

Two models of learning iterated dependencies.

Denis Béchet¹, Alexandre Dikovsky¹, and Annie Foret²

¹ LINA UMR CNRS 6241, Université de Nantes, France
Denis.Bechet@univ-nantes.fr,

Alexandre.Dikovsky@univ-nantes.fr

² IRISA, Université de Rennes1, France
Annie.Foret@irisa.fr

Abstract. We study the learnability problem in the family of Categorical Dependency Grammars (CDG). A class of categorial grammars defining unlimited dependency structures is shown incrementally learnable in the limit, under a reasonable condition on the iterant dependencies.

1 Introduction

The idea of inference of grammars in a family \mathcal{C} is as follows. With every grammar $G \in \mathcal{C}$ is related an “observation set” $\Phi(G)$ of G . It may be the language $L(G)$ or an image of constituent or dependency structures generated by G . An algorithm A is an **inference algorithm** for \mathcal{C} if, for every grammar $G \in \mathcal{C}$, A applies to the **training sequences** for G , i.e. to enumerations σ of $\Phi(G)$ and, for every initial subsequence $\sigma[i] = \{s_1, \dots, s_i\}$ of σ , it returns a **hypothetical grammar** $A(\sigma[i]) = G_i \in \mathcal{C}$. A **learns** a **target grammar** $G \in \mathcal{C}$ if on any training sequence σ for G A stabilizes on a grammar $\mathcal{A}(\sigma[T]) \equiv G$.³ The grammar $\lim_{i \rightarrow \infty} \mathcal{A}(\sigma[i]) = \mathcal{A}(\sigma[T])$ returned at the stabilization step is the **limit grammar**. A **learns** \mathcal{C} if it learns every grammar in \mathcal{C} . \mathcal{C} is **learnable** if there is an inference algorithm learning \mathcal{C} .

The author of this concept E.M. Gold [9] considered $\Phi(G) = L(G)$. In this case they say that the grammars are learned **from strings**. They are learned **from structures** if $\Phi(G)$ is a structure set of G . These concepts were intensively studied (see excellent surveys in [1] and [11]). Most results are pessimistic. In particular, any family of grammars generating all finite languages and at least one infinite language (as it is the case of all classical grammars) is not learnable from strings. Nevertheless, due to several sufficient conditions of learnability, such as **finite elasticity** [16, 13] and **finite thickness** [15], some interesting positive results were obtained. In particular, k -rule string and term generating grammars are learnable from strings for every k [15] and k -**rigid** (i.e. assigning no more than k types per word) classical categorial grammars (CG) are learnable from so called “function-argument” structures and also from strings [4, 11].

³ \mathcal{A} **stabilizes** on σ on step T means that T is the minimal number t for which there is no $t_1 > t$ such that $\mathcal{A}(\sigma[t_1]) \neq \mathcal{A}(\sigma[t])$.

In this paper we study the learnability problem in the family of Categorical Dependency Grammars (CDG) introduced in [8]. CDG is a class of categorial grammars defining unlimited dependency structures. In [5] it is shown that, in contrast with the categorial grammars, the *rigid* (i.e. 1-rigid) CDG are not learnable. This negative effect is due to the use of iterated subtypes. On the other hand, it is also shown that the k -rigid CDG with iteration-free types are learnable from the so called “dependency nets” (an analogue of the function-argument structures adapted to CDG) and also from strings. However, the iteration-free CDG cannot be considered as an acceptable compromise because real application CDG use iterated dependencies. Below, we propose a pragmatic solution of the learnability problem for CDG with iterated dependency subtypes by limiting the family of CDG to the grammars satisfying a strong condition on the so called *iterated*, i.e. unlimited repeatable dependencies (those of noun modifiers and of verb circumstantials). Intuitively, in the grammars satisfying this condition, the iterated dependencies and the dependencies repeatable at least K times for some fixed K are *indiscernible*. This constraint, called below *K -star-revealing*, is more or less generally accepted in the traditional dependency syntax (cf. [12], where $K = 2$). For the class of K -star-revealing CDG, we show an algorithm which incrementally learns the target CDG from the dependency structures in which the iteration is not marked. We compare this new model of learning grammars from structures with the traditional one as applied to iterated dependencies. As one might expect, the CDG with unlimited iterated dependencies are not learnable from input functor/argument-like structures. Moreover, this is true even for the rigid CDG.

2 Background

2.1 Categorical Dependency Grammars

Categorical dependency grammars [7] may be seen as an assignment to words of first order dependency types of the form: $t = [l_m \backslash \dots \backslash l_1 \backslash g / r_1 / \dots / r_n]^P$. Intuitively, $w \mapsto [\alpha \backslash d \backslash \beta]^P$ means that the word w has a left subordinate through dependency d (similar for the right subtypes $[\alpha / d / \beta]^P$). The *head subtype* g in $w \mapsto [\alpha \backslash g / \beta]^P$ intuitively means that w is governed through dependency g . In this way t defines all local (projective) dependencies of a word.

Example 1. For instance, the assignment:
 $in \mapsto [c\text{-copul}/prepos\text{-}in]$, $the \mapsto [det]$, $Word \mapsto [det \backslash pred]$
 $beginning \mapsto [det \backslash prepos\text{-}in]$, $was \mapsto [c\text{-copul} \backslash S / pred]$

determines the projective dependency structure in Fig. 1.

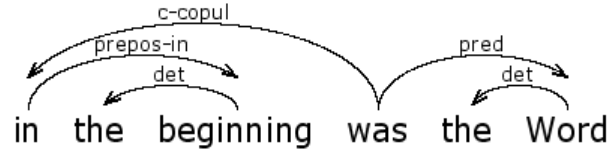


Fig. 1. Projective dependency structure

The intuitive meaning of subtype P , called *potential*, is that it defines the distant (non-projective, discontinuous) dependencies of the word w . P is a string of *polarized valencies*, i.e. of symbols of four kinds: $\swarrow d$ (left negative valency d), $\searrow d$ (right negative valency d), $\nearrow d$ (left positive valency d), $\nwarrow d$ (right positive valency d). Intuitively, $v = \nearrow d$ requires a subordinate through dependency d situated *somewhere* on the left, whereas the *dual* valency $\check{v} = \swarrow d$ requires a governor through the same dependency d situated *somewhere* on the right. So together they describe the discontinuous dependency d . Similar for the other pairs of dual valencies. For negative valencies $\swarrow d, \searrow d$ are provided special kind of subtypes $\#(\swarrow d), \#(\searrow d)$. Intuitively, they serve to check the adjacency of a distant subordinate through discontinuous dependency d to a *host word*. The dependencies of these types are called *anchor*. An *elementary dependency type* is either a *local dependency name* d or its *iteration* d^* or an anchor type $\#(v)$.

Example 2. For instance, the assignment:

$elle \mapsto [pred]$

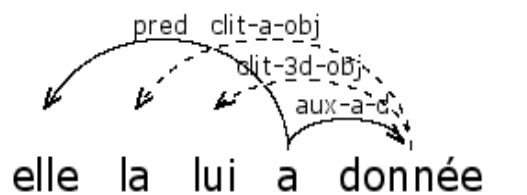
$la \mapsto [\#(\swarrow clit-a-obj)]^{\swarrow clit-a-obj}$

$lui \mapsto [\#(\swarrow clit-3d-obj)]^{\swarrow clit-3d-obj}$

$a \mapsto [\#(\swarrow clit-3d-obj) \#(\swarrow clit-a-obj) \backslash pred \backslash S / aux]$

$donnée \mapsto [aux]^{\nwarrow clit-3d-obj \nwarrow clit-a-obj}$

determines the non projective DS in Fig. 2.



(fr. *she it_{g=fem} to him has given)

Fig. 2. Non-projective dependency structure

Definition 1. Let $w = a_1 \dots a_n$ be a string, W be the set of all occurrences of symbols in w and $C = \{d_1, \dots, d_m\}$ be a set of dependency names. A graph

$D = (W, E)$ with labeled arcs is a dependency structure (DS) of w if it has a root, i.e. a node $a_0 \in W$ such that (i) for any node $a \in W$, $a \neq a_0$, there is a path from a_0 to a and (ii) there is no arc (a', d, a_0) .⁴ An arc $(a_1, d, a_2) \in E$ is called dependency d from a_1 to a_2 . The linear order on W induced by w is the precedence order on D .

Definition 2. Let \mathbf{C} be a set of elementary dependency types (sometimes also called local dependency names) and \mathbf{V} be a set of valency names.

The expressions of the form $\swarrow v$, $\nwarrow v$, $\searrow v$, $\nearrow v$, where $v \in \mathbf{V}$, are called polarized valencies. $\nwarrow v$ and $\nearrow v$ are positive, $\swarrow v$ and $\searrow v$ are negative; $\nwarrow v$ and $\swarrow v$ are left, $\nearrow v$ and $\searrow v$ are right. Two polarized valencies with the same valency name and orientation, but with the opposite signs are dual.

An expression of one of the forms $\#(\swarrow v)$, $\#(\searrow v)$, $v \in \mathbf{V}$, is called anchor type or just anchor. An expression of the form d^* where $d \in \mathbf{C}$, is called iterated dependency type.

Elementary dependency types, iterated dependency types and anchor types are called primitive.

An expression of the form $t = [l_m \setminus \dots \setminus l_1 \setminus H / \dots / r_1 \dots / r_n]$ in which $m, n \geq 0$, $l_1, \dots, l_m, r_1, \dots, r_n$ are primitive types and H is either an elementary dependency type or an anchor type, is called basic (or local) dependency type. l_1, \dots, l_m and r_1, \dots, r_n are respectively left and right argument subtypes of t . When H is not empty, it is called head subtype of t (or head type for short).

A (possibly empty) string P of polarized valencies is called potential.⁵

A dependency type (or a category) is an expression B^P in which B is a basic dependency type and P is a potential. $\mathbf{CAT}(\mathbf{C}, \mathbf{V})$ and $\mathbf{B}(\mathbf{C})$ will denote respectively the set of all dependency types over \mathbf{C} and \mathbf{V} and the set of all basic dependency types over \mathbf{C} .

CDG are defined using the following calculus of dependency types⁶

$$\mathbf{L}^1. C^{P_1} [C \setminus \beta]^{P_2} \vdash [\beta]^{P_1 P_2}$$

$$\mathbf{I}^1. C^{P_1} [C^* \setminus \beta]^{P_2} \vdash [C^* \setminus \beta]^{P_1 P_2}$$

$$\mathbf{\Omega}^1. [C^* \setminus \beta]^P \vdash [\beta]^P$$

$\mathbf{D}^1. \alpha^{P_1 (\swarrow C) P (\nwarrow C) P_2} \vdash \alpha^{P_1 P P_2}$, if the potential $(\swarrow C) P (\nwarrow C)$ satisfies the following pairing rule **FA** (first available):

$$\mathbf{FA} : \quad P \text{ has no occurrences of } \swarrow C, \nwarrow C.$$

\mathbf{L}^1 is the classical elimination rule. Eliminating the argument subtype $C \neq \#(\alpha)$ it constructs the (projective) dependency C and concatenates the potentials. $C = \#(\alpha)$ creates the anchor dependency. \mathbf{I}^1 derives $k > 0$ instances of C . $\mathbf{\Omega}^1$ serves for the case $k = 0$. \mathbf{D}^1 creates discontinuous dependencies. It pairs

⁴ Evidently, every DS is connected and has a unique root.

⁵ In fact, the potentials should be defined as multi-sets. We define them as strings in order to simplify definitions and notation. Nevertheless, to make the things clear, below we will present potentials in the normal form, where all left valencies precede all right valencies.

⁶ We show left-oriented rules. The right-oriented are symmetrical.

and eliminates dual valencies with name C satisfying the rule **FA** to create the discontinuous dependency C .

Definition 3. A categorial dependency grammar (CDG) ⁷ is a system $G = (W, \mathbf{C}, S, \lambda)$, where W is a finite set of words, \mathbf{C} is a finite set of elementary categories containing the selected category S , and λ , called lexicon, is a finite substitution on W such that $\lambda(a) \subset \mathbf{CAT}(\mathbf{C}, \mathbf{V})$ for each word $a \in W$.

For a DS D and a string x , let $G(D, x)$ denote the relation: D is constructed in a proof $\Gamma \vdash S$ for some $\Gamma \in \lambda(x)$. Then the language generated by G is the set $L(G) =_{af} \{w \mid \exists D G(D, w)\}$ and the DS-language generated by G is the set $\Delta(G) =_{af} \{D \mid \exists w G(D, w)\}$. $\mathcal{D}(CDG)$ and $\mathcal{L}(CDG)$ will denote the families of DS-languages and languages generated by these grammars.

Example 3. For instance, the proof in **Fig. 3** shows that the DS in **Fig. 2** belongs to the DS-language generated by a grammar containing the type assignments shown above for the french sentence *Elle la lui a donnée*.

$$\begin{array}{c}
 \frac{\frac{\frac{[\#^l(\swarrow \text{clit} - 3d - \text{obj})]^{\swarrow \text{clit} - 3d - \text{obj}} \ [\#^l(\swarrow \text{clit} - 3d - \text{obj}) \setminus \#^l(\swarrow \text{clit} - a - \text{obj}) \setminus \text{pred} \setminus S / \text{aux}]}{(\mathbf{L}^1)} \quad [\#^l(\swarrow \text{clit} - a - \text{obj})]^{\swarrow \text{clit} - a - \text{obj}}}{\frac{[\text{pred} \setminus S / \text{aux}]^{\swarrow \text{clit} - a - \text{obj}} \ [\#^l(\swarrow \text{clit} - a - \text{obj}) \setminus \text{pred} \setminus S / \text{aux}]^{\swarrow \text{clit} - 3d - \text{obj}}}{(\mathbf{L}^1)} \quad [\text{pred}]}{\frac{[S / \text{aux}]^{\swarrow \text{clit} - a - \text{obj}} \ [\#^l(\swarrow \text{clit} - a - \text{obj}) \setminus \text{pred} \setminus S / \text{aux}]^{\swarrow \text{clit} - 3d - \text{obj}}}{(\mathbf{L}^1)} \quad [\text{aux}]^{\swarrow \text{clit} - 3d - \text{obj}} \setminus \text{clit} - a - \text{obj}}{(\mathbf{L}^r)} \quad \frac{[S]^{\swarrow \text{clit} - a - \text{obj}} \setminus \text{clit} - 3d - \text{obj} \setminus \text{clit} - 3d - \text{obj} \setminus \text{clit} - a - \text{obj}}{S} \quad (\mathbf{D}^1 \times 2)}{S}
 \end{array}$$

Fig. 3. Dependency structure correctness proof

CDG are very expressive. Evidently, they generate all CF-languages. They can also generate non-CF languages.

Example 4. [8]. The CDG:

$$a \mapsto A \swarrow^A, [A \setminus A] \swarrow^A,$$

$$b \mapsto [B/C] \swarrow^A, [A \setminus S/C] \swarrow^A,$$

$$c \mapsto C, [B \setminus C]$$

generates the language $\{a^n b^n c^n \mid n > 0\}$.⁸

Seemingly, the family $\mathcal{L}(CDG)$ of CDG-languages is incomparable with mildly context sensitive languages [10, 14] generated by multi-component TAG, linear CF rewrite systems and some other grammars. $\mathcal{L}(CDG)$ contains non-TAG languages, e.g. $L^{(m)} = \{a_1^n a_2^n \dots a_m^n \mid n \geq 1\}$ for all $m > 0$. In particular, it contains the language $MIX = \{w \in \{a, b, c\}^+ \mid |w|_a = |w|_b = |w|_c\}$ [2], for which E. Bach conjectures that it is not mildly CS. On the other hand, [6] conjectures that this family does not contain the TAG language $L_{copy} = \{xx \mid x \in \{a, b\}^*\}$. This comparison shows a specific nature of the valencies' pairing rule **FA**. It can be expressed in terms of valencies' bracketing. For this, one should interpret $\swarrow d$ and $\nearrow d$ as *left brackets* and $\swarrow d$ and $\searrow d$ as *right brackets*. A potential is *balanced* if it is well bracketed in the usual sense.

⁷ They are called *generalized CDG* in [7] in order to distinguish them from CDG generating DT, which we do not consider here.

⁸ One can see that the DS may be not trees.

CDG have an important property formulated in terms of two images of sequences of categories γ : the *local projection* $\|\gamma\|_l$ and the *valency projection* $\|\gamma\|_v$:
1. $\|\varepsilon\|_l = \|\varepsilon\|_v = \varepsilon$; $\|\alpha\gamma\|_l = \|\alpha\|_l \|\gamma\|_l$ and $\|\alpha\gamma\|_v = \|\alpha\|_v \|\gamma\|_v$ for a category α .
2. $\|C^P\|_l = C$ et $\|C^P\|_v = P$ for every category C^P .

Theorem 1. [6, 7] For a CDG G with lexicon λ and a string x , $x \in L(G)$ iff there is $\Gamma \in \lambda(x)$ such that $\|\Gamma\|_l$ is reduced to S without the rule **D** and $\|\Gamma\|_v$ is balanced.

On this property resides a polynomial time parsing algorithm for CDG [6, 7].

2.2 Learnability, finite elasticity and limit points

A class of languages described by a class of grammars \mathcal{G} is learnable iff there exists a learning algorithm ϕ from finite sets of words to \mathcal{G} that converges⁹, for any $G \in \mathcal{G}$ and for any growing partial enumeration of $L(G)$, to a grammar $G' \in \mathcal{G}$ such that $L(G) = L(G')$.

Learnability and unlearnability properties have been widely studied from a theoretical point of view. A very useful property for our purpose is the finite elasticity property of a class of languages. This term was first introduced in [16, 13] and, in fact, it induces learnability. A very nice presentation of this notion can be found in [11].

Definition 4 (Finite Elasticity). A class \mathcal{L} of languages has infinite elasticity iff $\exists (e_i)_{i \in \mathbb{N}}$ an infinite sequence of sentences, $\exists (L_i)_{i \in \mathbb{N}}$ an infinite sequence of languages of \mathcal{L} such that $\forall i \in \mathbb{N} : e_i \notin L_i$ and $\{e_0, \dots, e_{i-1}\} \subseteq L_i$. A class has finite elasticity iff it has not infinite elasticity.

Theorem 2. [Wright 1989] A class that is not learnable has infinite elasticity.

Corollary 1. A class that has finite elasticity is learnable.

Finite elasticity is a very nice property because it can be extended from a class to every class obtained by a *finite-valued relation*¹⁰. We use here a version of the theorem that has been proved in [11] and is useful for various kinds of languages (strings, structures, nets) that can be described by lists of elements over some alphabets.

Theorem 3. [Kanazawa 1998] Let \mathcal{L} be a class of languages over Γ that has finite elasticity, and let $R \subseteq \Sigma^* \times \Gamma^*$ be a finite-valued relation. Then the class of languages $\{R^{-1}[L] = \{s \in \Sigma^* \mid \exists u \in L \wedge (s, u) \in R\} \mid L \in \mathcal{L}\}$ has finite elasticity.

Definition 5 (Limit Points). A class \mathcal{L} of languages has a limit point iff there exists an infinite sequence $(L_n)_{n \in \mathbb{N}}$ of languages in \mathcal{L} and a language $L \in \mathcal{L}$ such that: $L_0 \subsetneq L_1 \dots \subsetneq \dots \subsetneq L_n \subsetneq \dots$ and $L = \bigcup_{n \in \mathbb{N}} L_n$ (L is a limit point of \mathcal{L}).

⁹ In fact, it is not the output grammars that converge but their associated languages.

¹⁰ A relation $R \subseteq \Sigma^* \times \Gamma^*$ is finite-valued iff for every $s \in \Sigma^*$, there are at most finitely many $u \in \Gamma^*$ such that $(s, u) \in R$.

Limit Points Imply Unlearnability. If the languages of the grammars in a class \mathcal{G} have a limit point then the class \mathcal{G} is *unlearnable*.¹¹

2.3 Limit points for CDGs with iterative subtypes

In [5] it is shown that, in contrast with the classical categorial grammars, the **rigid** (i.e. 1-rigid) CDG are not learnable. This negative effect is due to the use of iterated subtypes. We recall the limit point construction of [5] concerning iterative subtypes and discuss it later.

Limit point construction.

Definition 6. Let S, A, B be three elementary types. Grammars G'_n, G'_* are defined as follows:

$$\begin{aligned} C'_0 &= S \\ C'_{n+1} &= C'_n / A^* / B^* \\ G'_0 &= \{a \mapsto A, b \mapsto B, c \mapsto C'_0\} \\ G'_n &= \{a \mapsto A, b \mapsto B, c \mapsto [C'_n]\} \\ G'_* &= \{a \mapsto A, b \mapsto A, c \mapsto [S / A^*]\} \end{aligned}$$

These constructions yield a limit point as follows [5].

Theorem 4. $L(G'_n) = \{c(b^*a^*)^k \mid k \leq n\}$ and $L(G'_*) = c\{b, a\}^*$

Corollary 2. *The constructions show the non-learnability from strings for the underlying classes of (rigid) grammars: those allowing iterative subtypes (A^*).*

We observe that in these constructions, the number of iterative subtypes (A^*) is not bound.

3 Incremental Learning

Below we show an incremental algorithm strongly learning CDG from DS. This means that $\Delta(G)$ serves as the observation set $\Phi(G)$ and that the limit grammar is strongly equivalent to the target grammar. From the very beginning, it should be clear that, in contrast with the constituent structure grammars and also with the CG, the existence of such learning algorithm is not guaranteed because, due to the iterated subtypes, the straightforward arguments of subtypes' set cardinality do not work. In particular, even the rigid CDG (monotonic with respect to the subgrammar partial order (PO)) do not satisfy the finite thickness condition. On the other hand, the learning algorithm \mathcal{A} below is **incremental** in the sense that every next hypothetical CDG $\mathcal{A}(\sigma[i+1])$ is an "extension" of the preceding grammar $\mathcal{A}(\sigma[i])$ and it is so **without any rigidity constraint**. Incremental learning algorithms are rare. Those we know, are unification based and apply only to *rigid* grammars (cf. [4] and [5]). They cannot be considered as

¹¹ This implies that the class has infinite elasticity.

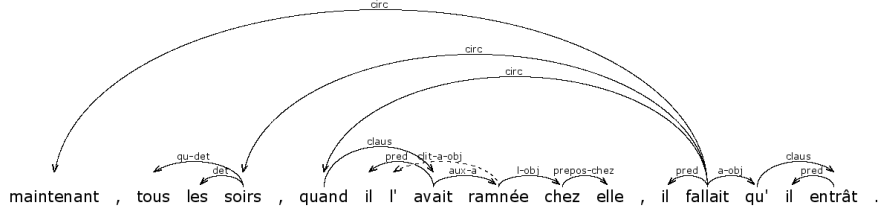
practical (at least for the NLP) because the real application grammars are never rigid. In the cases when the k -rigid learnability is a consequence of the rigid learnability, it is only of a theoretical interest because the existence of a learning algorithm is based on the Kanazawa's finite-valued-relation reduction [11].

Our notion of incrementality is based on a partial "flexibility" order \preceq on CDGs. Basically, the order corresponds to grammar expansion in the sense that $G_1 \preceq G_2$ means that G_2 defines no less dependency structures than G_1 and at least as precise dependency structures as G_1 . This PO is the reflexive-transitive closure of the following preorder $<$.

Definition 7. For a type $t = [l_m \setminus \dots \setminus l_1 \setminus g / r_1 \dots / r_n]^P$ a dependency name c , $i \geq 0$, $0 \leq j \leq m$, let $t_c^{(i \setminus, j)} = [l_m \setminus \dots \setminus l_j \setminus c \dots \setminus c \setminus l_{j-1} \setminus \dots \setminus l_1 \setminus g / r_1 \dots / r_n]^P$ (i times) and $t_c^{(* \setminus, j)} = [l_m \setminus \dots \setminus l_j \setminus c^* \setminus l_{j-1} \setminus \dots \setminus l_1 \setminus g / r_1 \dots / r_n]^P$. Respectively, for $0 \leq k \leq n$ $t_c^{(i /, k)} = [l_m \setminus \dots \setminus l_1 \setminus g / r_1 \dots / r_{k-1} / c \dots / c / r_k / \dots / r_n]^P$ and $t_c^{(* /, k)} = [l_m \setminus \dots \setminus l_1 \setminus g / r_1 \dots / r_{k-1} / c^* / r_k / \dots / r_n]^P$. Then:

1. $t_c^{(i \setminus, j)} < t_c^{(* \setminus, j)}$ and $t_c^{(i /, k)} < t_c^{(* /, k)}$ for all $i \geq 0$, $0 \leq j \leq m$ and $0 \leq k \leq n$
2. $\tau < \tau'$ for sets of types τ, τ' , if either:
 - (i) $\tau' = \tau \cup \{t\}$ for a type $t \notin \tau$ or
 - (ii) $\tau = \tau_0 \cup \{t'\}$ and $\tau' = \tau_0 \cup \{t''\}$
for a set of types τ_0 and some types t', t'' such that $t' < t''$.
3. $\lambda < \lambda'$ for two type assignments λ and λ' , if $\lambda(w') < \lambda'(w')$ for a word w' and $\lambda(w) = \lambda'(w)$ for all words $w \neq w'$.
4. \preceq is the PO which is the reflexive-transitive closure of the preorder $<$.

It is not difficult to prove that the expressive power of CDG monotonically grows with respect to this PO.



(fr. *now all the evenings when he took her home he had to enter [M.Proust])

Fig. 4. Iterated circumstantial dependency

Proposition 1. Let G_1 and G_2 be two CDG such that $G_1 \preceq G_2$. Then $\Delta(G_1) \subseteq \Delta(G_2)$ and $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$.

The flexibility PO \preceq serves to define the following main notion of **incremental learning**.

Definition 8. Let \mathcal{A} be an inference algorithm for CDG from DS, σ be a training sequence for a CDG G and $\mathcal{A}(\sigma[i]) = G_i$ for $i \geq 1$.

1. \mathcal{A} is **monotonic** on σ if $\mathcal{A}(\sigma[i]) \preceq \mathcal{A}(\sigma[j])$ for all $i \leq j$.
2. \mathcal{A} is **faithful** on σ if $\Delta(\mathcal{A}(\sigma[i])) \subseteq \Delta(G)$ for all i .
3. \mathcal{A} is **expansive** on σ if $\sigma[i] \subseteq \Delta(\mathcal{A}(\sigma[i]))$ for all i .

Theorem 5. *Let σ be a training sequence for a CDG G . If an inference algorithm \mathcal{A} is monotonic, faithful, and expansive on σ , and if \mathcal{A} stabilizes on σ then $\lim_{i \rightarrow \infty} \mathcal{A}(\sigma[i]) \equiv_s G$.*

Proof. Indeed, stabilization implies that $\lim_{i \rightarrow \infty} \mathcal{A}(\sigma[i]) = \mathcal{A}(\sigma[T])$ for some T . Then $\Delta(\mathcal{A}(\sigma[T])) \subseteq \Delta(G)$ because of faithfulness. At the same time, by expansiveness and monotonicity, $\Delta(G) = \sigma = \bigcup_{i=1}^{\infty} \sigma[i] \subseteq \bigcup_{i=1}^{\infty} \Delta(\mathcal{A}(\sigma[i])) \subseteq \bigcup_{i=1}^T \Delta(\mathcal{A}(\sigma[i])) \subseteq \Delta(\mathcal{A}(\sigma[T]))$.

As we explain in Section 4, the unlearnability of rigid or k-rigid CDG is due to the use of iterated types. Such types are unavoidable in real grammars (cf. the iterated dependency *circ* in Fig. 4). But in particular in the real application grammars, the iterated types have very special properties. Firstly, the distant and discontinuous dependencies are never iterated. Secondly, in natural languages, the optional constructions repeated successively several times (two or more) are exactly those iterated. This is the resource we use to resolve the learnability problem. To formalize these properties we need some notations and definitions. The main definition concerns a restriction on the class of grammars that is learned. This class corresponds to grammars where an argument that is used at least K times in a DS must be an iterated argument. Such grammars are called **K-star revealing grammars**.

Definition 9.

1. Repetition blocks (*R-blocks*) : For $d \in \mathbf{C}$,

$$LB_d = \{x_1 \setminus \cdots \setminus x_i \mid i > 0, x_i \in \{d, d^{\#}\}\}$$

$$RB_d = \{x_1 / \cdots / x_i \mid i > 0, x_i \in \{d, d^{\#}\}\}$$

2. Patterns: Patterns are defined exactly as types, but in the place of \mathbf{C} , we use \mathbf{G} , where \mathbf{G} is the set of gaps $\mathbf{G} = \{\langle d \rangle \mid d \in \mathbf{C}\}$. Moreover, for any α, β, P and d , $[\alpha \setminus \langle d \rangle \setminus \langle d \rangle \setminus \beta]^P$ and $[\alpha / \langle d \rangle / \langle d \rangle / \beta]^P$ are not a pattern.

3. Vicinity: Let D be a DS in which an occurrence of a word w has : the incoming local dependency h (or the axiom S), the left projective dependencies or anchors l_k, \dots, l_1 (in this order), the right projective dependencies or anchors r_1, \dots, r_m (in this order), and discontinuous dependencies $p_1(d_1), \dots, p_n(d_n)$, where p_1, \dots, p_n are polarities and $d_1, \dots, d_n \in \mathbf{V}$ are valency names.

Then the vicinity of w in D is the type $V(w, D) = [l_1 \setminus \dots \setminus l_k \setminus h / r_m / \dots / r_1]^P$, in which P is a permutation of $p_1(d_1), \dots, p_n(d_n)$ in a standard lexicographical order, for instance, compatible with the polarity order $\nwarrow < \searrow < \swarrow < \nearrow$.

4. Superposition and indexed occurrences of maximal R-blocks :

(i) Let π be a pattern, β_1, \dots, β_k be R-blocks and $\langle d_1 \rangle, \dots, \langle d_k \rangle$ be gaps. Then $\pi(\langle d_1 \rangle \leftarrow \beta_1, \dots, \langle d_k \rangle \leftarrow \beta_k)$ is the expression resulting from π by the parallel substitution of the R-blocks for the corresponding gaps.

(ii) Let E be a type or a vicinity. Then π is superposable on E if for some $\langle d_1 \rangle, \dots, \langle d_k \rangle, \beta_1, \dots, \beta_k$:

- $E = \pi(\langle d_1 \rangle \leftarrow \beta_1, \dots, \langle d_k \rangle \leftarrow \beta_k)$ and
- every R-block β_i is maximal in E .

A vicinity corresponds to the part of a type that is used in a DS. The superposition, in this context, is a notation that puts together in a R-block a list of dependencies with the same name. For instance, in Fig. 4, the vicinity of `fallait` is $[pred \setminus circ \setminus circ \setminus circ \setminus S/a - obj]$. The pattern that is superposable for this vicinity is $\pi = [\langle pred \rangle \setminus \langle circ \rangle \setminus S/ \langle a - obj \rangle]$ and the type is obtained by the following substitution:

$$\pi(\langle pred \rangle \leftarrow pred, \langle circ \rangle \leftarrow circ \setminus circ \setminus circ, \langle a - obj \rangle \leftarrow a - obj)$$

For `ramnée`, the vicinity is $[aux - a/l - obj]^{\setminus clit - a - obj}$ and it is equal to :

$$[aux - a/ \langle l - obj \rangle]^{\setminus clit - a - obj} (\langle l - obj \rangle \leftarrow l - obj)$$

Proposition 2. For every type (vicinity) E there is a single pattern π superposable on E and a single decomposition (R-decomposition)

$$E = \pi(\langle d_1 \rangle \leftarrow \beta_1, \dots, \langle d_k \rangle \leftarrow \beta_k)^P$$

Notation

(i) Let E be a type or a vicinity with the R-decomposition

$$E = \pi(\langle d_1 \rangle \leftarrow \beta_1, \dots, \langle d_k \rangle \leftarrow \beta_k)^P.$$

Then for $1 \leq i \leq k$,

$$E|_i =_{af} \beta_i$$

$$E[i \leftarrow \beta] =_{af} \pi(\langle d_1 \rangle \leftarrow \beta_1, \dots, \langle d_i \rangle \leftarrow \beta, \dots, \langle d_k \rangle \leftarrow \beta_k)^P$$

(ii) Let G be a CDG with lexicon λ and $t \in \lambda(w)$ be a type with R-decomposition $t = \pi(\langle d_1 \rangle \leftarrow \beta_1, \dots, \langle d_k \rangle \leftarrow \beta_k)^P$. Then $G_w^{t[i \leftarrow \beta]}$ denotes the CDG with lexicon $\lambda \cup \{w \mapsto t[i \leftarrow \beta]\}$.

Definition 10. Let $K > 1$ be an integer. CDG G is K -star revealing for dependency d if for every word w , every type $t \in \lambda(w)$ such that for some i , $t = t[i \leftarrow \beta]$ and β has at least one occurrence of d^* or at least K occurrences of d :

$$G_w^{t[i \leftarrow d^*]} \equiv_s G$$

G is K -star revealing if it is K -star revealing for all dependencies.

If we define the grammar $G(t)$ by $A \mapsto [a], B \mapsto [b], C \mapsto t$ where t is a type, we can prove that:

- $G([a^* \setminus S/a^*])$, $G([a^* \setminus b^* \setminus a^*/S])$ and $G([a^* \setminus b \setminus a^* \setminus S])$ are 2-star revealing,
- $G([a^* \setminus a \setminus S])$, $G([a^* \setminus b^* \setminus a \setminus S])$ are not 2-star revealing.

Usually, it means that a K -star revealing grammar can use the same iterated dependency type d^* several times but each occurrence must not be in the same block as the elementary dependency type d . Also, there are less than K occurrences of d in a block.

Theorem 6. *The class $CDG^{K \rightarrow *}$ of K -star revealing CDG is learnable from DS.*

To prove the theorem, we present an inference algorithm $\mathbf{TGE}^{(K)}$ (see Fig. 5) which, for every next DS in a training sequence, transforms the observed local, distant and anchor dependencies of every word into its vicinity. At that, when possible, it generalizes the types with repeated local dependencies by introducing iteration. $\mathbf{TGE}^{(K)}$ is learning $CDG^{K \rightarrow *}$ due to the following two statements.

Algorithm $\mathbf{TGE}^{(K)}$ (type-generalize-expand):
Input: $\sigma[i]$ (σ being a training sequence).
Output: CDG $\mathbf{TGE}^{(K)}(\sigma[i])$.
let $G_H = (W_H, \mathbf{C}_H, S, \lambda_H)$ where
 $W_H := \emptyset; \mathbf{C}_H := \{S\}; \lambda_H := \emptyset; k := 0$
(loop) **for** $i \geq 0$ //Infinite loop on σ
let $\sigma[i+1] = \sigma[i] \cdot D$;
let $(x, E) = D$;
(loop) **for every** $w \in x$
 $W_H := W_H \cup \{w\}$;
let $V(w, D) = \pi(\langle d_1 \rangle \leftarrow \beta_1, \dots, \langle d_k \rangle \leftarrow \beta_k)^P$
(loop) **for** $j := 1, \dots, k$
if $\beta_j \in LD_d \cup RD_d$ **and** $length(\beta_j) \geq K$
then $\gamma_j := d^*$
else $\gamma_j := \beta_j$ **end end**
let $t_w := \pi(\langle d_1 \rangle \leftarrow \gamma_1, \dots, \langle d_k \rangle \leftarrow \gamma_k)^P$
 $\lambda_H(w) := \lambda_H(w) \cup \{t_w\}$; // expansion
end end

Fig. 5. *Inference algorithm $\mathbf{TGE}^{(K)}$*

Lemma 1. *The inference algorithm $\mathbf{TGE}^{(K)}$ is monotonic, faithful and expansive on every training sequence σ of a K -star revealing CGD.*

Proof. By definition, the algorithm $\mathbf{TGE}^{(K)}$ is monotonic. It is expansive because for $\sigma[i]$, we add types to the grammar that are based on the vicinities of the words of $\sigma[i]$. Thus, $\sigma[i]$ is a DS of $\mathbf{TGE}^{(K)}(\sigma[i])$. To prove that $\mathbf{TGE}^{(K)}$ is faithful for $\sigma[i]$ of $\Delta(G)$, we must prove that $\Delta(\mathbf{TGE}^{(K)}(\sigma[i])) \subseteq \Delta(G)$. In fact, this property comes from the fact that G is K -star revealing. Because $\Delta(\mathbf{TGE}^{(K)})$ is monotonic, we only have to show that the new types introduced by the last DS D of $\sigma[i]$, do not result in DS that are not in $\Delta(G)$. In fact, for this DS D , $\mathbf{TGE}^{(K)}$ introduces the types that come from vicinities $V(w^j, D) = \pi^j(\langle d_1^j \rangle \leftarrow \beta_1^j, \dots, \langle d_{k^j}^j \rangle \leftarrow \beta_{k^j}^j)^{P^j}$, $1 \leq j \leq J$, where $w^j, 1 \leq j \leq J$ are the words of D . Because $D \in \Delta(G)$, $\Delta(G \cup \{V(w^j, D), j = 1, \dots, J\}) = \Delta(G)$. By induction on the number of R-blocks $\beta_{k^j}^j$, we prove that when $\gamma_{k^j}^j$ is computed by $\mathbf{TGE}^{(K)}$ either as $\beta_{k^j}^j$ or as a generalization d^* (when the length of $\beta_{k^j}^j$ is greater or equal to K), $D \in \Delta(G)$, $\Delta(G \cup \{t^j, j = 1, \dots, J\}) = \Delta(G)$, where $t^j = \pi^j(\langle d_1^j \rangle \leftarrow \gamma_1^j, \dots, \langle d_{k^j}^j \rangle \leftarrow \gamma_{k^j}^j)^{P^j}$ is also true. On each induction step, we transform one of the $\beta_{k^j}^j$ into a $\gamma_{k^j}^j$. If $\beta_{k^j}^j = \gamma_{k^j}^j$, there is nothing to prove.

If $\beta_{kj}^j \neq \gamma_{kj}^j$, it means that $\gamma_{kj}^j = d^*$ for some d . The length of β_{kj}^j is greater than K , thus we can apply the K -star revealing property of G to the grammar found at the previous step : we can replace β_{kj}^j by d^* without changing the DS language.

Lemma 2. *The inference algorithm $\mathbf{TGE}^{(K)}$ stabilizes on every training sequence σ of a K -star revealing CGD.*

Proof. Because G has a finite number of types, the number of corresponding patterns is also finite. Thus the number of patterns that correspond to the DS in $\Delta(G)$ (and of course in σ) is also finite. Because the R-blocks are generalized using $*$ by $\mathbf{TGE}^{(K)}$ when their length is greater or equal to K , the number of R-blocks used by $\mathbf{TGE}^{(K)}$ is finite. Thus the number of generated types is finite and the algorithm certainly stabilizes.

4 Learnability from Positive Examples

4.1 Original algorithm on functor-argument data

FA structures Let Σ be an alphabet, a *FA structure* over Σ is a binary tree where each leaf is labelled by an element of Σ and each internal node is labelled by the name of the binary rule.

Background - RG algorithm We recall Buszkowski's Algorithm called RG as in [11] it is defined for AB grammars, based on $/_e$ and \setminus_e (binary elimination rules, like the local rules of CDG \mathbf{L}^f and \mathbf{L}^l , without potentials) :

$$/_e : A / B, B \Rightarrow A \quad \text{and} \quad \setminus_e : B, B \setminus A \Rightarrow A$$

The RG algorithm takes a set D of functor-argument structures as positive examples and returns a rigid grammar $RG(D)$ compatible with the input if there is one (compatible means that D is in the set of functor-argument structures generated by the grammar).

Sketch of RG-algorithm, computing $RG(D)$:

1. assign S to the root of each structure
2. assign distinct variables to argument nodes
3. compute the other types on functor nodes according to $/_e$ and \setminus_e
4. collect the types assigned to each symbol, this provides $GF(D)$
5. unify (classical unification) the types assigned to the same symbol in $GF(D)$, and compute the most general unifier σ_{mgu} of this family of types.
6. The algorithm fails if unification fails, otherwise the result is the application of σ_{mgu} to the types of $GF(D)$: $RG(D) = \sigma_{mgu}(GF(D))$.

4.2 Functor-argument structures for CDG with iteration

The **functor-argument structure** and **labelled functor-argument structure** associated to a dependency structure proof, are obtained as below.

Definition 11. Let \mathcal{D} be a dependency structure proof, ending in a type t . The **labelled functor-argument structure** associated to \mathcal{D} , denoted $lfa_{iter}(\mathcal{D})$, is defined by induction on the length of the dependency proof \mathcal{D} considering the last rule in \mathcal{D} :

- if \mathcal{D} has no rule, then it is reduced to a type t assigned to a word w , let $lfa_{iter}(\mathcal{D}) = w$;
 - if the last rule is: $c^{P_1} [c \setminus \beta]^{P_2} \vdash [\beta]^{P_1 P_2}$, by induction let \mathcal{D}_1 be a dependency structure proof for c^{P_1} and $\mathcal{T}_1 = lfa_{iter}(\mathcal{D}_1)$; and let \mathcal{D}_2 be a dependency structure proof for $[c \setminus \beta]^{P_2}$ and $\mathcal{T}_2 = lfa_{iter}(\mathcal{D}_2)$: then $lfa_{iter}(\mathcal{D})$ is the tree with root labelled by $\mathbf{L}^1_{[c]}$ and subtrees $\mathcal{T}_1, \mathcal{T}_2$;
 - if the last rule is: $[c^* \setminus \beta]^{P_2} \vdash [\beta]^{P_2}$, by induction let \mathcal{D}_2 be a dependency structure proof for $[c^* \setminus \beta]^{P_2}$ and $\mathcal{T}_2 = lfa_{iter}(\mathcal{D}_2)$: then $lfa_{iter}(\mathcal{D})$ is \mathcal{T}_2 ;
 - if the last rule is: $c^{P_1} [c^* \setminus \beta]^{P_2} \vdash [c^* \setminus \beta]^{P_1 P_2}$, by induction let \mathcal{D}_1 be a dependency structure proof for c^{P_1} and $\mathcal{T}_1 = lfa_{iter}(\mathcal{D}_1)$ and let \mathcal{D}_2 be a dependency structure proof for $[c^* \setminus \beta]^{P_2}$ and $\mathcal{T}_2 = lfa_{iter}(\mathcal{D}_2)$: $lfa_{iter}(\mathcal{D})$ is the tree with root labelled by $\mathbf{L}^1_{[c]}$ and subtrees $\mathcal{T}_1, \mathcal{T}_2$;
 - we define similarly the function lfa_{iter} when the last rule is on the right, using $/$ and \mathbf{L}^r instead of \setminus and \mathbf{L}^1 ;
 - if the last rule is the one with potentials, $lfa_{iter}(\mathcal{D})$ is taken as the image of the proof above.
- is applied k times with this type.*

The functor-argument, $fa_{iter}(\mathcal{D})$ is the one obtained from $lfa_{iter}(\mathcal{D})$ (the labelled one), by erasing the labels $[c]$.

Example 5. Let $\lambda(\text{John}) = N$, $\lambda(\text{ran}) = [N \setminus S / A^*]$, $\lambda(\text{fast}) = \lambda(\text{yesterday}) = A$, then $s'_3 = \mathbf{L}^1_{[N]}(\text{John}, \mathbf{L}^r_{[A]}(\mathbf{L}^r_{[A]}(\text{ran}, \text{fast}), \text{yesterday})$ (labelled structure) and $s_3 = \mathbf{L}^1(\text{John}, \mathbf{L}^r(\mathbf{L}^r(\text{ran}, \text{fast}), \text{yesterday}))$ are associated to \mathcal{D}_1 below :

$$\mathcal{D}_1 : \frac{\frac{\frac{N}{[N \setminus S]} \mathbf{L}^1}{[N \setminus S / A^*]} \mathbf{L}^1}{[N \setminus S / A^*] A} \mathbf{L}^r \quad \frac{A}{[N \setminus S / A^*] A} \mathbf{L}^r}{[N \setminus S / A^*] A} \mathbf{I}^r \quad \frac{[N \setminus S / A^*] A}{[N \setminus S / A^*]} \mathbf{I}^r}{[N \setminus S / A^*]} \mathbf{I}^r \quad \frac{[N \setminus S / A^*]}{N} \mathbf{I}^r}{S} \mathbf{L}^1$$

John ran fast yesterday
(dependency structure)

4.3 On RG-like algorithms and iteration

Example 6. We consider the following functor-argument structures :

$$\begin{aligned} s_1 &= \mathbf{L}^1(\text{John}, \text{ran}) \\ s_2 &= \mathbf{L}^1(\text{John}, \mathbf{L}^r(\text{ran}, \text{fast})) \\ s_3 &= \mathbf{L}^1(\text{John}, \mathbf{L}^r(\mathbf{L}^r(\text{ran}, \text{fast}), \text{yesterday})) \\ s_4 &= \mathbf{L}^1(\text{John}, \mathbf{L}^r(\mathbf{L}^r(\mathbf{L}^r(\text{ran}, \text{fast}), \text{yesterday}), \text{nearby})) \end{aligned}$$

An RG-like algorithm could compute the following assignments and grammar from $\{s_1, s_2, s_3\}$:

$$\begin{aligned} & \mathbf{L}^1(\text{John} : X_1, \text{ran} : X_1 \setminus S) : S \\ & \mathbf{L}^1(\text{John} : X'_1, \mathbf{L}^r(\text{ran} : X'_1 \setminus S / X_2, \text{fast} : X_2) : X'_1 \setminus S) : S \\ & \mathbf{L}^1(\text{John} : X''_1, \mathbf{L}^r(\mathbf{L}^r(\text{ran} : X''_1 \setminus S / X''_2 / X'_2, \text{fast} : X'_2) : X''_1 \setminus S / X''_2, \\ & \quad \text{yesterday} : X''_2) : X''_1 \setminus S) : S \\ & \dots \end{aligned}$$

	general form	unification	flat rigid grammar for 2-iteration
<i>John</i>	X_1, X'_1, X''_1	$X_1 = X'_1 = X''_1$	X_1
<i>ran</i>	$X_1 \setminus S$ $X'_1 \setminus S / X_2$ $X''_1 \setminus S / X''_2 / X'_2$	<i>fails</i>	$X_1 \setminus S / X_2^*$ with $X_2 = X'_2 = X''_2$
<i>fast</i>	X_2, X'_2	X_2	X_2
<i>yesterday</i>	X''_2	X''_2	X_2

Notice that the next example s_4 would not change the type of *ran*.

In fact, such an RG-like algorithm, when the class of grammars is restricted to rigid grammars, when positive examples are functor-argument structures (without category names), cannot converge (in the sense of Gold).

This can be seen, as explained below, using the same grammars as in the limit point construction for string languages in [5], involving $*$ categories. In fact, the functor-argument structures are all flat structures, with only $/$ operators.

$$\begin{aligned} C'_0 &= S & G'_0 &= \{a \mapsto A, b \mapsto B, c \mapsto C'_0\} \\ C'_{n+1} &= C'_n / A^* / B^* & G'_n &= \{a \mapsto A, b \mapsto B, c \mapsto [C'_n]\} \\ & & G'_* &= \{a \mapsto A, b \mapsto A, c \mapsto [S / A^*]\} \end{aligned}$$

Positive structured examples are then of the form :

$$c, \mathbf{L}^r(c, b), \mathbf{L}^r(\mathbf{L}^r(c, b), b), \mathbf{L}^r(c, a), \mathbf{L}^r(\mathbf{L}^r(c, a), a), \mathbf{L}^r(\mathbf{L}^r(c, b), a), \dots$$

Definition 12. We define $\text{flat}_{\mathbf{L}^r[A]}$ on words by : $\text{flat}_{\mathbf{L}^r[A]}(x1) = x1$ for words of length 1, and $\text{flat}_{\mathbf{L}^r[A]}(x1.w1) = \mathbf{L}^r_{[A]}(x, \text{flat}_{\mathbf{L}^r[A]}(w1))$; we extend the notation $\text{flat}_{\mathbf{L}^r[A]}$ to sets of words (as the set word images).

Definition 13. We define $\text{flat}_{\mathbf{L}^r}$ and $\text{flat}_{\mathbf{L}^r[A]}$ on words by : $\text{flat}_{\mathbf{L}^r}(x1) = x1 = \text{flat}_{\mathbf{L}^r[A]}(x1)$ for words of length 1, and $\text{flat}_{\mathbf{L}^r}(x1.w1) = \mathbf{L}^r(x, \text{flat}_{\mathbf{L}^r}(w1))$; $\text{flat}_{\mathbf{L}^r[A]}(x1.w1) = \mathbf{L}^r_{[A]}(x, \text{flat}_{\mathbf{L}^r[A]}(w1))$; we extend the notation $\text{flat}_{\mathbf{L}^r}$ and $\text{flat}_{\mathbf{L}^r[A]}$ to sets of words (as the set of word images).

Let $FL(G)$ denote the language of functor-arguments structures of G , we have :

Theorem 7. $FL(G'_n) = \text{flat}_{\mathbf{L}^r}(\{c(b^*a^*)^k \mid k \leq n\})$ and $FL(G'_*) = \text{flat}_{\mathbf{L}^r}(c\{b, a\}^*)$

Corollary 3. The limit point establishes the non-learnability from functor-argument structures for the underlying classes of (rigid) grammars: those allowing iterative categories (A^*).

A *limit point, for labelled functor-arguments structures*. If we drop restrictions such as k -rigid, and consider learnability from labelled functor-arguments structures, we have a limit point as follows :

$$\begin{aligned} C_0 &= S \\ C_{n+1} &= (C_n / A) \\ G_0 &= \{a \mapsto A, c \mapsto C_0\} \\ G_n &= \{a \mapsto A, c \mapsto [C_n], c \mapsto [C_{n-1}], \dots c \mapsto C_0\} \\ G_* &= \{a \mapsto [A], c \mapsto [S / A^*]\} \end{aligned}$$

In fact, the functor-argument structures are all flat structures, with only / operators and always the same label A . Let $LFL(G)$ denote the language of labelled functor-argument structures of G , we have :

Theorem 8. $LFL(G_n) = flat_{\mathbf{L}^r_{[A]}}(\{c a^k \mid k \leq n\})$ and $LFL(G_*) = flat_{\mathbf{L}^r_{[A]}}(c a^*)$

Corollary 4. *The limit point establishes the non-learnability from labelled functor-argument structures for the underlying classes of grammars: those allowing iterative categories (A^*).*

The similar question for rigid or k -rigid CDG with iteration is left open.

4.4 Bounds and string learnability

A List-like Simulation In order to simulate an iterated type such that :

$$[\beta / a^*]^{P_0} a^{P_1} \dots a^{P_n} \vdash [\beta]^{P_0 P_1 \dots P_n}$$

we can distinguish two types, one type a for a first use in a sequence and one type $a \setminus a$ for next uses in a sequence of elements of type a (this encodes in fact one or more iterations of a). As in :

$$\begin{array}{ccccccc} \textit{John} & \textit{ran} & \textit{fast} & \textit{yesterday} & \textit{nearby} & & \\ n & n \setminus s / a & a & a \setminus a & a \setminus a & & \end{array}$$

We have two assignments for \textit{ran} : in “John ran”, $\textit{ran} \mapsto n \setminus s$ but in “John ranfast, yesterday”, “ran” $\mapsto n \setminus s / a$. Unfortunately, this approach increases the number of types in the lexicon: if a type has N iterated subtypes, the simulation associates 2^N types. A similar encoding is given for an extension of pregroups in [3].

As it concerns structures, note that, however, such a simulation can change the dependency structure (in the example, the last adverbs are linked to a preceding adverb, rather than directly to the verb). It is pertinent only for strings or for theoretical issues.

Bounds As a corollary, for a class of CDG *without potentials* for which the number of iterated types is bound by a fixed N , the simulation leads to a class of grammars without iterated types, which is also k -rigid: the number of assignments per word is bound by a large but fixed number ($k = 2^N$). This means that the class of rigid CDG allowing at most N iterated types is learnable from strings.

The fact also extends to k -rigid CDG, not only to rigid (1-rigid) CDG.

5 Conclusion

We propose a new model of incremental learning of categorial dependency grammars with unlimited iterated types from input dependency structures without marked iteration. It reflects the real situation of deterministic inference of a dependency grammar from a dependency treebank. The sufficient condition of K -star-revealing we use is widely accepted in traditional linguistics for small K , which makes this model interesting for practical use. On the other hand, in this paper this condition is defined in “semantic” terms. It is an interesting question, whether one can find its syntactic decidable formulation.

References

1. Angluin, D.: Inductive inference of formal languages from positive data. *Information and Control* 45, 117–135 (1980)
2. Béchet, D., Dikovskiy, A., Foret, A.: Dependency structure grammars. In: Proc. of the 5th Int. Conf. “Logical Aspects of Computational Linguistics” (LACL’2005). pp. 18–34. LNAI 3492 (2005)
3. Béchet, D., Dikovskiy, A., Foret, A., Garel, E.: Optional and iterated types for pregroup grammars. In: Proceedings of LATA 2008: 2nd International Conference on Language and Automata Theory and Applications, LNCS Vol. 5196 (2008)
4. Buszkowski, W., Penn, G.: Categorial grammars determined from linguistic data by unification. *Studia Logica* 49, 431–454 (1990)
5. Bchet, D., Dikovskiy, A., Foret, A., Moreau, E.: On learning discontinuous dependencies from positive data. In: Proc. of the 9th Intern. Conf. “Formal Grammar 2004” (FG 2004). pp. 1–16. Nancy, France (Aug 2004)
6. Dekhtyar, M., Dikovskiy, A.: Categorial dependency grammars. In: Proc. of Intern. Conf. on Categorial Grammars. pp. 76–91. Montpellier (2004)
7. Dekhtyar, M., Dikovskiy, A.: Generalized categorial dependency grammars. In: Trakhtenbrot/Festschrift, pp. 230–255. LNCS 4800, Springer (2008)
8. Dikovskiy, A.: Dependencies as categories. In: “Recent Advances in Dependency Grammars”. COLING’04 Workshop. pp. 90–97 (2004)
9. Gold, E.M.: Language identification in the limit. *Information and control* 10, 447–474 (1967)
10. Joshi, A.K., Shanker, V.K., Weir, D.J.: The convergence of mildly context-sensitive grammar formalisms. In: Foundational issues in natural language processing. pp. 31–81. Cambridge, MA (1991)
11. Kanazawa, M.: Learnable classes of categorial grammars. *Studies in Logic, Language and Information, FoLLI & CSLI* (1998)
12. Mel’čuk, I.: *Dependency Syntax*. SUNY Press, Albany, NY (1988)
13. Motoki, T., Shinohara, T., Wright, K.: The correct definition of finite elasticity: Corrigendum to identification of unions. In: The fourth Annual Workshop on Computational Learning Theory. p. 375. San Mateo, Calif. (1991)
14. Shanker, V.K., Weir, D.J.: The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory* 27, 511–545 (1994)
15. Shinohara, T.: Inductive inference of monotonic formal systems from positive data. *New Generation Computing* 8(4), 371–384 (1991)
16. Wright, K.: Identifications of unions of languages drawn from an identifiable class. In: The 1989 Workshop on Computational Learning Theory. pp. 328–333. San Mateo, Calif. (1989)