Conversions between D and $MCFG$

August 2, 2012

Abstract

We study the Displacement Calculus D and Multiple Context Free Grammars. We show that a restricted fragment of $MCFG$ can be recognized by higher-order constructions in D. Furthermore, we show that a restricted fragment of D, namely the one without product unit in negative positions, and with only $(A \uparrow_k B) \downarrow_k C$ constructions suffices to define the class of well-nested Multiple Context Free Languages ($MCFL_{wn}$). We make the connection between these two constructions more explicit by showing how to construct a higher-dimensional but first-order grammar from such a restricted two-dimensional grammar. The construction in [5] shows that the resulting grammar is indeed a well-nested one.

1 Introduction

In [5], a construction was given that shows that $L(D^1) = MCFL_{wn}$, i.e. the class of languages generated by first-order Displacement grammars coincides with the class of languages generated by well-nested Multiple Context Free Grammar. Moreover, the construction is *dimension-specific*, i.e. given a D^1 grammar of sort k, there is a (weakly) equivalent $MCFG$ of dimension $k + 1$. This paper complements that result by showing that by allowing specific higher-order constructions in $1D^1$ (sort 1, first order) grammars, one can generate any $MCFL_{wn}$. Furthermore, a construction is given that transforms this kind of D grammars to first-order D^1 grammars. The construction in [5] then verifies well-nestedness, so we get that the class of languages of these formalisms coincide.

2 Definitions

We define several fragments of the *Displacement Calculus*, as well as *Multiple Context Free Grammars.*

2.1 MCFG

Definition 1 (MCFG) *A Multiple Context Free Grammar is a 6-tuple* (N, T, F, P, S, dim) *such that:*

- N *is a finite set of non-terminal symbols, and dim assigns a dimension to every non-terminal,*
- T *is a finite set of terminal symbols,*
- F *is a finite set of mcf-functions,*
- P *is a finite set of production rules of the form* $A_0 \to f[A_1, ..., A_k]$ *with* $k \geq 0$ $f : (T^*)^{dim(A_1)} \times ... \times (T^*)^{dim(A_k)} \to (T^*)^{dim(A_0)}$ *and* $f \in F$.
- $S \in N$ *is a distinguished start symbol such that* $dim(S) = 1$.

Definition 2 *f is a mcf-function if:*

- $f(\overrightarrow{x_1}, ..., \overrightarrow{x_k}) = \alpha_1\beta_1...\alpha_n\beta_n\alpha_{n+1}$ *where* $\alpha_i \in T^*$ *and* β_j *a variable from some* $\overrightarrow{x_m}$.
- *Each variable* x_{ij} *from some vector* $\overrightarrow{x_m}$ *occurs at most (or exactly) once in the right hand side (**linearity**)*

Definition 3 *The dimension of a MCFG G is given by the maximal dimension of the non-terminals, i.e.* $max(dim(N))$. *We call a MCFG of dimension k a k-MCFG.*

Definition 4 (Well-nestedness) *An mcf-function f is well-nested if it satisfies the following condition:*

- *if* $i \neq i'$, $j \leq q_i$, $j' \leq q_{i'}$ *then* $\alpha_1\beta_1, ...\alpha_n\beta_n, \alpha_{n+1}$ *where* $\alpha_i \notin (T \cup Var) * x_{ij}(T \cup Var) * x_{i'j'}(T \cup Var) * x_{ij+1}(T \cup Var) * x_{i'j'+1}$

We call a MCFG G well-nested if all the mcf-functions are well-nested.

Definition 5 (Lexicalized $MCFG$) *We call a $MCFG_{(wn)}$ G* lexicalized*, denoted $lMCFG_{(wn)}$, if for every production rule p it holds that there is exactly one $a \in T$ such that for the corresponding mcf-function it holds that $f(\overrightarrow{x_1}, ..., \overrightarrow{x_k}) = a\beta_1...\beta_n$, i.e. each rule produces only one terminal symbol, called the* anchor*.*

Definition 6 (MCFL) *Let $G = (N, T, F, P, S)$ be a $MCFG$.*

- *For every $A \in N$:*
  1. *For every $(A \to f[]) \in P : f[] \in yield(A)$,*
  2. *For every $(A \to f[A_1, ..., A_k]) \in P(k \geq 1)$ and all tuples $\tau_1 \in yield(A_1)...\tau_k \in yield(A_k) : f[\tau_1, ..., \tau_k] \in yield(A)$.*
  3. *Nothing else is in $yield(A)$.*
- *The string language of G is $L(G) = \{w | \langle w \rangle \in yield(S)\}$.*
- *We denote by k-$MCFL$ the class of languages generated by the set of $MCFGs$ of dimension at most k. We denote by $MCFL$ the class of languages generated by an arbitrary $MCFG$. We use a* wn *subscript to denote the well-nested variants.*

The following facts about these different language classes are known:

1. For each k, k-$MCFL_{(wn)} \subset (k+1)$-$MCFL_{(wn)}$
2. For each k, k-$MCFL_{wn} \subset k$-$MCFL$.
3. $MCFL_{wn} \subset MCFL$.

2.2 Displacement Grammar

Here we define Displacement Grammar and several fragments. General Displacement Calculus is obtained by extending the (associative) Lambek Calculus **L** with infinitely many residual connective families $(\uparrow_k, \downarrow_k, \odot_k)$ that, in effect, enable *wrapping* and *extraction* from specific points in a string. Therefore, the string algebra acted upon contains a special element, called a *separator*. More surprisingly, apart from the separator the string algebra is just a free monoid, as the arrow families are interpreted by means of *defined operations*.

Definition 7 (Types and order) *The types of the Displacement Calculus are inductively defined as follows:*

$$T := p \mid T \bullet T \mid T \backslash T \mid T / T \mid T \odot_k T \mid T \uparrow_k T \mid T \downarrow_k T$$
$$\text{where } k \in NAT$$

Definition 8 (Order) *The order of a type is defined by the following homomorphism:*

$$ord(p) = 0$$
$$ord(A \bullet B) = max(ord(A), ord(B))$$
$$ord(A \backslash B) = ord(B / A) = max(ord(A) + 1, ord(B))$$
$$ord(A \odot_k B) = max(ord(A), ord(B))$$
$$ord(A \downarrow B) = ord(B \uparrow A) = max(ord(A) + 1, ord(B))$$

We place an important note here: If we look at types as functions, one may specify the input and output types of that function. A semantic type map does this, and indeed, higher-order types are mapped to higher-order lambda terms/functions. We want to state here, however, as we are going to be dealing with higher-order $\uparrow, \downarrow$ types, what we mean by input and output position in these cases. So, given the type $(A \uparrow B) \downarrow C$ or $C \uparrow (B \downarrow A)$ respectively, a semantic type map would give us $(B \to A) \to C$ in both cases. In other words, C is in the output position in these types, whereas $A \uparrow B$ $(B \downarrow A)$ is in an input position.

Definition 9 (Hypersequent Calculus) *The proof theory for the full General Displacement Calculus is as follows:*

$$\frac{}{\overrightarrow{A} \to A}\ id \qquad \frac{\Gamma \Rightarrow A \quad \Delta\langle \overrightarrow{A} \rangle \Rightarrow B}{\Delta\langle \Gamma \rangle \Rightarrow B}\ Cut$$

$$\frac{\Gamma \Rightarrow A \quad \Delta\langle \overrightarrow{B} \rangle \Rightarrow C}{\Delta\langle \Gamma, \overrightarrow{A \backslash B} \rangle \Rightarrow C}\ \backslash L \qquad \frac{\overrightarrow{A}, \Gamma \Rightarrow B}{\Gamma \Rightarrow A \backslash B}\ \backslash R$$

$$\frac{\Gamma \Rightarrow A \quad \Delta\langle \overrightarrow{B} \rangle \Rightarrow C}{\Delta\langle \overrightarrow{B / A}, \Gamma \rangle \Rightarrow C}\ / L \qquad \frac{\Gamma, \overrightarrow{A} \Rightarrow B}{\Gamma \Rightarrow B / A}\ / R$$

$$\frac{\Delta\langle \overrightarrow{A}, \overrightarrow{B} \rangle \Rightarrow C}{\Delta\langle \overrightarrow{A \bullet B} \rangle \Rightarrow C}\ \bullet L \qquad \frac{\Gamma \Rightarrow A \quad \Delta \Rightarrow B}{\Gamma, \Delta \Rightarrow A \bullet B}\ \bullet R$$

$$\frac{\Delta\langle \Lambda \rangle \Rightarrow A}{\Delta\langle \overrightarrow{I} \rangle \Rightarrow A}\ IL \qquad \frac{}{\Lambda \Rightarrow I}\ IR$$

$$\frac{\Gamma \Rightarrow A \quad \Delta\langle \overrightarrow{B} \rangle \Rightarrow C}{\Delta\langle \Gamma|_k \overrightarrow{A \downarrow_k B} \rangle \Rightarrow C}\ \downarrow_k L \qquad \frac{\overrightarrow{A}|_k \Gamma \Rightarrow B}{\Gamma \Rightarrow A \downarrow_k B}\ \downarrow_k R$$

$$\frac{\Gamma \Rightarrow A \quad \Delta\langle \overrightarrow{B} \rangle \Rightarrow C}{\Delta\langle \overrightarrow{B \uparrow_k A}|_k \Gamma \rangle \Rightarrow C}\ \uparrow_k L \qquad \frac{\Gamma|_k \overrightarrow{A} \Rightarrow B}{\Gamma \Rightarrow B \uparrow_k A}\ / R$$

$$\frac{\Delta\langle \overrightarrow{A}|_k \overrightarrow{B} \rangle \Rightarrow C}{\Delta\langle \overrightarrow{A \odot_k B} \rangle \Rightarrow C}\ \odot_k L \qquad \frac{\Gamma \Rightarrow A \quad \Delta \Rightarrow B}{\Gamma|_k \Delta \Rightarrow A \odot_k B}\ \odot_k R$$

$$\frac{\Delta\langle [] \rangle \Rightarrow A}{\Delta\langle \overrightarrow{J} \rangle \Rightarrow A}\ JL \qquad \frac{}{[] \Rightarrow J}\ JR$$

where $k \in NAT$.

Fact 1 *The full General Displacement Calculus enjoys Cut-elimination.*

Definition 10 (Weaker fragments) *We define the following fragments of the full Hypersequent Calculus :*

1. *We obtain* first-order *Displacement Calculus, denoted* $\mathbf{D^1}$*, by dropping the* $/R, \backslash R, \bullet L, \uparrow_k R, \downarrow_k R, \odot_k L$ *rules.*
2. *We obtain* product-unit-free *Displacement Calculus, denoted* $\mathbf{D_J}$*, by dropping the* IL *rule.*

3. *We obtain* full General *Displacement Calculus, denoted* $\mathbf{D_{IJ}}$*, by doing nothing.*

The reader may verify that restricting the hypersequent calculus to $\mathbf{D^1}$, one indeed has no use for higher-order types.

Definition 11 *The dimension of a D grammar G is given by the maximal k such that* $\uparrow_k, \downarrow_k$ *or* $\odot_k$ *is allowed to appear in proofs. We call a* $\mathbf{D}$ *grammar of dimension k a k-*$\mathbf{D}$ *grammar.*

Fact 2 $L(\mathbf{D^1}) \subseteq L(\mathbf{D_J}) \subseteq L(\mathbf{D_{IJ}})$.

2.3 Relations between D and $MCFG$

As noted earlier, the goal of this paper is to relate restrictions on $MCFG$s to restrictions on D grammars. One result was already given in [5], where it is shown that for any k, $L(k\text{-}\mathbf{D^1}) = (k+1)\text{-}MCFL_{wn}$. The result is nice because the dimension as it is represented in well-nested $MCFG$ precisely matches that of first-order General Displacement Calculus, i.e. the number of strings acted upon by a predicate of a $MCFG_{wn}$ equals the number of strings that the corresponding type in a $\mathbf{D^1}$ grammar holds. A major disadvantage is that, although dealing with only first-order types should be computationally attractive, for each dimension one requires a new family of connectives.

The main result of this paper is the rather surprising fact that $L(\mathbf{D^1}) = L(1 - \mathbf{D_J})$. We prove this by giving a construction from an arbitrary well-nested $MCFG$ to a 1-$\mathbf{D_J}$ grammar, which exploits the use of higher-order types. By the result of [5] we then have inclusion from left to right. The converse is shown by showing how an arbitrary 1-$\mathbf{D_J}$ grammar can be converted into a $L(\mathbf{D^1})$ grammar, where the use of higher-order constructions determines the dimension of the latter grammar. We thus get $L(\mathbf{D^1}) = L(1-\mathbf{D_J})$, although we lose the dimension-specific equality. Thus, we get that only one extra family of connectives on top of the associative Lambek Calculus $\mathbf{L}$ is enough to describe well-nested $MCFL$s.

On top of this result, we show an algorithm that construct from a specific class of *ill-nested* $MCFG$s weakly equivalent $L(1 - \mathbf{D_J})$ grammars. This gives a partial result on what kind of ill-nestedness of $MCFG$s describes languages that are still well-nested languages (although maybe of higher dimension).

Furthermore, we will show a construction from $L(\mathbf{D_{IJ}}$ to $MCFG$s, i.e. that includes higher-order types involving the empty string type I. This gives rise to the conjecture that $L(\mathbf{D_{IJ}}) = MCFL$.

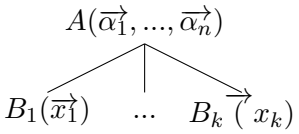
3 Preliminary: two different modes of presentation

In this section, we briefly outline the intuition behind the relation between $MCFG$ derivations and D grammar derivations. It should become clear then, that specific fragments of these two constructs admit a direct conversion.

3.1 $MCFG$ derivations as D proofs

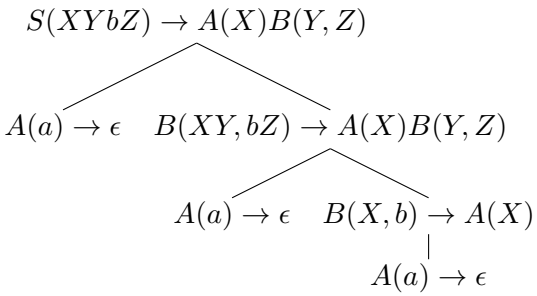
Although we speak of $MCFG$ in this paper, another, equivalent formalism was defined in [1] called Simple Range Concatenation Grammar ($sRCG$). Although the definition of an $MCFG$ is more appropriate to work with here, the presentation of rules in $sRCG$ format is more simple. Given an $MCFG$ rule $A \rightarrow f[B_1, ..., B_k]$ and an mcf-function $f(\overrightarrow{x_1}, ..., \overrightarrow{x_k}) = \alpha_1\beta_1, ...\alpha_n\beta_n, \alpha_{n+1}$, the corresponding $sRCG$ rule is $A(\alpha_1\beta_1, ...\alpha_n\beta_n, \alpha_{n+1}) \rightarrow B_1(\overrightarrow{x_1})...B_k(\overrightarrow{x_k})$. Just for the sake of clarity, we choose to visualize $MCFG$ production rules in $sRCG$ format, although the proofs use the $MCFG$ definitions. Henceforth, when we refer to production rules, we mean $MCFG$ rules, but will write $sRCG$ rules.

Production rules of an $MCFG$ may be seen as *derivation schemes*. In fact, they are linear Horn clauses. Such a rule can be represented by a tree in the following sense: given a rule $A(\overrightarrow{\alpha_1}, ..., \overrightarrow{\alpha_n}) \rightarrow B_1(\overrightarrow{x_1})...B_k(\overrightarrow{x_k})$, we draw its tree representation like:

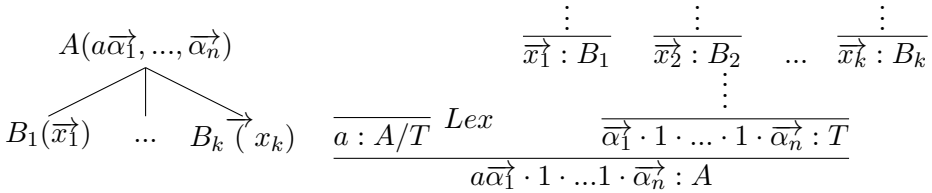


A derivation can be unambiguously represented as a general tree, with the nodes labelled with production rules, for each non-terminal associated with the rule, an edge runs to a production rule that starts with that non-

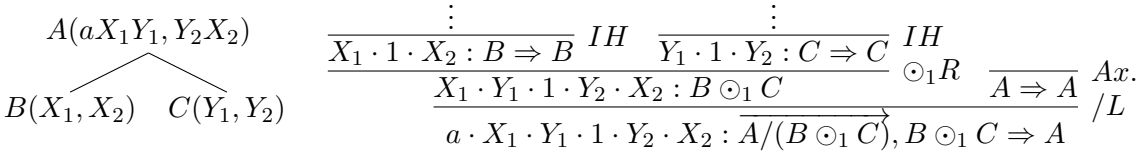
terminal, for instance:

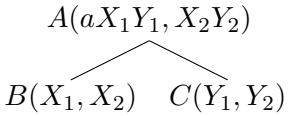


The general intuition behind constructing a D lexicon that mimics an $lMCFG$ production rule is that one turns the tree representation of the rule upside down, and constructs a type assignment for the anchor such that, given the leaves of the tree as derivations and the anchor as lexical item, one can derive the root of the tree as the conclusion of a D derivation and nothing else. So, we get the following picture:



Once the types for each anchor are constructed, it is clear that for every possible $lMCFG$ derivation there is a corresponding D proof and vice-versa. So the only tricky thing here is when exactly a production rule is such that one can directly read off an anchor type. In [5] it is shown that a direct construction is possible in the case of well-nested production rules (corresponding to first-order anchor types). However, for ill-nested rules a direct interpretation is absent, as one can see in the following typical examples:





In our proofs, we will assume the correspondence between well-nested production rules and first-order D types.

4 Main result: $L(\mathbf{D^1}) = L(1\text{-}\mathbf{D_J})$

In this section, our main result is proven, namely that first-order General Displacement Calculus is equal in terms of generative capacity to higher-order *two-dimensional* Displacement Calculus. Given that $L(\mathbf{D^1}) = MCFL_{wn}$, we show respectively that $MCFL_{wn} \subseteq L(1\text{-}\mathbf{D_J})$ and $L(1\text{-}\mathbf{D_J}) \subseteq L(\mathbf{D^1})$. What follows is the main result that $L(\mathbf{D^1}) = L(1\text{-}\mathbf{D_J})$, in other words, there is a trade-off between dimension and type order in Displacement Calculus.

4.1 From left to right: $MCFL_{wn} \subseteq L(1\text{-}\mathbf{D_J})$

We show a direct construction of a 1-$\mathbf{D_J}$ grammar, given an arbitrary well-nested $MCFG$. The construction follows [5] except that the tuple delimiters in the $MCFG_{wn}$ are interpreted prosodically rather than using a separator. Each intercalation is represented then using these 'variables' in a higher-order position. Consider for instance the production rule $A(aXYZ) \rightarrow B(X,Z)C(Y)$. We add a lexical entry $s^1_B : S^1_B$ for the first tuple delimiter of the B predicate, and add a lexical entry $a : A/((B \uparrow_1 S^1_B) \odot_1 C)$. By induction then, we may assume that any B tuple X, Y is represented in the resulting grammar as $X \cdot S^1_B \cdot Y$ and so (given $Z : C$) we may derive $aXYZ : A$.

Lemma 1 *For any $lMCFG_{wn}$ G, there is a 1-D_J grammar G' such that $L(G) = L(G')$.*

CONSTRUCTION: Let $G = (N, T, F, P, S, dim)$ be a lexicalized well-nested $MCFG$. For each $A \in N$, add type assignments $s^i_a : S^i_A$ for $1 \leq i \leq dim(A)$. Following the construction of [5], we consider each rule separately, and add type assignments for the anchors of the rules. Given a rule $A(a\overrightarrow{\alpha_1}, ..., \overrightarrow{\alpha_n}) \rightarrow B_1(\overrightarrow{x_1})...B_k(\overrightarrow{x_k})$, we add a type assignment $a : A/T$ where T is determined by the way $\overrightarrow{x_1}...\overrightarrow{x_k}$ are combined to obtain $\overrightarrow{\alpha_1}, ..., \overrightarrow{\alpha_n}$. As G is well-nested, $\overrightarrow{\alpha_1}, ..., \overrightarrow{\alpha_n}$ is a concatenation of subparts and wrap units (separators), combined with wrapping of subparts and product units. As each rule is by definition linear, each $\overrightarrow{x_i}$ is used exactly once. Hence, there is a way of describing such a combination using a first-order type, say T'. We transform

this T' into T by the following means:

1. Replace every $D \odot_k E$ by $(D \uparrow_1 S^k_D) \odot_1 E$,
2. Replace every $D \bullet J^k$ by $D \bullet S^k_A$ (where J^k means J but as the k-th occurrence in the final A tuple.

CORRECTNESS: The construction of [5] shows that the translation into first-order D types mimics the production rules of G exactly, so we only need to show that the additional transformation gives us (1) a two-dimensional grammar and (2) the same string language.

(1) First of all, there are no occurrences of arrow connectives, so the only higher-dimensional connectives are occurrences of $\odot$ with an index greater than 1. These are all handled by the first part of the transformation, and reduced to the $\uparrow_1, \odot_1$ connectives. Furthermore, all the newly introduced separators are replaced by prosodic variables. Therefore, the only moment in a derivation when a string is split is when producing a $D \uparrow_1 S^k_E$ tuple.

(2) Given an occurrence of some $D \odot_k E$ in T', a derivation in G will look like this:

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ X \cdot 1^k \cdot Y : \Gamma \Rightarrow D & Y : \Delta \Rightarrow E\end{array}}{XYZ : \Gamma|_k\Delta \Rightarrow D \odot_k E}\ \odot_k$$

The transformation then allows the following derivation in G':

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ X' \cdot s^k_D \cdot Y' : \Gamma'|_1 S^k_D \Rightarrow D\end{array}}{X' \cdot 1 \cdot Y' : \Gamma' \Rightarrow D \uparrow_1 S^k_D} \quad \begin{array}{c}\vdots \\ Y' : \Delta' \Rightarrow E\end{array}}{X'Y'Z' : \Gamma'|\Delta' \Rightarrow (D \uparrow_1 S^k_D) \odot_k E}\ \odot_k$$

The second part of the transformation then guarantees that instead of separators, there are actually prosodic variables available, hence we conclude that the transformation indeed preserves string language.

Example. Consider the following grammar for $\{a^n b^n c^n d^n e^n f^n | n \geq 1\}$:

$$S(XYZ) \to A(X, Y, Z)$$
$$A(aXP_1, P_2YP_3, P_4ZP_5) \to A(X, Y, Z)B(P_1)C(P_2)D(P_3)E(P_4)F(P_5)$$
$$A(aP_1, P_2P_3, P_4P_5) \to B(P_1)C(P_2)D(P_3)E(P_4)F(P_5)$$
$$B(b).$$
$$C(c).$$
$$D(d).$$
$$E(e).$$

The construction of [5] gives the following grammar:

$$S := (A \odot_2 I) \odot_1 I$$
$$a : A/(((A \odot_1 (B \bullet J \bullet C)) \odot_2 (D \bullet J \bullet E)) \bullet F)$$
$$a : A/(B \bullet J \bullet C \bullet D \bullet J \bullet E \bullet F)$$
$$b : B$$
$$c : C$$
$$d : D$$
$$e : E$$
$$f : F$$

The additional transformation gets rid of the higher-index $\odot$ connectives and the J types, resulting in the following grammar:

$$S := (((A \uparrow_1 S_A^2) \odot_1 I) \uparrow_1 S_A^1) \odot_1 I$$
$$a : A/(((((A \uparrow_1 S_A^1) \odot_1 (B \bullet S_A^1 \bullet C)) \uparrow S_A^2) \odot_1 (D \bullet S_A^2 \bullet E)) \bullet F)$$
$$a : A/(B \bullet S_A^1 \bullet C \bullet D \bullet S_A^2 \bullet E \bullet F)$$
$$b : B$$
$$c : C$$
$$d : D$$
$$e : E$$
$$f : F$$
$$s_A^1 : S_A^1$$
$$s_A^2 : S_A^2$$

The reader may try out some derivations to verify that the latter two grammars are indeed weakly equivalent. ■

4.2 From right to left: $L(1\text{-}\mathbf{D_J}) \subseteq L(\mathbf{D^1})$

We show a direct construction of a first-order D grammar from an arbitrary 1-$\mathbf{D_J}$ grammar. The intuition is as follows: for any $\uparrow, \downarrow$ introduction, the natural deduction rule is one of the following:

$$\frac{\begin{array}{c}\alpha : A \\ \vdots \\ \alpha|\gamma : B\end{array}}{\gamma : A \downarrow B}\ I\downarrow \qquad\qquad \frac{\begin{array}{c}\alpha : A \\ \vdots \\ \gamma|\alpha : B\end{array}}{\gamma : B \uparrow A}\ I\uparrow$$

meaning that, as the grammars are constructive and we do not allow the product unit in the antecedent position ($\alpha \neq 0$), the $\alpha : A$ has to be constructed first by means of some other legitimate derivation. We show that it is possible to construct a first-order grammar that effectively allows the derivations of γ but without using the mentioned introduction rules. The introduction rules are only used when the lexicon contains $B \uparrow A$ ($A \downarrow B$) in a negative position. Thinking of $B \uparrow A$ as an expression of type B but at the place of the separator lacking an expression of type A and $A \downarrow B$ as an expression that, wrapped in an A tuple would produce an expression of type B, it is clear that $A \uparrow J$ and $J \downarrow A$ are trivial types for any type A. Indeed, $A \uparrow J \equiv A \equiv J \downarrow A$ (both in the hypersequent calculus as in the prosodic interpretation). We exploit this fact in the construction.

To make the construction as explicit as possible, we elaborate a bit on types as functions here. Given a first-order lexical item $x : X$, we can have a direct interpretation of types as simply typed (syntactic) functions, by assigning a syntactic map, together with a semantic type map, which would assign for example to $b : B/A$, the function $f(X) = bX : A \multimap B$ and to $c : A \downarrow C$ the function $g(\langle X, Y\rangle) = XcY : A \multimap C$, and we see in the construction of [5] that this interpretation of types exactly covers all the well-nested mcf-functions. Also, every D^1 derivation essentially is a recipe for a repeated application of these functions to one another, yielding finally some string. So the key point of our construction is that, instead of deriving (by means of arrow introduction rules) types as $A \uparrow B$ to be used as *input* to some higher-order type, for example $(A \uparrow B)\backslash C$, we add a *lifted* type A' (lifted in the sense that as A is one-dimensional, A' is two-dimensional) and add lexical items that allow exactly those derivations of A' (which will be a D^1 derivation) such that the syntactic function (or yield) is the same as a derivation of $A \uparrow B$. As we can track all the places in derivations where the specific expression of type B was inserted that is later extracted, we allow a derivation in which there is a separator inserted at one of these places.

Consider, for example, the following lexical items:

$$e : (A \uparrow B) \downarrow E$$
$$a : A/B$$
$$b : B/A$$
$$c : A$$

we could derive for instance $c : A$, $abc : A$, $ababc : A$, etc., and hence we could derive $a1 : A \uparrow B$, $aba1 : A \uparrow B$, etc.
The trick is to replace the $A \uparrow B$ in the first lexical assignment (as it causes a higher-order type) with a lifted type A', and then add types that allow to derive $a1 : A'$, $aba1 : A'$ etc. So, we replace the first lexical assignment by $e : A' \downarrow E$, and we search through all types that output an expression of type A, namely the second and last entry. We 'traverse' these entries from output to input, replacing each type by a fresh marked type and repeating for all entries that output the replaced type, until we find an input of type B, which we replace by a J, so that the place where an expression of type B would be inserted to finally derive an expression of type A will now be filled by a separator, but consequently we may only derive an expression of type A' (this is to guarantee that any other derivation that use an ordinary expression of type A does not get disturbed by our surgery). However, we need to make sure that after inserting one separator, we can still derive longer strings, i.e. at this point of the surgery we might be able to derive $a1 : A'$, but not $aba1 : A'$. To do this, traverse all types that have A as input, traversing them from this type, and replacing all types by their marked doubles. For the example, this will result in the following lexicon:

$$e : A' \downarrow E$$
$$a : A/B$$
$$b : B/A$$
$$c : A$$

$$a : A'/J$$

$$a : A'/B'$$
$$b : B'/A'$$

Now we can derive $a1 : A'$ using just the fifth lexical entry, and $aba1 : A'$ by additionally using the last two entries.

Lemma 2 *For any* 1-$\mathbf{D_J}$ *grammar* G*, there is a* D^1 *grammar* G' *such that* $L(G) = L(G')$.

CONSTRUCTION: For each lexical assignment, do the following:

1. For each higher-order type $(A \uparrow B) \downarrow C$ (or $(A \uparrow B)\backslash C$ or $C \uparrow (B \downarrow A)$ or $C/(B \downarrow A)$) in an output position, replace the $A \uparrow B$ ($B \downarrow A$) in the input position by A' (A').

2. For each A', consider all lexical assignments that have A as their final output type. Traverse these items from output to input, and for each input type, replace it by a marked double, and traverse all items that output the original type. Do this until a B input is encountered, and replace this by a J and stop. Now, for each entry that has A as an input type, traverse the type from input to output, starting at this A, replacing every type by its marked double.

3. Replace each $\uparrow, \downarrow$ connective with the corresponding $\uparrow_k, \downarrow_k$ connectives (corresponding in the sense that, as A' is of higher-dimension than A, $\uparrow_k, \downarrow_k$ connectives might need to be replaced by, say, $\uparrow_{k+1}, \downarrow_{k+1}$).

4. Delete useless entries.

CORRECTNESS: We have to show that G', the grammar that results from the described transformation, is language-preserving and that it is first-order. Clearly, G' is first-order, as the only connectives that give rise to higher order types are $\uparrow, \downarrow$, but precisely these cases are handled by step 1 of the construction. Now we need to show that $L(G) = L(G')$. Let $\mathfrak{D}$ be a derivation containing an introduction of $\uparrow$, so it contains a subderivation of the form

$$\dfrac{\dfrac{}{c : (B \uparrow A) \downarrow C}\ Lex \quad \begin{array}{c}\alpha : A \\ \vdots \\ \gamma|\alpha : B\end{array}}{\gamma : B \uparrow A}\ I\uparrow}{\gamma|c : C}$$

By the transformation, we cannot derive $c : (B \uparrow A) \downarrow C$ anymore, but only $c : B' \downarrow C$. However, part 2 of the transformation enables us to derive $\gamma' : B'$, because the place where α was inserted in γ in order to derive B,

can now be a separator by using the alternative lexical item, given by part 2 of the transformation.
For the converse, let $\mathfrak{D}$ be a derivation of G'. Either it uses only original types from G, in which case it is derivable in G. The other case is when it uses new types, given by the transformation. As all the atomic types in the new types are solely different from those in G, nothing of the derivation that uses these types can interfere with lexical items in G. So, it suffices to show that these types mimic exactly the relevant introduction rules for G, in which case any derivation from these types can be derived in G. It is clear, by the choice of fresh types, that such derivations are uniquely determined, and that they thus only derive expressions of the form $\gamma' : B'$, for which it holds that one can derive $\gamma' : B \uparrow A$ in G. As the type $B' \downarrow C$ is unique in G, and comes from some type $(B \uparrow A) \downarrow C$ in G, we know that we can derive $\gamma'|a : C$ in G', but also that we can derive $\gamma'|a : C$ in G.

Example. Consider the following 1-D_J grammar for $\{www|w \in \{a, b\}^*\}$:

$$S := (((P \uparrow X) \odot I) \uparrow Y) \odot I$$
$$a : A$$
$$b : B$$
$$x : X$$
$$y : Y$$
$$a : (((P/A)/Y)/A)/X$$
$$a : ((P \uparrow X) \downarrow T_1)/X$$
$$a : ((T_1 \uparrow Y) \downarrow (P/A))/Y$$
$$b : (((P/B)/Y)/B)/X$$
$$b : ((P \uparrow X) \downarrow T_3)/X$$
$$b : ((T_3 \uparrow Y) \downarrow (P/B))/Y$$

We follow the construction. By step 1, we replace the start symbol, the last two assignments for a and the last two assignments for b to get

$$S := (P' \odot I) \odot I$$
$$a : (P' \downarrow T_1)/X$$
$$a : (T_1' \downarrow (P/A))/Y$$
$$b : (P' \downarrow T_3)/X$$
$$b : (T_3' \downarrow (P/B))/Y$$

Step 2 instructs us to do a traversal from output to input for P, T_1, T_3 with 'parameters' X, Y, Y respectively, giving us new types:

$$a : (((P'/A)/J)/A)/J$$
$$a : (P' \downarrow_1 T'_1)/J$$
$$a : (T'_1 \downarrow_2 (P'/A))/J$$
$$b : (((P'/B)/J)/B)/J$$
$$b : (P' \downarrow_1 T'_3)/J$$
$$b : (T'_3 \downarrow_2 (P'/B))/J$$

Still following step 2, we need to do an upwards traversal, to ensure that we can derive the rest of the strings, but in this example, this is not relevant, so we skip it.
By step 3, we need to replace the $\uparrow, \downarrow$ connectives by corresponding $\uparrow_k, \downarrow_k$ connectives. We did this implicitly in the previous steps.
The $x : X$ and $y : Y$ are useless, and so are deleted. The final grammar thus looks as follows:

$$S := (P' \odot I) \odot I$$
$$a : A$$
$$b : B$$
$$a : (((P'/A)/J)/A)/J$$
$$a : (P' \downarrow_1 T'_1)/J$$
$$a : (T'_1 \downarrow_2 (P'/A))/J$$
$$b : (((P'/B)/J)/B)/J$$
$$b : (P' \downarrow_1 T'_3)/J$$
$$b : (T'_3 \downarrow_2 (P'/B))/J$$

■

5 From MCFG to D

A very recent result [3] shows that $MIX_3 = \{w \in \{a, b\}^* | \ |w|_a = |w|_b = |w|_c\}$, i.e. the permutation closure of $\{a^n b^n c^n | n \geq 1\}$ is not a well-nested $MCFL$. It is shown in ([4]) that 1-D_{IJ} can recognize MIX_n for an arbitrary n, i.e. the permutation closure of $\{a_1^n ... a_m^n | n \geq 1\}$ for fixed m. The latter fact is obtained by exploiting $c : (A \uparrow I) \downarrow B$ lexical entries, i.e. take an expression of type A, non-deterministically insert a separator somewhere in that expression, and infixate c in that location to produce an expression of type B. Now, given the result of the previous section, unless MIX is sóme well-nested $MCFL$, this means that the described use of product units may discriminate between well-nested and ill-nested, i.e. as $L(1\text{-}D_J) =$

$MCFL_{wn}$, it might be so that $L(1\text{-}D_{IJ}) = lMCFL$. We do not provide a proof of this here, but rather show how a specific subclass of $MCFG$ can be converted to a $L(1\text{-}D_J)$ grammar. In other words, this says that for specific kinds of ill-nested grammars, there also is a well-nested grammar, though of possibly higher dimension.

Definition 12 *Given some production rule $p_1 = A(\alpha) \rightarrow B_1(\beta_1)...B_n(\beta_n)$ and a production rule $p_2 = C(\gamma) \rightarrow D_1(\delta_1)...D_m(\delta_m)$, we say that p_2 is in the* scope *of p_1 iff p_2 is reachable from p_1, i.e. there is some B_i in the RHS of p_1 such that there is a production rule with B_i on the LHS etc etc. that reaches C.*

Definition 13 (Single ill-nested MCFG) *An $MCFG$ G is single ill-nested, denoted $MCFG_{sin}$ iff for all ill-nested production rules p, there is no ill-nested production rule p' such that p' is in the scope of p.*

We saw above that translating an ill-nested rule into a D lexical entry (or a set of entries) is not straightforward at all. However, it can be done using higher-order constructs, given that every argument of the tuples on the right-hand side has some specific type. For instance, consider the rule

$$A(aX_1Y_1, X_2Y_2) \rightarrow B(X_1, X_2)C(Y_1, Y_2)$$

If we would be able to keep track of the types of Y_1, Y_2 as the C tuple is built up, say that Y_1 has type C_1 and Y_2 has type C_2, we could have the type assignment $a : (A/C_2)/(B \odot (C \uparrow C_2))$, which means something like 'concatenate a to a B tuple with a C tuple without the right component inserted, and then concatenate the missing right component of C to the outside'. Now, if an ill-nested rule has only well-nested rules in its scope, we can indeed keep track of the tuple components as they are built up. So, instead of first building up B, C tuples in the example, we would build up a B tuple, and then insert a C tuple as it is built up. So the derivation of C is carried over to the one ill-nested rule. Without loss of generality, we assume that any ill-nested rule is of the form $A_0(\alpha_1^1...\alpha_n^1...\alpha_1^n...\alpha_n^n) \rightarrow A_1(\alpha_1^1, ..., \alpha_1^n)...A_n(\alpha_n^1, ..., \alpha_n^n)$. Given such an ill-nested rule, we consider the A_1 tuple and add type assignments for A_2 through A_n such that we get the following subsequent derived strings:

$$\alpha_1^1, ..., \alpha_1^n : A_1$$

$$\alpha_1^1\alpha_2^1, ..., \alpha_1^n\alpha_2^n : C_{A_2}$$
$$\alpha_1^1\alpha_2^1\alpha_3^1, ..., \alpha_1^n\alpha_2^n\alpha_3^n : C_{A_3}$$
$$\vdots$$
$$\alpha_1^1...\alpha_n^1...\alpha_1^n...\alpha_n^n : A_0$$

So our construction consists of two parts: assigning types such that we can cycle through all RHS predicates in order to simulate the ill-nested rule, and for each RHS predicate, simulating the construction of the tuple, and we describe the two parts separately.
Firstly, we distinguish between two kinds of rules. We call a production rule $A(\overrightarrow{\alpha}) \rightarrow B_1(\overrightarrow{\beta_1})...B_n(\overrightarrow{\beta_n})$ *distributed* iff there are $x_i, x_j \in \overrightarrow{\beta_k}$ such that $x_i \in \alpha_m$ and $x_j \notin \alpha_m$. So, given two variables in the same tuple on the RHS, they do not occurr in the same component in the LHS. Henceforth, we distinguish distributed and non-distributed rules. Furthermore, we call a rule recursive when $A = B_i$ for some i, and logically, the other rules are non-recursive.
We first show how to go from a tuple $\alpha_1, ..., \alpha_n : T_i$ to a tuple $\alpha_1\beta_1, ..., \alpha_n\beta_n : T_o$ given some tuple $\beta_1, ..., \beta_n : B$. The idea is to trace the components β_i of this tuple and to create a derivation cycle in which we insert each component into the input tuple. We assume working on an $MCFG$ that has an ordering on the non-terminals such that for all clauses $c \in P$ it holds that if $A_i \rightarrow A_j...A_k$ then $i \leq min(j, ..., k)$.[1] Furthermore, we assume that well-nested rules of the grammar are lexicalized *componentwise*, i.e. every component of a tuple starts with a non-terminal item.[2]

Construction 1 *Given that we want to construct $\alpha_1\beta_1, ..., \alpha_n\beta_n : T_o$ from $\alpha_1, ..., \alpha_n : T_i$ and $\beta_1, ..., \beta_n : B$, for all B clauses c do the following:*

- *If c is recursive, add type assignments $x_i^A : X_i^A$ for each distributed variable X_i^A associated with the non-terminal A. Now we construct two type assignments, (1) one for initiating the rule from the T_i tuple, and (2) one for the actual recursion.*

[1] The assumption is without loss of generality, for if there is a rule such that the requirement does not hold, i.e. $i > j$ for some RHS A_j, perform a substitution, where you replace the rule by a set of rules, created by substituting A_j with all possible A_j rules. Do this for all rules and the property eventually holds. As substitution preserves string language, this algorithm also does.

[2] This is probably also without loss of generality, as well-nested $MCFL$ can be lexicalized preserving dimension and string language ([5],p.47, lemma 4). Extending the algorithm to work componentwise should not be difficult.

Assignment 1 *Following the components from left (1) to right (n), assign the anchor of the first component the type* $(T_i \downarrow T_1)/P'_1$. *For every component (j) except for the last one, assign the anchor the type* $(T_j \downarrow T_{j+1})/P'_j$, *and for the last component assign the anchor the type* $(T_m \downarrow T_{m+1})/P'_m$ *where* P'_k *is constructed as in the construction from [5] but including the distributed variables.*

Assignment 2 *Following the components from left (1) to right (n), assign the anchor of the first component the type* $((T_{m+1} \uparrow X_1^A) \downarrow R_1)/Q'_1$. *For every component (j) except for the last one, assign the anchor the type* $((R_j \uparrow X_j^A) \downarrow R_{j+1})/Q'_j$, *and for the last component assign the anchor the type* $((T_m \uparrow X_m^A) \downarrow T_{m+1})/Q'_m$ *where* Q'_k *is constructed as in the construction from [5] but including the distributed variables.*

- *If c is non-recursive, we do the same as in the recursive case, but in assignment 1, we do not include distributed variables,* $P'_m = A_0$ *and* $Q'_m = A_0$.

Repeatedly applying this construction, we can give a 1-$\mathbf{D_J}$ grammar, given an arbitrary single ill-nested $MCFG$.

Theorem 1 *Single ill-nested* $MCFL_{sin} \subseteq L(1\text{-}\mathbf{D_J})$.

PROOF: Let $R = (N, T, V, P, S)$ be a lexicalized $MCFG_{sin}$. Without loss of generality, we assume that there exists an ordering on N such that for all clauses $c \in P$ it holds that if $c = A_j \rightarrow A_k\gamma$ then $ord(A_j) \leq ord(A_k)$.[3]
For every well-nested rule that is in a path from the start symbol without passing an ill-nested rule, we add a lexical entry according to the construction in [5]. For each ill-nested rule, we follow the following construction. Let $p = A_0(a\alpha_1^1...\alpha_n^1, ..., \alpha_n^1...\alpha_n^n) \rightarrow A_1(\alpha_1^1, ..., \alpha_1^n)...A_n(\alpha_n^1, ..., \alpha_n^n)$ be an ill-nested rule in P such that none of the RHS rules except for A_1 do not reach an ill-nested rule. We add new primitive types C_{A_i} for $2 \leq i \leq n$ denoting in which phase of the cycle we are. Then, for each A_i for $2 \leq i \leq n$ we construct new type assignments as follows:

- If $i = 2$, then we construct new type assignments according to construction 1 with input type A_1 and output type C_{A_2}.
- If $2 < i < n$ then construct new type assignments according to construction 1 with input type C_{A_i} and output type $C_{A_{i+1}}$.

[3]The Greibach Normal Form for Context Free Grammar applies this concept.[2]

- If $i = n$ then construct new type assignments according to construction 1 with input type C_{A_n} and output type A_0.

Correctness: It is obvious that when well-nested rules can be used only out of the scope of ill-nested rules that the construction from [5] suffices, so we need to show that construction 1 suffices to simulate ill-nested rules. Given an ill-nested rule $c = A_0(a\alpha_1^1...\alpha_n^1, ..., \alpha_n^1...\alpha_n^n) \to A_1(\alpha_1^1, ..., \alpha_1^n)...A_n(\alpha_n^1, ..., \alpha_n^n)$ it is clear that the construction of theorem 1 cycles through the $A_2, ..., A_n$ tuples, adding them in order of appearance. For any derivation crossing this ill-nested rule, a specific combination of $A_2, ..., A_n$ tuples is added to an unspecified A_1 tuple. Thus, we need to show that construction 2 allows to independently add each specific A_i tuple such that each combination can be made. In construction 2, we distinguish between recursive and non-recursive rules. It is clear that for each rule, we add types that simulate one rule application. So, given a derivation for an A_i it either applies a bunch of recursive rules and then a non-recursive one, or it directly applies a non-recursive rule. In the first case, the translation of the recursive rules ensures that we end up with a tuple with some X_i variables, that get lifted out after application of a non-recursive rule, using case 2 of the translation, which uses $(T_i \uparrow_j X_j) \downarrow T_k)$ constructions. In the second case, we just have that we start from an A_1 tuple, and put in components without distributed variables. This is done by the $A \downarrow T_1$,$T_i \downarrow T_{i+1}$,$T_{n-1} \backslash T_n$ constructions.

Example. The following $MCFG$ recognizes $RESP_2 = \{a^n b^n c^m d^m e^n f^n g^m h^m | n, m \leq 1\}$:

$$S(X_1Y_1X_2Y_2) \to A(X_1, X_2)K(Y_1, Y_2)$$
$$A(aXb, eYf) \to A(X, Y)$$
$$A(ab, ef) \to \epsilon$$
$$K(cXd, gYh) \to K(X, Y)$$
$$K(cd, gh) \to \epsilon$$

The A rules are not reached by the second or greater argument on the RHS of an ill-nested rule and the B rules are only reached by the second or greater argument on the RHS of an ill-nested rule. So we follow [5] to get type assignments for the A rules:
$a : (((A/F)/E)/J)/B$
$a : (A/F)/(A \odot (B \bullet J \bullet E))$
$b : B$
$e : E$

$f : F$
Now we follow construction 1 to get rules for the first, ill-nested rule:
For the first, recursive B rule, we add types for x, y,c and g:
$x : X$
$y : Y$
$c : ((A \downarrow T_3)/D)/X$
$g : ((T_3 \backslash T_1)/H)/Y$
$c : (((T_1 \uparrow X) \downarrow T_2)/D)/X$
$g : (((T_2 \uparrow Y) \downarrow T_1)/H)/Y$
$d : D$
$h : H$
For the second, non-recursive B rule, we add types for c, d:
$c : (A \downarrow T_4)/D$
$g : (T_4 \backslash S)/H$
$c : ((T_1 \uparrow X) \downarrow T_5)/D$
$g : ((T_5 \uparrow Y) \downarrow S)/H$

■

6 Conclusion & Future work

The main result of this paper is that general (of arbitrary dimension) first-order Displacement Calculus ($\mathbf{D^1}$) is weakly equivalent to higher-order, but one-dimensional Displacement Calculus without the use of product unit in higher-order types (1-D_J). Combined with the result of [5] that $\mathbf{D^1}$ is weakly equivalent to the class of well-nested Multiple Context Free Languages ($MCFL_{wn}$), 1-D_J is also weakly equivalent to $MCFL_{wn}$.
The combined results give a trade-off between order and dimension: D^1 has an infinite number of residuated triples $\uparrow_k, \downarrow_k, \odot_k$, but only first-order types, whereas 1-D_J has only two residuated triples, namely the standard Lambek connectives $/, \backslash, \bullet$ and $\uparrow, \downarrow, \odot$. However, the latter system allows higher-order types. When laid down next to well-nested Multiple Context Free Grammar, however, the relation between D^1 and $MCFL_{wn}$ is more direct and refined, as for each k, the class of D^1 grammars with dimension at most k coincides with $(k+1)$-$MCFL_{wn}$, so the place of separators in an expression really corresponds to tuple delimiters in $MCFG_{wn}$, whereas the equivalence between 1-D_J and $MCFL_{wn}$ does not consist of an equivalence at each dimension.
We also showed that a specific subclass of lexicalized $MCFL$ is a subset of

lexicalized $MCFL$ by showing that one can construct a weakly equivalent 1-D_J grammar.
As is shown in [4], the permutation closure of any context-free language can be generated by a D grammar by exploiting product units in higher-order position. As MIX_3 is the permutation closure of a context-free language, but not a well-nested $MCFL$ of dimension 2 (and quite likely not well-nested at all), this shows that 1-D_{IJ}, i.e. the full two-dimensional Displacement Calculus goes beyond well-nesteds $MCFL$ in terms of generative capacity. The question is, however, how much further this goes. We conjecture here that it is limited to lexicalized $MCFL$, although a full proof of equivalence has not been constructed yet. Constructively going from an arbitrary lexicalized $MCFL$ to a 1-D_{IJ} seems a quite challenging route, and we have no clue as to how this might be done. However, the other way around may be simpler. A type assignment of the form $x : (A \uparrow I) \downarrow B$ means something as, given an expression of type A, insert a separator anywhere and place the x there to obtain an expression of the form B. This behaviour might be simulated in an $MCFG$ by introducing a (predicate) type A' of a higher dimension than A, and adding a set of production rules that gives the combinatorics of creating all possible discontinous A-strings. Further study may provide an explicit proof of the conjecture.

References

[1] P. Boullier. Proposal for a natural language processing syntactic backbone. 1998.

[2] S.A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *Journal of the ACM (JACM)*, 12(1):42–52, 1965.

[3] M. Kanazawa and S. Salvati. Mix is not a tree-adjoining language.

[4] G. Morrill and O. Valentín. On calculus of displacement. In *TAG*, volume 10, pages 45–52, 2010.

[5] GJ Wijnholds. Investigations into categorial grammar: Symmetric pregroup grammar and displacement calculus. 2011.