as parse structure for MCG. Observe that there as well subtrees correspond to constituents.

3 Learning Framework: Gold's Model from Structured Examples

This sections recalls the basic formal notions of the learning model we use, Gold's model of learning [8], also called *identification in the limit from positive data*. We are given a universe T (for us the set of parse structures) and a class Ω of finite descriptions of subsets of T (for us categorial grammars), with a naming function N which maps each element of Ω to the subset of T that it generates. The learning question is given a subset S of T (positive examples), to find an element $G \in \Omega$ such that $N(G) = S$. The learning function we are looking for is a partial function φ from finite sequences of elements in T (or in S, since it is a partial function) to Ω which converges in the sense of Gold: whenever $S = N(G)$ for some $G \in \Omega$, then for any enumeration $(e_i)_{i\in\mathbb{N}}$ of S, there exists an $p \in \mathbb{N}$ such that $\forall n > p$ one has $\phi(e_1, \ldots, e_n) = G'$ and $N(G') = N(G)$.

Stated informally, we can think of a learning function as a formal model of the cognitive process (successful or unsuccessful) by which a learner conjectures that a given finite set of positive samples of a language are generated by a certain grammar. The convergence simply means that after a finite number of examples, the hypothesis made by the learner is a grammar equivalent to the correct one.

In Gold's model, we will say that a class of grammars is *learnable* when for *each* language generated by its grammars there exists a learning function which converges

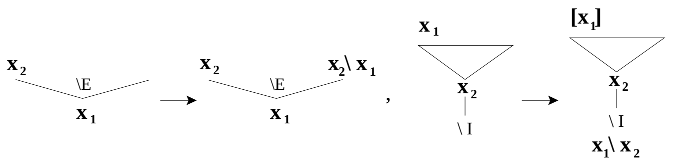

Fig. 2. Typing rules

to one correct underlying grammar; it is called *effectively learnable* if that function is computable.

4 An Algorithm for Learning Rigid Lambek Grammars

In the present section we describe an extension of Buszkowski's RG algorithm [4] for learning rigid BCGs from structured positive examples to rigid Lambek grammars, and rigid minimalist grammars. We also sketch the first author's convergence proof [2] inspired from [10].

4.1 Argument Nodes and Typing Algorithm

Our learning algorithm is based on a process of labeling for the nodes of a set of proof-tree structures. We introduce here the notion of *argument node* for a normal form proof tree. Sometimes we will use the same notation to indicate a node and its type label, since the graphical representation of trees avoids any confusion.

Definition 2. *Let T be a normal form proof-tree structure. Let's define inductively the set $Arg(T)$ of argument nodes of T. There are four cases to consider:*

- *T is a single node, which is the only member of $Arg(T)$;*
- *$T \equiv [\backslash E](T_1, T_2)$, then $Arg(T) = \{Root(T)\} \cup Arg(T_1) \cup Arg(T_2) - Root(T_2)$;*
- *$T \equiv [/E](T_1, T_2)$, then $Arg(T) = \{Root(T)\} \cup Arg(T_1) \cup Arg(T_2) - Root(T_1)$;*
- *$T \equiv [\backslash I](T_1)$ or $T \equiv [/I](T_1)$, then $Arg(T) = Arg(T_1)$.*

Applying the principal type algorithm from λ-calculus to Lambek proofs, as in [20, 2], one obtains the following result on argument nodes:

Proposition 1. *Let T be a well formed normal form proof-tree structure. If each argument node is labeled, then any other node in T can be labeled with one and only one type.*

This proposition allow to define the notion of principal parse:

Definition 3. *A* principal parse *of a proof-tree structure T is a partial parse tree $\mathcal{T}$ of T, such that for any other partial parse tree $\mathcal{T}'$ of T, there exists a substitution σ such that, if a node of T is labeled by type A in $\mathcal{T}$, it's labeled by $\sigma(A)$ in $\mathcal{T}'$.*

To compute in linear time the principal parse of any well-formed normal proof-tree structure, provide argument nodes with distinct variables, and then apply the rules in figure 2 (the $/E$ and $/I$ cases are symmetrical).

4.2 RLG (Rigid Lambek Grammar) Algorithm

– **input:** a finite set D of *normal* proof-trees (since all proof trees normalize)
– **output:** a rigid Lambek grammar G such that $D \subset \mathrm{PL}(G)$, if there is one.

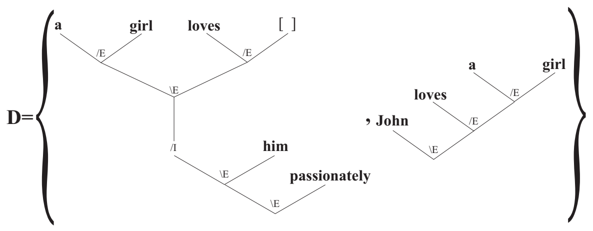

Fig. 3. The set D of positive structured samples for RLG algorithm.

Step 1. Assign a type to each node of the structure in D as follows:

– Assign s to each root node;
– assign distinct variables to the argument nodes (see Fig. 4);
– compute types for the remaining nodes as after definition 3 (Fig. 5).

Step 2. Collect the types assigned to the leaf nodes into a grammar $GF(D)$ called the *general form* induced by D. One has $GF(D) : c \mapsto A$ if and only if the previous step assigns A to a leaf node labeled by symbol c.

$$
\begin{array}{lll}
GF(D): & \textbf{passionately} \mapsto x_1\backslash s & \textbf{him} \mapsto (x_2/x_5)\backslash x_1 \\
& \textbf{a} \mapsto x_3/x_4, x_7/x_8 & \textbf{girl} \mapsto x_4, x_8 \\
& \textbf{loves} \mapsto (x_3\backslash x_2)/x_5, (x_6\backslash s)/x_7 & \textbf{John} \mapsto x_6
\end{array}
$$

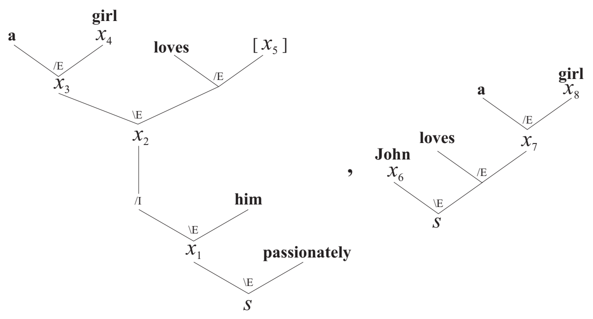

Fig. 4. Set D after labeling argument nodes.

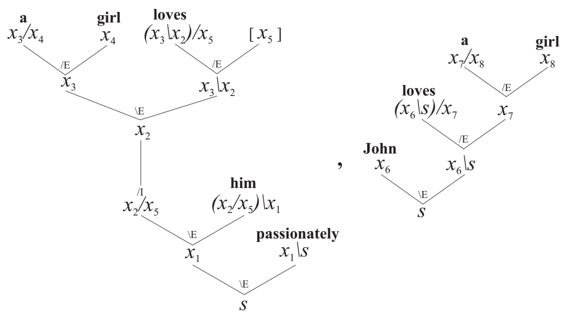

Fig. 5. Set D after computing the label for every node.

Step 3. Unify the types assigned to the same symbol. Let $\mathcal{A} = \{\{A \mid GF(D) : c \mapsto A\} \mid c \in dom(GF(D))\}$, and compute the most general unifier $\sigma = mgu(\mathcal{A})$. There are various well-known algorithms for unification (see [11] for a survey), so a most general unifier of a finite set of types can be computed in linear time. The algorithm fails if unification fails.

$$\sigma = \{x_7 \mapsto x_3, x_8 \mapsto x_4, x_6 \mapsto x_3, x_2 \mapsto s, x_5 \mapsto x_3\}$$

Step 4. Let $RLG(D) = \sigma[GF(D)]$.

$$\begin{array}{lll} RLG(D): & \mathbf{passionately} \mapsto x_1\backslash s & \mathbf{him} \mapsto (s/x_3)\backslash x_1 \\ & \mathbf{a} \mapsto x_3/x_4 & \mathbf{girl} \mapsto x_4 \\ & \mathbf{loves} \mapsto (x_3\backslash s)/x_3 & \mathbf{John} \mapsto x_3 \end{array}$$

Our algorithm is based on the *principal parse algorithm* described in the previous section, which has been proved to be correct and terminate, and the unification algorithm. The result is, intuitively, the *most general* rigid Lambek Grammar which can generate all the proof tree structures appearing in the input sequence.

Theorem 1. *Let φ_{RLG} be the learning function for the grammar system $\langle G_{rigid}, \Sigma^P, \mathrm{PL}\rangle$ defined by $\varphi_{RLG}(\langle T_0, \ldots, T_n\rangle) \simeq RLG(\{T_0, \ldots, T_n\})$. Then φ_{RLG} learns $\mathcal{G}_{rigid}$ from structures.*

We have no room for the complete proof which can be found in [2], we only state the key lemmas below. Given two non necessarily rigid LCGs, let us write $G \sqsubset G'$ whenever there exists a substitution σ such that $\sigma(G) \subset G'$ — that is every type assignment of G is a type assignment of $\sigma(G)$. Given a finite set D of examples, let us call $GF(D)$ the non rigid grammar obtained by collecting all the principal types associated to the words by the examples.

1. Given a grammar G are finitely many grammars H such that $H \sqsubset G$.
2. If $G \sqsubset G'$ then $\mathrm{PL}(G) \subset \mathrm{PL}(G')$.
3. If $D \subset \mathrm{PL}(G)$ then $GF(D) \sqsubset G$.

4. If $GF(D) \sqsubset G$ then $D \subset \mathrm{PL}(G)$.
5. If $RLG(D)$ exists and $RLG(D) \subset G$ then $D \subset FL(RLG(D))$.
6. If $D \subset \mathrm{PL}(G)$ then $RLG(D)$ exists and $RLG(D) \sqsubset G$.

Given these lemmas, it is not difficult to see that $D \subset D' \subset \mathrm{PL}(G)$ entails that $RLG(D), RLG(D'$ exists and $RLG(D) \sqsubset RLG(D') \sqsubset G$. Because of the first item a correct grammar is necessarily met in finitely many steps.

When RLG is applied successively to a sequence of increasing set of proof-tree structures $D_0 \subset D_1 \subset D_2 \subset \cdots$, it is more efficient to make use of the previous value $RLG(D_{i-1})$ to compute the current value $RLG(D_i)$.

It is easy to see that φ_{RLG} can be implemented to run in linear time if positive structured samples are in normal form. As said earlier both the principal type algorithm for Lambek proof trees [20] and efficient unification algorithms [11] are linear.

Learning MCG Rigid minimalist grammars admit a similar algorithm RMG. Here are the differences. To compute the principal type when an $[\otimes]$ rule is met, the subproof sp with conclusion $A \otimes B$ should be typed after the other subproof has been typed, hence the value of $A \otimes B$ is known for typing sp. Whenever a set of examples D is included in $\mathrm{PL}(G)$ then there also exists a substitution σ such that $\sigma(GF(D)) \subset G$. The unification of the collected types is more difficult, because the connective $\otimes$ is associative and commutative, but it has been shown by Fages that unification modulo associativity and commutativity is decidable, even in the presence of other connectives [11]; the difference is that there does not exists a most general unifier, but a finite set of minimal unifiers $(\sigma_i)_{i\in[1,p]}$, and for at least one index $i_0 \in [1,p]$, $\sigma = \tau\sigma_{i_0}$. We define $RMG(D)$ to be $\sigma_{i_0}(GF(D))$. Thus $\tau(RMG(D)) = \tau\sigma_{i_0}(GF(D)) = \sigma(GF(D)) \subset GF(D)$ and $RMG(D) \sqsubset G$. Thus the argument for the convergence of RLG also applies to RMG.

5 Conclusion

Meanwhile we implement some algorithms in the family we presented in Objective CaML with input as DAGs in XML, and interface in Tcl/Tk, we would like to address the following questions:

– What is a good notion of structure for learning syntax from structures, that is a notion of structure which yields good grammars, but is rather computable from corpora?
– How can one automatically obtain parse structure from more realistic examples that can actually be produced by taggers and robust parsers?
– How can we take Montague like semantics into account, first for Lambek grammars, and then for minimalist grammars?
– What are the language classes of rigid grammars?
– Can we used the fact that types for minimalist grammars are of a given shape to avoid associative commutative unification and bound the complexity?
– Does there exists a general result stating that every type logical grammars enjoying certain properties is learnable from structures?

References

1. Y. Bar-Hillel. A quasi arithmetical notation for syntactic description. *Language*, 29:47–58, 1953.
2. Roberto Bonato. A study on learnability of rigid Lambek grammars. Tesi di Laurea & Mémoire de D.E.A, Università di Verona & Université Rennes 1, 2000. available at http://www.irisa.fr/aida/aida-new/Fmembre_rbonato.html.
3. Wojciech Buszkowski. Mathematical linguistics and proof theory. In van Benthem and ter Meulen [21], chapter 12, pages 683–736.
4. Wojciech Buszkowski and Gerald Penn. Categorial grammars determined from linguistic data by unification. *Studia Logica*, 49:431–454, 1990.
5. Noam Chomsky. *The minimalist program*. MIT Press, Cambridge, MA, 1995.
6. Philippe de Groote, Glyn Morrill, and Christian Retoré, editors. *Logical Aspects of Computational Linguistics, LACL'2001*, volume 2014 of *LNCS/LNAI*. Springer-Verlag, 2001.
7. L.R. Gleitman and M. Liberman, editors. *An invitation to cognitive sciences, Vol. 1: Language*. MIT Press, 1995.
8. E.M. Gold. Language identification in the limit. *Information and control*, 10:447–474, 1967.
9. Henk Harkema. A characterisation of minimalist languages. In de Groote et al. [6], pages 193–211.
10. Makoto Kanazawa. *Learnable classes of categorial grammars*. Studies in Logic, Language and Information. FoLLI & CSLI (distributed by Cambridge University Press), 1998. published version of a 1994 Ph.D. thesis (Stanford).
11. Claude Kirchner and Hélène Kirchner. *Rewriting, Solving, Proving*. LORIA, 2000. Book draft available from http://www.loria.fr/˜ckirchne.
12. Joachim Lambek. The mathematics of sentence structure. *American mathematical monthly*, 65:154–169, 1958.
13. Alain Lecomte and Christian Retoré. Extending Lambek grammars: a logical account of minimalist grammars. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics, ACL 2001*, pages 354–361, Toulouse, July 2001. ACL.
14. Jens Michaelis. Transforming linear context-free rewriting systems into minimalist grammars. In de Groote et al. [6], pages 228–244.
15. Michael Moortgat. Categorial type logic. In van Benthem and ter Meulen [21], chapter 2, pages 93–177.
16. Jacques Nicolas. Grammatical inference as unification. Rapport de Recherche RR-3632, INRIA, 1999. http://www.inria.fr/RRRT/publications-eng.html.
17. Christian Retoré, editor. *Logical Aspects of Computational Linguistics, LACL'96*, volume 1328 of *LNCS/LNAI*. Springer-Verlag, 1997.
18. Edward Stabler. Derivational minimalism. In Retoré [17], pages 68–95.
19. Isabelle Tellier. Meaning helps learning syntax. In *Fourth International Colloquium on Grammatical Inference, ICG'98*, 1998.
20. Hans-Jörg Tiede. Lambek calculus proofs and tree automata. In Michael Moortgat, editor, *Logical Aspects of Computational Linguistics, LACL'98, selected papers*, volume 2014 of *LNCS/LNAI*. Springer-Verlag, 2001.
21. J. van Benthem and A. ter Meulen, editors. *Handbook of Logic and Language*. North-Holland Elsevier, Amsterdam, 1997.