# Two Models of Learning Iterated Dependencies.

Denis Béchet[1], Alexandre Dikovsky[1], and Annie Foret[2]

[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 Categorial Dependency Grammars (CDG), a class of categorial grammars defining unlimited dependency structures. CDG satisfying a reasonable condition on iterated (i.e., repeatable and optional) dependencies are shown to be incrementally learnable in the limit.

## 1 Introduction

The idea of grammatical inference is as follows. A class of languages defined using a class of grammars $\mathcal{G}$ is learnable if there exists a learning algorithm $\phi$ from finite sets of words generated by the target grammar $G_0 \in \mathcal{G}$ to hypothetical grammars in $\mathcal{G}$, such that $(i)$ the sequence of languages generated by the output grammars converges to the target language $L(G_0)$ and $(ii)$ this is true for any increasing enumeration of finite sublanguages of $L(G_0)$.

This concept due to E.M. Gold [8] is also called `learning from strings`. More generally, the hypothetical grammars may be generated from finite sets of structures defined by the target grammar. This kind of learning is called `learning from structures`. Both concepts were intensively studied (see excellent surveys in [1] and [10]). 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` [15, 12] and `finite thickness` [14], some interesting positive results were obtained. In particular, $k$-rule string and term generating grammars are learnable from strings for every $k$ [14] 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, 10].

In this paper we study the learnability problem in the family of Categorial Dependency Grammars (CDG) introduced in [7]. CDG is a class of categorial grammars defining unlimited dependency structures. In [3] 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 which express the *iterated* dependencies i.e. unlimited repeatable optional dependencies (those of noun modifiers and of verb circumstantials). 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 the linguistically relevant dependency grammars must express the iterated dependencies. Below we propose a pragmatic solution of the learnability problem for CDG with iterated dependency subtypes. It consists in limiting the family of CDG to the grammars satisfying a strong condition on the iterated dependencies. 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. [11], 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 model 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 Categorial Dependency Grammars

*Categorial dependency grammars* [6] may be seen as an assignment to words of first order dependency types of the form: $t = [l_m \backslash \ldots \backslash l_1 \backslash g / r_1 / \ldots / r_n]^P$. 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:
$in \mapsto [c{-}copul / prepos{-}in]$, $the \mapsto [det]$, $Word \mapsto [det \backslash pred]$
$beginning \mapsto [det \backslash prepos{-}in]$, $was \mapsto [c{-}copul \backslash S / pred]$
determines the projective dependency structure in 1.

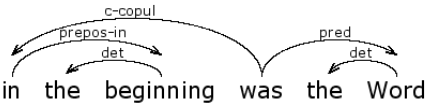
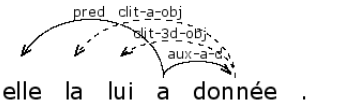



**Fig. 1.** Projective dependency structure

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* $d$), $\nwarrow d$ (*left positive valency* $d$), $\nearrow d$ (*right positive valency* $d$). Intuitively, $v = \nwarrow d$ requires a subordinate through dependency $d$ situated *somewhere* on the left, whereas the *dual* valency $\breve{v} = \swarrow d$ 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 a special kind of subtypes $\#(\swarrow d)$, $\#(\searrow d)$. Intuitively, they serve to check the adjacency of a distant word subordinate through discontinuous dependency $d$ to a *host word*. The dependencies of these types are called *anchor*. A *primitive dependency type* is either a *local dependency name* $d$ or its *iteration* $d^*$ or an anchor type $\#(v)$.

*Example 2.* For instance, the assignment:
$elle \mapsto [pred]$, $la \mapsto [\#(\swarrow clit{-}a{-}obj)]^{\swarrow clit\text{–}a\text{–}obj}$,
$lui \mapsto [\#(\swarrow clit{-}3d{-}obj)]^{\swarrow clit\text{–}3d\text{–}obj}$, $donnée \mapsto [aux]^{\nwarrow clit\text{–}3d\text{–}obj \nwarrow clit\text{–}a\text{–}obj}$,
$a \mapsto [\#(\swarrow clit{-}3d{-}obj)\backslash\#(\swarrow clit{-}a{-}obj)\backslash pred\backslash S/aux{-}a{-}d]$
determines the non projective DS in Fig. 2.[3]




(*fr.* $^*$*she* $it_{\mathbf{g}=fem}$ *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* **C** *be a set of* local dependency names *and* **V** *be a set of* valency names.

[3] Anchors are not displayed for a better readability.
[4] Evidently, every DS is connected and has a unique root.

*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.

*Local dependency names, iterated dependency types and anchor types are* primitive types.

*An expression of the form* $t = [l_m \backslash \ldots \backslash l_1 \backslash H / \ldots / r_1 \ldots / r_n]$ *in which* $m, n \geq 0$, $l_1, \ldots, l_m, r_1, \ldots, r_n$ *are primitive types and* $H$ *is either a local dependency name or an anchor type, is called* basic dependency type. $l_1, \ldots, l_m$ *and* $r_1, \ldots, r_n$ *are respectively left and right* argument subtypes *of* $t$. $H$ *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 *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 types [6]

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

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

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

$\mathbf{D^1}$. $\alpha^{P_1(\swarrow C)P(\nwarrow C)P_2} \vdash \alpha^{P_1 P P_2}$, 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 and eliminates dual valencies with name $C$ satisfying the rule **FA** to create the discontinuous dependency $C$.

**Definition 3.** *A* categorial dependency grammar (CDG) *is a system* $G = (W, \mathbf{C}, S, \lambda)$, *where* $W$ *is a finite set of words,* $\mathbf{C}$ *is a finite set of local dependency names containing the selected name* $S$ *(an axiom), and* $\lambda$, *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*

[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.

*set* $L(G)=_{df}\{w \mid \exists D\ G(D,w)\}$ *and the* DS-language *generated by* $G$ *is the set* $\Delta(G)=_{df}\{D \mid \exists w\ G(D,w)\}$. $\mathcal{D}(CDG)$ *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*.

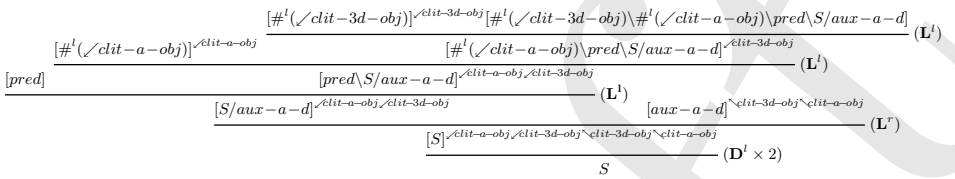



**Fig. 3.** Dependency structure correctness proof

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

*Example 4.* [7]. The CDG:
$a \mapsto A^{\swarrow A}, [A\backslash A]^{\swarrow A}$, $b \mapsto [B/C]^{\nwarrow A}, [A\backslash S/C]^{\nwarrow A}$, $c \mapsto C, [B\backslash C]$
generates the language $\{a^n b^n c^n \mid n > 0\}$.[7]

Seemingly, the family $\mathcal{L}(CDG)$ of CDG-languages is different from that of the mildly context sensitive languages [9, 13] generated by multi-component TAG, linear CF rewrite systems and some other grammars. $\mathcal{L}(CDG)$ contains non-TAG languages, e.g. $L^{(m)} = \{a_1^n a_2^n ... a_m^n \mid n \geq 1\}$ 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 has conjectured that it is not mildly CS. On the other hand, [5] 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 $\nwarrow d$ and $\searrow d$ as *right brackets*. A potential is *balanced* if it is well bracketed in the usual sense.

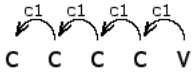
CDG have an important property formulated in terms of two images of sequences of types $\gamma$: the *local projection* $\|\gamma\|_l$ and the *valency projection* $\|\gamma\|_v$:
**1**. $\|\varepsilon\|_l = \|\varepsilon\|_v = \varepsilon$; $\|\alpha\gamma\|_l = \|\alpha\|_l\|\gamma\|_l$ and $\|\alpha\gamma\|_v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

**Theorem 1.** *[5, 6] For a CDG* $G$ *with lexicon* $\lambda$ *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.*

[7] One can see that the DS may be not trees.

On this property resides a polynomial time parsing algorithm for CDG [5, 6].

It is important to understand why the iterated subtypes are unavoidable in dependency grammars. This is due to one of the basic principles of dependency syntax, which concerns the optional repeatable dependencies (cf. [11]): all modifiers of a noun $n$ share $n$ as their *governor* and, similar, all circonstants of a verb $v$ share $v$ as their *governor*. For instance, in the dependency structure in Figure 4 there are three circonstants dependent on the same verb *fallait* (`fr.` *had to*). In particular, this means that the iterated dependencies cannot be simulated through recursive types. Indeed, $a \mapsto [\alpha\backslash d]$ and $b \mapsto [d\backslash\beta]$ derives the dependency $a \overset{d}{\longleftarrow} b$ for *ab*. Therefore, the recursive types derive sequenced dependencies. E.g., $v \mapsto [c1\backslash S]$, $c \mapsto [c1\backslash c1], [c1]$ derives for *ccccv* the

DS: 

contradicting the above mentioned principle.

### 2.2 Learnability, Finite Elasticity and Limit Points

With every grammar $G \in \mathcal{C}$ is related an `observation set` $\Phi(G)$ of $G$. This may be the generated language $L(G)$ or an image of the constituent or dependency structures generated by $G$. Below we call `training sequence` for $G$ an enumeration of $\Phi(G)$. An algorithm $A$ is an `inference algorithm` for $\mathcal{C}$ if, for every grammar $G \in \mathcal{C}$, $A$ applies to its training sequences $\sigma$ of $\Phi(G)$ and, for every initial subsequence $\sigma[i] = \{s_1, \ldots, s_i\}$ of $\sigma$, it returns a `hypothetical grammar` $A(\sigma[i]) \in \mathcal{C}$. $A$ `learns` a `target grammar` $G \in \mathcal{C}$ if on any training sequence $\sigma$ for $G$ $A$ stabilizes on a grammar $\mathcal{A}(\sigma[T]) \equiv G$.[8] The grammar $\lim_{i\to\infty} \mathcal{A}(\sigma[i]) = \mathcal{A}(\sigma[T])$ 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 $\mathcal{C}$.

Learnability and unlearnability properties have been widely studied from a theoretical point of view. In particular, in [15, 12] was introduced finite elasticity, a property of classes of languages implying their learnability. The following elegant presentation of this property is cited from [10].

**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 $\{e_0, \ldots, e_{i-1}\} \subseteq L_i$. 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.*

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

The finite elasticity can be extended from a class to every class obtained by a *finite-valued relation*[9]. We use here a version of the theorem that has been proved in [10] 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 $\Gamma$ 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 $(L_n)_{n\in N}$ of languages in $\mathcal{L}$ and a language $L \in \mathcal{L}$ such that: $L_0 \subsetneq L_1 \ldots \subsetneq \ldots \subsetneq L_n \subsetneq \ldots$ and $L = \bigcup_{n\in N} L_n$ ($L$ is* a limit point *of $\mathcal{L}$).*

*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.* [10]

### 2.3 Limit Points for CDGs With Iterated Subtypes

In [3] 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 [3] concerning iterative subtypes and discuss it later.

**Definition 6.** *Let S, A, B be local dependency names. We define $G'_n, G'_*$ by:*

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

**Theorem 4.** *These constructions yield a limit point as follows [3]:* $L(G'_n) = \{c(b^*a^*)^k \mid k \leq n\}$ *and* $L(G'_*) = c\{b,a\}^*$

**Corollary 2.** *The constructions show the non-learnability from strings for the classes of (rigid) grammars 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

[9] 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 $(s,u) \in R$.

[10] This implies that the class has infinite elasticity.

be clear that, in contrast with the constituent structure grammars and also with the classical 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 [3]). They cannot be considered as 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 [10].

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* $t = [l_m \backslash \cdots l_1 \backslash g / r_1 \cdots / r_n]^P$ *a dependency name* $c$, $i \geq 0$, $0 \leq j \leq m$, *let* $t_c^{(i\backslash,j)} = [l_m \backslash \cdots \backslash l_j \backslash c \cdots \backslash c \backslash l_{j-1} \backslash \cdots l_1 \backslash g / r_1 \cdots / r_n]^P$ *(*$i$ *times) and* $t_c^{(*\backslash,j)} = [l_m \backslash \cdots \backslash l_j \backslash c* \backslash l_{j-1} \backslash \cdots l_1 \backslash g / r_1 \cdots / r_n]^P$. *Respectively, for* $0 \leq k \leq n$ $t_c^{(i/,k)} = [l_m \backslash \cdots l_1 \backslash g / r_1 \cdots / r_{k-1} / c \cdots / c / r_k / \cdots / r_n]^P$ *and* $t_c^{(*/,k)} = [l_m \backslash \cdots l_1 \backslash g / r_1 \cdots / r_{k-1} / c* / r_k / \cdots / r_n]^P$. *Then:*
*1.* $t_c^{(i\backslash,j)} < t_c^{(*\backslash,j)}$ *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* $\tau, \tau'$, *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* $\tau_0$ *and some types* $t', t''$ *such that* $t' < t''$.
*3.* $\lambda < \lambda'$ *for two type assignments* $\lambda$ *and* $\lambda'$, *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.

**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`.

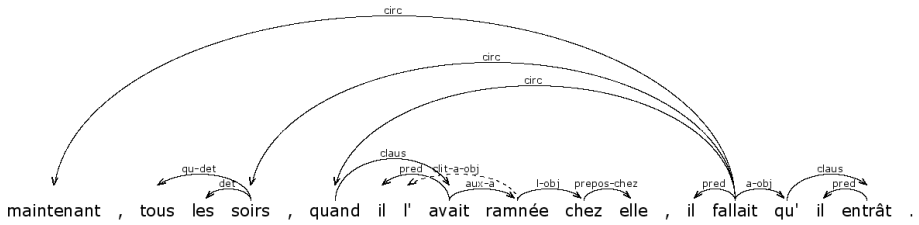
**Definition 8.** *Let* $\mathcal{A}$ *be an inference algorithm for* $\mathcal{CDG}$ *from DS and* $\sigma$ *be a training sequence for a CDG* $G$.
*1.* $\mathcal{A}$ *is* `monotonic` *on* $\sigma$ *if* $\mathcal{A}(\sigma[i]) \preceq \mathcal{A}(\sigma[j])$ *for all* $i \leq j$.
*2.* $\mathcal{A}$ *is* `faithful` *on* $\sigma$ *if* $\Delta(\mathcal{A}(\sigma[i])) \subseteq \Delta(G)$ *for all* $i$.
*3.* $\mathcal{A}$ *is* `expansive` *on* $\sigma$ *if* $\sigma[i] \subseteq \Delta(\mathcal{A}(\sigma[i]))$ *for all* $i$.

**Definition 9.** *Let $G_1$ and $G_2$ be two CDG, $G_1 \equiv_s G_2$ iff $\Delta(G_1) = \Delta(G_2)$.*

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

*Proof.* Indeed, stabilization implies that $\lim_{i\to\infty} \mathcal{A}(\sigma[i]) = \mathcal{A}(\sigma[T])$ 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 it 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




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

**Fig. 4.** Iterated circumstantial dependency

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 10.**
*1.* `Repetition blocks` *(R-blocks) : For $d \in \mathbf{C}$,*

$$LB_d = \{t_1 \backslash \cdots \backslash t_i \mid i > 0, t_1, \ldots, t_i \in \{d\} \cup \{x^* \mid x \in \mathbf{C}\}\}$$

*and symmetrically for $RB_d$.*
*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 $\mathbf{G} = \{< d > \parallel d \in \mathbf{C}\}$. *Moreover, for any* $\alpha, \beta, P$ *and* $d$, $[\alpha \backslash < d > \backslash < d > \backslash \beta]^P$ *and* $[\alpha / < d > / < d > / \beta]^P$ *are not patterns.*
*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* $l_k, \ldots, l_1$ *(in this order), the right projective dependencies or anchors* $r_1, \ldots, r_m$ *(in this order), and discontinuous dependencies* $p_1(d_1), \ldots, p_n(d_n)$*, where* $p_1, \ldots, p_n$ *are polarities and* $d_1, \ldots, d_n \in \mathbf{V}$ *are valency names. Then the* `vicinity` *of* $w$ *in* $D$ *is the type*

$$V(w, D) = [l_1 \backslash \ldots \backslash l_k \backslash h / r_m / \ldots / r_1]^P,$$

*in which* $P$ *is a permutation of* $p_1(d_1), \ldots, p_n(d_n)$ *in a standard lexicographical order, for instance, compatible with the polarity order* $\nwarrow < \searrow < \swarrow < \nearrow$.
*4.* `Superposition` *and* `indexed occurrences` *of R-blocks :*
(*i*) *Let* $\pi$ *be a pattern,* $\beta_1, \ldots, \beta_k$ *be R-blocks and* $< d_1 >, \ldots, < d_k >$ *be gaps. Then* $\pi(< d_1 > \leftarrow \beta_1, \ldots, < d_k > \leftarrow \beta_k)$ *is the expression resulting from* $\pi$ *by the parallel substitution of the R-blocks for the corresponding gaps.*
(*ii*) *Let* $E$ *be a type or a vicinity. Then* $\pi$ *is* `superposable` *on* $E$ *if:*

$$E = \pi(< d_1 > \leftarrow \beta_1, \ldots, < d_k > \leftarrow \beta_k)$$

*for some* $< d_1 >, \ldots, < d_k >, \beta_1, \ldots, \beta_k$.

A vicinity corresponds to the part of a type that is used in a DS. The superposition, in this context, puts together in an R-block a list of dependencies with the same name some of which may be defined by iterative types. For instance, the verb *fallait* in the DS in Fig. 4 has the vicinity $[pred \backslash circ \backslash circ \backslash circ \backslash S / a{-}obj]$. The pattern superposable on this vicinity is $\pi = [< pred > \backslash < circ > \backslash S / < a{-}obj >]$ and the corresponding type is obtained through the following substitution:

$$\pi(< pred > \leftarrow pred, < circ > \leftarrow circ \backslash circ \backslash circ, < a{-}obj > \leftarrow a{-}obj).$$

The vicinity of the participle *ramenée* is $[aux{-}a / l{-}obj]^{\nwarrow clit{-}a{-}obj}$. It is the same as the type: $[aux{-}a / < l{-}obj >]^{\nwarrow clit{-}a{-}obj}(< l{-}obj > \leftarrow l{-}obj)$.

**Proposition 2.** *For every vicinity* $V$ *there is a single pattern* $\pi$ *superposable on* $V$ *and a single decomposition (*R-decomposition*)*

$$V = \pi(< d_1 > \leftarrow \beta_1, \ldots, < d_k > \leftarrow \beta_k)^P$$

**Proposition 3.** *For* $D \in \Delta(G)$ *and an occurrence* $w$ *of a word in* $D$*, let* $V(w, D) = \pi(< d_1 > \leftarrow \beta_1, \ldots, < d_k > \leftarrow \beta_k)^P$ *be the R-decomposition of the vicinity of* $w$ *in* $D$*. Then, for every type* $t \in \lambda(w)$ *which can be used in a proof of* $D$ *for* $w$*, there exists a permutation* $P'$ *of* $P$ *such that* $\pi^{P'}$ *is superposable on* $t$.

**Notation**. Let $G$ be a CDG with lexicon $\lambda$, $w$ be a word and $t$ be a type. Then $G_w^t$ denotes the CDG with lexicon $\lambda \cup \{w \mapsto t\}$.

**Definition 11.** *Let* $K > 1$ *be an integer. We define a CDG* $\mathcal{C}^K(G)$*, the* $K$-`star-generalization` *of* $G$*, by recursively adding for every word* $w$ *and every local dependency name* $d$ *the types*

$$[l_1 \backslash \cdots \backslash l_a \backslash d^* \backslash m_1 \backslash \cdots \backslash m_b \backslash h / r_1 / \cdots / r_c]^P$$

*and*

$$[l_1\backslash\cdots\backslash l_a\backslash m_1\backslash\cdots\backslash m_b\backslash h/r_1/\cdots/r_c]^P$$

*when $w$ has a type assignment $w \mapsto t$, where*

$$t = [l_1\backslash\cdots\backslash l_a\backslash t_1\backslash\cdots\backslash t_k\backslash m_1\backslash\cdots\backslash m_b\backslash h/r_1/\cdots/r_c]^P,$$

*every $t_1, \ldots, t_k$ is either $d$ or some iterated dependency type $x^*$ and among $t_1, \ldots, t_k$ there are* **at least $K$ occurrences of $d$ or at least one occurrence of $d^*$.** *Symmetrically, we also add the corresponding types if $t_1, \ldots, t_k$ appear in the right part of $t$.*

For instance, with $K = 2$, for the type $[a\backslash b^*\backslash a\backslash S/a^*]$, we add $[a\backslash a\backslash S/a^*]$ and $[a\backslash b^*\backslash a\backslash S]$ but also $[a^*\backslash S/a^*]$ and $[S/a^*]$. Recursively, we also add $[a\backslash a\backslash S]$, $[a^*\backslash S]$ and $[S]$. The size of $\mathcal{C}^K(G)$ can be exponential with respect to the size of $G$.

**Definition 12.** *Let $K > 1$ be an integer. CDG $G$ is* **$K$-star-revealing** *if $\mathcal{C}^K(G) \equiv_s G$*

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

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

We see that in a K-star-revealing grammar, one and the same iterated subtype $d^*$ may be used in a type several times. Usually, each occurrence is not in the same block as the local dependency name $d$. Besides this, there should be less than K occurrences of $d$ in a block if there is no occurrence of $d^*$ and this block is separated from other blocks by types that are not iterated.

**Theorem 6.** *The class $\mathcal{CDG}^{K\rightarrow *}$ of $K$-star-revealing CDG is (incrementally) 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, anchor and discontinuous dependencies of every word into a type with repeated local dependencies by introducing iteration for each group of at least $K$ local dependencies with the same name. $\mathbf{TGE}^{(K)}$ is learning $\mathcal{CDG}^{K\rightarrow *}$ due to the following two statements.

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

*Proof.* By definition, the algorithm $\mathbf{TGE}^{(K)}$ is `monotonic` (the lexicon is always extended). 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, $\sigma[i] \subseteq \Delta(\mathbf{TGE}^{(K)}(\sigma[i]))$. To prove that $\mathbf{TGE}^{(K)}$ is `faithful` for $\sigma[i]$ of $\Delta(G) = \Delta(\mathcal{C}^K(G))$, we have to remark that $\mathbf{TGE}^{(K)}(\sigma[i]) \preceq \mathcal{C}^K(G)$.

**Algorithm TGE**$^{(K)}$ (type-generalize-expand):
**Input**: $\sigma[i]$ ($\sigma$ being a training sequence).
**Output**: CDG **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 $\sigma$
**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** $V(w, D) = \pi(<d_1>\leftarrow \beta_1, \ldots, <d_k>\leftarrow \beta_k)^P$
(loop) **for** $j := 1, \ldots, k$
**if** $\beta_j \in LD_d \cup RD_d$ **and** $length(\beta_j) \geq K$
**then** $\gamma_j := d^*$ // `generalization`
**else** $\gamma_j := \beta_j$ **end end**
**let** $t_w := \pi(<d_1>\leftarrow \gamma_1, \ldots, <d_k>\leftarrow \gamma_k)^P$ // `typing`
$\lambda_H(w) := \lambda_H(w) \cup \{t_w\}$; // `expansion`
**end end**

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

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

*Proof.* Because $\mathcal{C}^K(G)$ 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 $\Delta(\mathcal{C}^K(G))$ (and of course in $\sigma$) is also finite. Because the R-blocks are generalized using $*$ by **TGE**$^{(K)}$ when their length is greater or equal to $K$, the number of R-blocks used by **TGE**$^{(K)}$ is finite. Thus the number of generated types is finite and the algorithm certainly stabilizes.

## 4 Learnability from Positive Examples

Below we study the problem of learning CDG from positive examples of structures analogous to the FA-structures used for learning of categorial grammars.

### 4.1 Original Algorithm on Functor-Argument Data

An *FA structure* over an alphabet $\Sigma$ is a binary tree where each leaf is an element of $\Sigma$ 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 [10] it is defined for $AB$ grammars, based on $/_e$ and $\backslash_e$ (binary elimination rules, like the local rules of CDG $\mathbf{L^r}$ and $\mathbf{L^l}$, without potentials) :

$/_e : A \,/\, B, B \Rightarrow A$ and $\backslash_e : B, B \setminus A \Rightarrow A$

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 $\backslash_e$
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 $\sigma_{mgu}$ of this family of types.
6. The algorithm fails if unification fails, otherwise the result is the application of $\sigma_{mgu}$ to the types of $GF(D) : RG(D) = \sigma_{mgu}(GF(D))$.

### 4.2 Functor-Argument Structures for CDG with Iterated Subtypes

**Definition 13.** *Let $\mathcal{D}$ be a dependency structure proof, ending in a type $t$. The* **labelled functor-argument structure** *associated to $\mathcal{D}$, $lfa_{iter}(\mathcal{D})$, is defined by induction on the length of the dependency proof $\mathcal{D}$ considering its last rule :*

- if $\mathcal{D}$ has no rule, it is 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 $\mathbf{L^l}_{[c]}$ 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 $\mathbf{L^l}_{[c]}$ 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 $\mathbf{L^l}$ ;

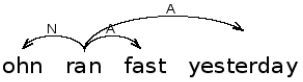
- if the last rule is the one with potentials, $lfa_{iter}(\mathcal{D})$ is taken as the image of the proof above.

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

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

$\mathcal{D}_1$ :

$$\dfrac{N \qquad \dfrac{\dfrac{\dfrac{[N \setminus S \,/\, A^*]\ A}{[N \setminus S \,/\, A^*]}\mathbf{I^r} \qquad A}{[N \setminus S \,/\, A^*]}\mathbf{I^r}}{[N \setminus S]}\mathbf{\Omega^r}}{S}\mathbf{L^l}$$




(dependency structure)

### 4.3 On RG-like Algorithms and Iteration

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

$s_1 = \mathbf{L^l}(John, ran)$
$s_2 = \mathbf{L^l}(John, \mathbf{L^r}(ran, fast))$
$s_3 = \mathbf{L^l}(John, \mathbf{L^r}(\mathbf{L^r}(ran, fast), yesterday)$
$s_4 = \mathbf{L^l}(John, \mathbf{L^r}(\mathbf{L^r}(\mathbf{L^r}(ran, fast), yesterday), nearby)$

An RG-like algorithm could compute the following assignments and grammar from $\{s_1, s_2, s_3\}$: $\mathbf{L^l}(John : X_1, ran : X_1 \setminus S) : S$

$\mathbf{L^l}(John : X'_1, \mathbf{L^r}(ran : X'_1 \setminus S \,/\, X_2, fast : X_2) : X'_1 \setminus S) : S$

$\mathbf{L^l}(John : X"_1, \mathbf{L^r}(\ \mathbf{L^r}(ran : X"_1 \setminus S \,/\, X"_2 \,/\, X'_2, fast : X'_2) : X"_1 \setminus S \,/\, X"_2,$
$yesterday : X"_2) : X"_1 \setminus S) : S$

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

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 dependency 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 [3], involving iterated dependency types. In fact, the functor-argument structures are all flat structures, with only / operators.

$C'_0 = S$
$C'_{n+1} = C'_n \,/\, A^* \,/\, B^*$

$G'_0 = \{a \mapsto A, b \mapsto B, c \mapsto C'_0\}$
$G'_n = \{a \mapsto A, b \mapsto B, c \mapsto [C'_n]\}$
$G'_* = \{a \mapsto A, b \mapsto A, c \mapsto [S \,/\, A^*]\}$

Positive structured examples are then of the form :

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

**Definition 14.** *We define* $flat_{\mathbf{L^r}}$ *and* $flat_{\mathbf{L^r}_{[A]}}$ *on words by :* $flat_{\mathbf{L^r}}(x1) = x1$ $= flat_{\mathbf{L^r}_{[A]}}(x1)$ *for words of length 1, and* $flat_{\mathbf{L^r}}(x1.w1) = \mathbf{L^r}(x, flat_{\mathbf{L^r}}(w1))$ *;* $flat_{\mathbf{L^r}_{[A]}}(x1.w1) = \mathbf{L^r}_{[A]}(x, flat_{\mathbf{L^r}_{[A]}}(w1))$ *; we extend the notation* $flat_{\mathbf{L^r}}$ *and* $flat_{\mathbf{L^r}_{[A]}}$ *to sets of words (as the set of word images).*

Let $FL(G)$ denote the language of functor-arguments structures of $G$.

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

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

**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 :

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

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 $G$.

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

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

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 :

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

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 $a$, as in :

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

**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. This fact also extends to $k$-rigid CDG, not only to rigid (1-rigid) CDG.

## 5 Conclusion

In this paper, we propose a new model of incremental learning of categorial dependency grammars with unlimited iterated types from input dependency structures without marked iteration. The model reflects the real situation of deterministic inference of a dependency grammar from a dependency treebank. The learnability sufficient condition of $K$-star-revealing we use, is widely accepted in traditional linguistics for small $K$, which makes this model interesting for practical purposes. As shows our study, the more traditional unification based

learning from function-argument structures fails even for rigid categorial dependency grammars with unlimited iterated types.

On the other hand, in this paper, the $K$-star-revealing condition is defined in "semantic" terms. It is an interesting question, whether one can find a simple syntactic formulation.

## References


1. Angluin, D.: Inductive inference of formal languages from positive data. Information and Control 45, 117–135 (1980)
2. Béchet, D., Dikovsky, 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., Dikovsky, 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)
4. Buszkowski, W., Penn, G.: Categorial grammars determined from linguistic data by unification. Studia Logica 49, 431–454 (1990)
5. Dekhtyar, M., Dikovsky, A.: Categorial dependency grammars. In: Proc. of Intern. Conf. on Categorial Grammars. pp. 76–91. Montpellier (2004)
6. Dekhtyar, M., Dikovsky, A.: Generalized categorial dependency grammars. In: Trakhtenbrot/Festschrift, pp. 230–255. LNCS 4800, Springer (2008)
7. Dikovsky, A.: Dependencies as categories. In: "Recent Advances in Dependency Grammars". COLING'04 Workshop. pp. 90–97 (2004)
8. Gold, E.M.: Language identification in the limit. Information and control 10, 447–474 (1967)
9. 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)
10. Kanazawa, M.: Learnable classes of categorial grammars. Studies in Logic, Language and Information, FoLLI & CSLI (1998)
11. Mel'čuk, I.: Dependency Syntax. SUNY Press, Albany, NY (1988)
12. 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)
13. Shanker, V.K., Weir, D.J.: The equivalence of four extensions of context-free grammars. Mathematical Systems Theory 27, 511–545 (1994)
14. Shinohara, T.: Inductive inference of monotonic formal systems from positive data. New Generation Computing 8(4), 371–384 (1991)
15. 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)