Transforming Displacement Grammars into RCG Format

Kasper Luchsinger

August 20, 2012

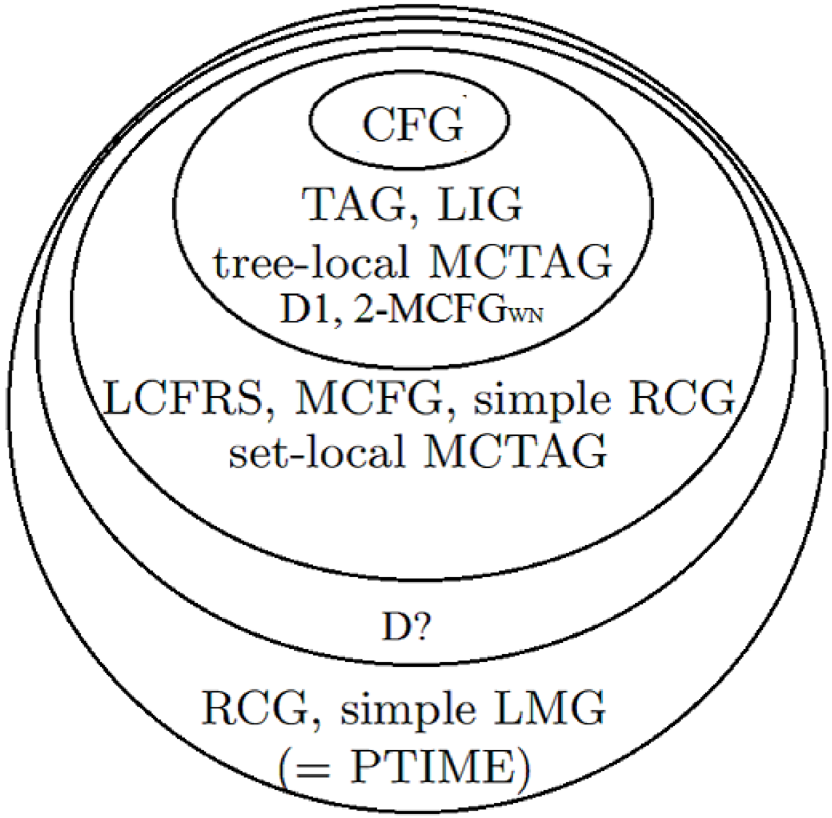
I would like to take a moment to thank my professor, Michael Moortgat, for helping me to understand and fix my many mistakes, and for giving some of my more crazy theories a chance which made this whole experience much more fun. I would also like to thank my parents Ria and Frits and my girlfriend Helen, for all their love and support, and for being so very patient with me.

1 Beyond CFG

To deal with the syntax of natural and artificial languages, the theory of formal languages was developed by Chomsky and others in the 1950s [1]. Chomsky proposed using Context-Free Grammars could be used for describing the structure of sentences and words in languages. CFGs can be used to analyse and explain the forming of many different types of sentence patterns. Unfortunately, not all natural and artificial language phenomena can be explained by context-free grammars. The following problems fall outside of the scope of Context-Free Grammars [2]:

- Three or more counting dependencies: $\{a^n b^n c^n | n > 0\}$
- Crossing dependencies: The copy language - $\{ww | w \in \{a, b\}^+\}$, MIX_3 - $\{w \in \{a, b, c\}^+| \ \ |w|_a = |w|_b = |w|_c\}$

These problems (and many others) are caused by the discontinuities, which also occur in natural language. Range Concatenation Grammars were first developed by Boullier [5] to help deal with such problems. More restrictive variations have also been developed, including Simple Range Concatenation Grammars, Linear Context Free Rewriting Systems and Multiple Context-Free Grammars. These three systems have been shown to be equivalent [2]. Also developed to deal with these natural language phenomena are Tree Adjoining Grammars, which have been shown equivalent to the class of well-nested 2-SRCGs (and therefore well-nested 2-MCFGs) [2]. This is significant because it means that, among other things, TAGs could be used to analyze as much as 99.89% of the natural language recorded into some of the major treebanks [3]. Another system that can succesfully handle many natural language phenomena is the Calculus of Displacement (**D**) [6]. As in every formalism, the key elements expressed by the Calculus of Displacement are the dependencies between lexical items. This would suggest it should be possible to compare its expressive power to that of the other formalisms. A fraction of this system (First-order Displacement Calculus) has already been shown to be equivalent to TAGs [7]. However, it has not yet been determined what level of expressivity the calculus as a whole has, in relation to any of the other systems. It is the goal of this paper to develop a method of transforming **D**-grammars into RCGs that are as restricted as possible, thereby approaching the SRCG format. In this manner we will prove displacement grammars to be strongly equivalent to RCGs with these restrictions. We will first handle displacement grammars with only a single separator. But we will show that this method can be used to succesfully transform displacement grammars with any number of separators.



The language hierarchy of different grammar formalisms [2], where the position of D is uncertain.

2 Displacement Grammars

2.1 Definition

A **D**-grammar G is a tuple (Σ, δ, S) where:
- Σ is a finite set of words
- δ is a relation that matches types to words in Σ
- S is the distinguished type (the start type). Note that S can be a *complex* type.

The calculus of Displacement is a logic of concatenation and intercalation. The types of the displacement calculus **D** classify strings over a vocabulary including a distinguished placeholder 1, also called the *separator*. The sort $i \in \mathcal{N}$ of a (discontinuous) string is the number of separators it contains, punctuating it into i+1 continuous substrings. The types of **D** are sorted into types $\mathcal{F}_i$ of sort

i as follows:

$$\mathcal{F}_j := \mathcal{F}_i \backslash \mathcal{F}_{i+j} \tag{1}$$
$$\mathcal{F}_j := \mathcal{F}_{i+j} / \mathcal{F}_i \tag{2}$$
$$\mathcal{F}_{i+j} := \mathcal{F}_i \bullet \mathcal{F}_j \tag{3}$$
$$\mathcal{F}_0 := I \tag{4}$$
$$\mathcal{F}_j := \mathcal{F}_{i+1} \downarrow_k \mathcal{F}_{i+j}, 1 \leq k \leq i+1 \tag{5}$$
$$\mathcal{F}_j := \mathcal{F}_{i+j} \uparrow_k \mathcal{F}_j, 1 \leq k \leq i+1 \tag{6}$$
$$\mathcal{F}_j := \mathcal{F}_{i+1} \odot_k \mathcal{F}_j, 1 \leq k \leq i+1 \tag{7}$$
$$\mathcal{F}_1 := J \tag{8}$$

If G is a displacement grammar, A, B and C will denote types in G. a will denote a terminal symbol. For now, we will only discuss discontinuities with a single separator. Therefore, we will abbreviate $\downarrow_1$ as $\downarrow$, $\uparrow_1$ as $\uparrow$ and $\odot_1$ as $\odot$. However, the transformation described in this paper would work for discontinuities with any number of separators.

2.2 Derivations in D

For derivations we will use the labelled natural deduction rules [8]. γ and α are used to represent *strings* of arbitrary length, where a string $\gamma : A$ indicates the type A can be assigned to the string γ. $\gamma : A \odot I$ means the string γ can be split into a *pair* of strings $(\gamma_1, \gamma_2) : A$ for some type A. $+$ is used for concatenation. a and b are used to represent *terminal symbols*, where $(a_1, a_2) : A$ is a pair of terminal symbols such that $a_1 + 1 + a_2 : A$. The rules are as follows::

$$\frac{\Delta : C}{\Delta(I) : C}\, I$$

$$\frac{}{1 : J}\, J$$

$$\frac{\begin{matrix}\vdots \\ \gamma : B\end{matrix} \qquad \begin{matrix}\vdots \\ \alpha : B\backslash C\end{matrix}}{\gamma + \alpha : C}\, E\backslash$$

$$\frac{\begin{matrix}\vdots \\ \gamma : B/C\end{matrix} \qquad \begin{matrix}\vdots \\ \alpha : C\end{matrix}}{\gamma + \alpha : B}\, E/$$

$$\frac{\begin{matrix} \\ \vdots \\ \gamma : B \bullet C \end{matrix} \qquad \begin{matrix} \text{a1: B} \quad \text{a2: C} \\ \vdots \\ \Delta(a1 + a2) : D \end{matrix}}{\Delta(\gamma) : D}\ E\bullet$$

$$\frac{\begin{matrix} \vdots \\ \alpha : B \downarrow C \end{matrix} \qquad \begin{matrix} \vdots \\ (\gamma 1, \gamma 2) : B \end{matrix}}{\gamma 1 + \alpha + \gamma 2 : C}\ E\downarrow$$

$$\frac{\begin{matrix} \vdots \\ \alpha : C \end{matrix} \qquad \begin{matrix} \vdots \\ (\gamma 1, \gamma 2) : B \uparrow C \end{matrix}}{\gamma 1 + \alpha + \gamma 2 : B}\ E\uparrow$$

$$\frac{\begin{matrix} \\ \vdots \\ \gamma : C \odot B \end{matrix} \qquad \begin{matrix} \text{(c1,c2) : C} \quad \text{b : B} \\ \vdots \\ \Delta(c1 + b + c2) : D \end{matrix}}{\Delta(\gamma) : D}\ E\odot$$

$$\frac{\begin{matrix} b : B \\ \vdots \\ a + \gamma : C \end{matrix}}{\gamma : B \backslash C}\ I\backslash$$

$$\frac{\begin{matrix} c : C \\ \vdots \\ \gamma + c : B \end{matrix}}{\gamma : B / C}\ I/$$

$$\frac{\begin{matrix} \vdots \\ \alpha : B \end{matrix} \qquad \begin{matrix} \vdots \\ \gamma : C \end{matrix}}{\alpha + \gamma : B \bullet C}\ I\bullet$$

$$\dfrac{\begin{matrix}(b1,b2):B\\ \vdots\\ b1+\alpha+b2:C\end{matrix}}{\alpha:B\downarrow C}\, I\downarrow$$

$$\dfrac{\begin{matrix}c:C\\ \vdots\\ \alpha1+c+\alpha2:B\end{matrix}}{(\alpha1,\alpha2):B\uparrow C}\, I\uparrow$$

$$\dfrac{\begin{matrix}\vdots\\ (\alpha1,\alpha2):C\end{matrix} \qquad \begin{matrix}\vdots\\ \gamma:B\end{matrix}}{\alpha1+\gamma+\alpha2:C\odot B}\, I\odot$$

Let $G = (\Sigma, \delta, S)$ be a **D**-grammar. We define the string language of G as $L_S(G) = \{w|w = a_1...a_n : S\}$ where $a_1...a_n \subset \Sigma^*$.

We will now present a few examples of labelled natural deduction proofs. In our first example, we present a toy grammar that generates the sentence *someone is needed* in two ways, in order to show how ambiguity can occur. *is needed* is treated as a single constituent for simplicity.
Toy Grammar:
someone : $S/(N\backslash S)$
is-needed : $(S/(N\backslash S))\backslash S$
Two ways of proving *someone* + *is-needed* : S:

(1)

$$\dfrac{\mathit{someone} : S/(N\backslash S) \qquad \mathit{is\text{-}needed} : (S/(N\backslash S))\backslash S}{\mathit{someone} + \mathit{is\text{-}needed} : S}\, E\backslash$$

(2)

$$\dfrac{\mathit{someone} : S/(N\backslash S) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p : N}\,1 \qquad \dfrac{}{q : N\backslash S}\,2}{p+q : S}\,E\backslash}{p : S/(N\backslash S)}\,I/^2 \qquad \mathit{is\text{-}needed} : (S/(N\backslash S))\backslash S}{p + \mathit{is\text{-}needed} : S}\,E\backslash}{\mathit{is\text{-}needed} : N\backslash S}\,I\backslash^1}{\mathit{someone} + \mathit{is\text{-}needed} : S}\, E/$$

In our next example, the toy grammar generates the language $MIX_3 = \{w \in \{a,b,c\}^+ || w|_a = |w|_b = |w|_c\}$. In other words the language of words consisting of a's, b's and c's, with exactly the same amount of each letter. It is a language with interesting properties that has been studied by Kanazawa and Salvati, among others [9].

Toy grammar:

$a : (S \uparrow I) \downarrow A \ ; \ A$

$b : (A \uparrow I) \downarrow B$

$c : (B \uparrow I) \downarrow S$

Proof that $abbcac \in MIX_3$:

$$
\dfrac{
\dfrac{
\dfrac{
b : (A \uparrow I) \downarrow B
\quad
\dfrac{
\dfrac{
\dfrac{
c : (B \uparrow I) \downarrow S
\quad
\dfrac{
\dfrac{
\dfrac{
a : (S \uparrow I) \downarrow A
\quad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{a : A}{a + I : A}\, I
}{(a, \epsilon) : (A \uparrow I)}\, I \uparrow
\quad
b : (A \uparrow I) \downarrow B
}{a + b : B}\, E \downarrow
}{a + b + I : B}\, I
}{(a + b, \epsilon) : (B \uparrow I)}\, I \uparrow
}{a + b + c : S}\, E \downarrow
}{a + b + I + c : A}\, I
}{(a + b, c) : (A \uparrow I)}\, I \uparrow
}{a + b + a + c : A}\, E \downarrow
}{a + b + I + a + c : A}\, I
}{(a + b, a + c) : (A \uparrow I)}\, I \uparrow
}{a + b + b + a + c : B}\, E \downarrow
}{a + b + b + I + a + c : B}\, I
}{(a + b + b, a + c) : (B \uparrow I)}\, I \uparrow
\quad
c : (B \uparrow I) \downarrow S
}{a + b + b + c + a + c : S}\, E \downarrow
$$

3 Range Concatenation Grammars

We borrow our definition of Range Concatenation Grammars from Kallmeyer [2], as well as a few important facts about PRCGs.

3.1 Definition

1) A Positive Range Concatenation Grammar (PRCG) is a tuple $G = (N, T, V, S, P)$ where:
N is a finite set of predicate names with an arity function $dim : N \to \mathbb{N}$
T and V are disjoint finite sets of terminals and variables
$S \in N$ is the start predicate, a predicate of arity 1. P is a finite set of clauses of the form:
$A_0(x_{01}, ..., x_{0a_0}) \to \epsilon$
or
$A(\alpha_1, ..., \alpha_{dim(A)}) \to A_1(X_1^{(1)}, ..., X_{dim(A_1)}^{(1)}) \cdots A_m(X_1^{(m)}, ..., X_{dim(A_m)}^{(m)})$
for $m \geq 0$ where $A, A_1, ..., A_m \in N, X_j^{(i)} \in V$ for $1 \leq i \leq m, 1 \leq j \leq dim(A_i)$ and $\alpha_i \in (T \cup V)^*$ for $1 \leq i \leq dim(A)$, and
2) A PRCG $G = (N, T, V, P, S)$ is a k-RCG if for all $A \in N, dim(A) \leq k$. We also call $dim(A)$ the *block-degree* of A. Predicates of block-degree k will contain k-1 gaps, giving it a *gap-degree* of k-1.

A PRCG G is:
• *non-combinatorial if for each clause $c \in P$, all the arguments in the right-hand side of c are single variables.*
• *bottom-up linear if for every clause $c \in P$, no variable appears more than once in the left-hand side of c.*
• *top-down linear if for every clause $c \in P$, no variable appears more than once in the right-hand side of c.*
• *linear if it is top-down and bottom-up linear.*
• *bottom-up erasing if for every clause $c \in P$, each variable occuring in the right-hand side of c occurs also in its left-hand side.*
• *top-down erasing if for every clause $c \in P$, each variable occuring in the left-hand side of c occurs also in its right-hand side.*
• *non-erasing if it is top-down and bottom-up non-erasing.*
• *simple if it is non-combinatorial, linear, and non-erasing.*

Our transformation will result in a linear, non-erasing, but combinatorial RCG. This means it can be transformed into a non-combinatorial RCG which would meet all the requirements of a *simple* RCG, except for top-down linearity [5]. We will suggest a possible alternative to this later. Because we allow combinatorial clauses, we do not stricly enforce resource sensitivity, unlike in **D**. However, our transformation method ensures that all operations applied to the input in our target RCG, could have been applied in our source grammar. This means the target grammar *is* resource sensitive, even though combinatorial RCGs are not. They are now resource sensitive for the same reason that Displacement grammars are resource sensitive: adding an item is only allowed if an item of higher arity has just been removed.

3.2 Range and string language

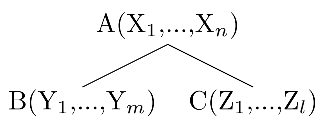
The range language *of an* $A \in N$ *with dim(A)* $= k$ *for some* w T^* *is:*
$R(A,w) = \{\boldsymbol{p} | \boldsymbol{p}$ *is a k-dimensional range vector, and* $A(\boldsymbol{p}) \overset{*}{\Rightarrow}_{G,w} \epsilon\}$.

The string language *of an* $A \in N$ *with dim(A)* $= k$ *for some* $w \in T^*$ *is:*
$L(A, w) = \{\mathbf{p}(w) | \mathbf{p} \in R(A, w)\}$.

The string language *of a PRCG G is:*
$L(G) = \{w \in T^* | \langle\langle 0, |w| \rangle\rangle \in R(S, w)\}$.

3.3 Derivations for PRCGs

Derivations and proofs for a PRCG $G = (N, T, V, P, S)$ will be done using derivation trees of the form:

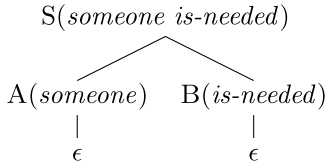


Where:
$\{A, B, C\} \subseteq N$
$\{X1, ..., X_n, Y_1, ..., Y_m, Z_1, ..., Z_l\} \subset (V \cup T)^*$
$\{A(X_1, ..., X_n) \to B(Y_1, ..., Y_m), C(Z_1, ..., Z_l)\} \subseteq P$

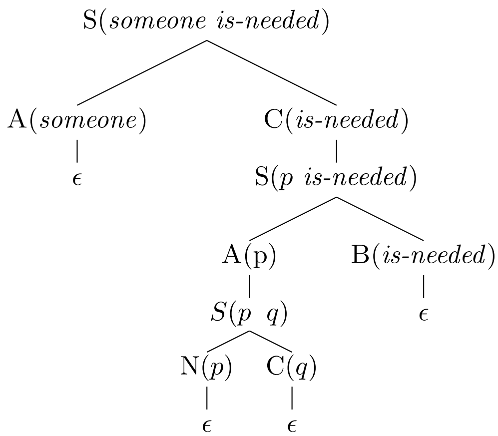
We will now present a few examples of derivations for PRCGs. To do this we present a toy grammar generating *someone is-needed* which, as we shall see, will also generate the sentence in two distinct ways (the same ambiguity is present):

$$\begin{aligned}
A(\textit{someone}) &\to \epsilon \\
B(\textit{is-needed}) &\to \epsilon \\
S(XY) &\to A(X), C(Y) \\
C(X) &\to S(pX) \\
S(XY) &\to A(X), B(Y) \\
S(XY) &\to N(X), C(Y) \\
A(X) &\to S(Xq) \\
C(q) &\to \epsilon \\
N(p) &\to \epsilon
\end{aligned}$$

One way to derive *someone is-needed* in G is:



Another way would be:



4 Transforming D into PRCG

In this section, we will present the function that transforms a set of lexical items of a displacement grammar (also called source grammar or G_S from now on) into a set of linear non-erasing combinatory and non-combinatory PRCG clauses. We apply the transformation to all types and (subtypes of)* subtypes assigned to the lexical items in G_S, while adding what we call base-clauses that represent axioma's. The equivalence of G_S and G_T is obvious from the fact that each clause performs the same operations as the application of some deduction rule in **D**, making any derivation of some word w in the string language of G_T strongly equivalent with the derivation for w in G_S.

We will use two functions to transform the *set of types* assigned to the lexical items of a source grammar G_S, into a *set of SRCG-clauses* for our target grammar G_T. $\lceil\ \rceil$ will be the *input* function corresponding to elimination rules, and $\lfloor\ \rfloor$ the *output* function corresponding with the introduction rules. Both functions are recursive. $\lceil A \rceil$ will represent applying the input function to some type A which can be complex or atomic. A_{atomic} will be used to indicate that

A is an *atomic type*. T_A is a single predicate name used to easily keep track of the fact that some (sub)type A from G_S was used to acquire it. $[a]$ and $[b]$ will be used as single terminal symbols such that $T_A([a]) \to \epsilon$ and $T_B([b]) \to \epsilon$. $([a_1], [a_2])$ will be a pair of terminal symbols such that $T_A([a_1]\#[a_2]) \to \epsilon$ where # is a terminal symbol used as a *separator* (much like 1 in **D**). These symbols are used for convenience and easy reading, but not required to maintain strong equivalence. The two functions will take a type assigned to a lexical item in G_S, and return a set of clauses. Our transformation is completed by applying the correct functions to each type in G_S.

$$a : t \longrightarrow T_A(...) \to (...)$$

Graphical representation of the input and output functions. $\alpha : A$ *means* α *is a string assigned a type* A *in* G_S*, and* $T_A(...) \to (...)$ *is an RCG-clause for* G_T.

$$\mathbf{D}(G_S) \longrightarrow RCG(G_T)$$

Graphical representation of the transformation, where G_S *is an arbitrary displacement grammar: a collection of lexical items with their assigned types.* G_T *is the target RCG: a set of clauses.*

4.1 Base clauses

When transforming a displacement grammar G_S into a RCG G_T, the first step is to include a corresponding base clause that produces each item in the lexicon, like the axioms in labelled natural deduction:
$a : A \Rightarrow T_A(a) \to \epsilon$
Informally, we could say the base clauses will serve as the lexicon for G_T, and the support clauses will serve as the natural deduction rules that would be used in a derivation in **D**. As such, each support clause will represent a valid rule that can be applied whenever the conditions for it are met (ie whenever an appropriate type is present in the input).

4.2 Support clauses

Next, we include support clauses that correspond to rules used in natural deduction. We will include two types of support clauses, depending on what the natural deduction proof for G_S would look like. We use elimination clauses when, in the corresponding labelled natural deduction proof, we would use elimination rules. We use introduction clauses when we would use introduction rules. When adding a hypothetical item, terms like $[a/b]$ will refer to a *unique* terminal symbol used to represent a hypothetical item of type A/B. We will use the notation $\lceil A \rceil$ to mean applying the transformation to type A.

Elimination clauses: The elimination clauses are as follows:
case 1 $\lceil A\backslash B\rceil = \lceil B\rceil \cup \lfloor A\rfloor \cup \{T_B(XY) \rightarrow T_A(X), T_{A\backslash B}(Y)\}$
Corresponding natural deduction rule:

$$\dfrac{\begin{array}{c}\vdots\\ \gamma : B\end{array} \qquad \begin{array}{c}\vdots\\ \alpha : B\backslash C\end{array}}{\gamma + \alpha : C}\ E\backslash$$

case 2 $\lceil A/B\rceil = \lceil A\rceil \cup \lfloor B\rfloor \cup \{T_A(XY) \rightarrow T_{A/B}(X), T_B(Y)\}$
Corresponding natural deduction rule:

$$\dfrac{\begin{array}{c}\vdots\\ \gamma : B/C\end{array} \qquad \begin{array}{c}\vdots\\ \alpha : C\end{array}}{\gamma + \alpha : B}\ E/$$

case 3 $\lceil A \bullet B\rceil$ This is somewhat of a special case. Recall the corresponding natural deduction rule:

$$\dfrac{\begin{array}{c}\vdots\\ \gamma : B \bullet C\end{array} \qquad \begin{array}{c}\text{a1: B} \quad \text{a2: C}\\ \vdots\\ \Delta(a1 + a2) : D\end{array}}{\Delta(\gamma) : D}\ E\bullet$$

What's different about this case is the type D. How do we find D? In other words, how do we know when to 'split' the type in two. The answer is: as soon as possible. Any product can be eliminated as soon as it is the main connective. If $\delta(c, A \bullet B)$ for some item $c \in \Sigma$, we can do this with the following set of clauses: $\{S(XYZ) \rightarrow T_{A\bullet B}(Y), S(X[a][b]Z)\} \cup \{T_A([a]) \rightarrow \epsilon\} \cup \{T_B([b]) \rightarrow \epsilon\} \cup \{T_{A\bullet B}(c) \rightarrow \epsilon\}$ where S is the distinguished type. Otherwise, we want to split each hypothetical item $[a \bullet b]$ as soon as it is added by a clause. We can do this by scanning for types $A \bullet B$ that will occur, and creating clauses to split them up straight away. To sum it up, here is an informal description of what needs to be done:

If $\delta(c, A \bullet B)$ and $c \in \Sigma$ then $\lceil A \bullet B\rceil = \{S(XYZ) \rightarrow T_{A\bullet B}(Y), S(X[a][b]Z)\} \cup \{T_A([a]) \rightarrow \epsilon\} \cup \{T_B([b]) \rightarrow \epsilon\} \cup \{T_{A\bullet B}(c) \rightarrow \epsilon\} \cup \lceil A\rceil \cup \lceil B\rceil$
Otherwise $\lceil A \bullet B\rceil = \{T_A([a]) \rightarrow \epsilon\} \cup \{T_B([b]) \rightarrow \epsilon\} \cup \{T_{A\bullet B}(c) \rightarrow \epsilon\} \cup \lceil A\rceil \cup \lceil B\rceil$
We also need to split the type, as soon as possible. We can do this by adding a clause $T_C(XYZ) \rightarrow T_C(X[a][b]Z), T_{A\bullet B}(Y)$ whenever there is some clause $T_B(...) \rightarrow T_C(...[a\bullet b]...)$, and adding a clause $T_S(XYZ) \rightarrow T_S(X[a][b]Z), T_{A\bullet B}(Y)$ if $(a, A \bullet B) \in \delta$. This will be done by the function *pcheck*, which takes a pair of types as input (a type on the input side, and a current goal type), and outputs clauses that deal specifically with this problem.

case 4 $\lceil A \downarrow B\rceil = \lceil B\rceil \cup \lfloor A\rfloor \cup \{T_B(XYZ) \rightarrow T_{A\downarrow B}(Y), T_A(X\#Z)\}$
Corresponding natural deduction rule:

$$\dfrac{\begin{array}{c}\vdots\\ \alpha : B \downarrow C\end{array} \qquad \begin{array}{c}\vdots\\ (\gamma 1, \gamma 2) : B\end{array}}{\gamma 1 + \alpha + \gamma 2 : C} E\downarrow$$

case 5 $\lceil A \uparrow B \rceil = \lceil A \rceil \cup \lfloor B \rfloor \cup \{T_A(XYZ) \rightarrow T_{A\uparrow B}(X\#Z), T_B(Y)\}$
Corresponding natural deduction rule:

$$\dfrac{\begin{array}{c}\vdots\\ \alpha : C\end{array} \qquad \begin{array}{c}\vdots\\ (\gamma 1, \gamma 2) : B \uparrow C\end{array}}{\gamma 1 + \alpha + \gamma 2 : B} E\uparrow$$

case 6 $\lceil C \odot B \rceil$ This case is much like case 3. Remember the corresponding Natural Deduction rule:

$$\dfrac{\begin{array}{c}\\ \vdots\\ \gamma : C \odot B\end{array} \qquad \begin{array}{c}\text{(c1,c2) : C} \quad \text{b : B}\\ \vdots\\ \Delta(c1 + b + c2) : D\end{array}}{\Delta(\gamma) : D} E\odot$$

Just like in case 3, we want to split this type as soon as possible. The approach is similar to that of case 3:
If $\delta(c, A \odot B)$ and $c \in \Sigma$ then $\lceil A \odot B \rceil = \{S(XYZ) \rightarrow T_{A\odot B}(Y), S(X[a1][b][a2]Z)\} \cup \{T_A([a1]\#[a2]) \rightarrow \epsilon\} \cup \{T_B([b]) \rightarrow \epsilon\} \cup \{T_{A\odot B}(c) \rightarrow \epsilon\} \cup \lceil A \rceil \cup \lceil B \rceil$
Otherwise $\lceil A \odot B \rceil = \{T_A([a1]\#[a2]) \rightarrow \epsilon\} \cup \{T_B([b]) \rightarrow \epsilon\} \cup \{T_{A\odot B}(c) \rightarrow \epsilon\} \cup \lceil A \rceil \cup \lceil B \rceil$
We also need to split the type. We can do this by adding a clause $T_C(XYZ) \rightarrow T_C(X[a1][b][a2]Z), T_{A\odot B}(Y)$ whenever there is some clause $T_B(...) \rightarrow T_C(...[a \odot b]...)$, and adding a clause $T_S(XYZ) \rightarrow T_S(X[a1][b][a2]Z), T_{A\odot B}(Y)$ if $(a, A \odot B) \in \delta$. This will be done by the function *pcheck*, which takes a pair of types as input (a type on the input side, and a current goal type) and outputs clauses that deal specifically with this problem.

Introduction clauses: The introduction clauses are as follows:
case 1 $\lfloor A \backslash B \rfloor = \lfloor B \rfloor \cup \lceil A \rceil \cup \{T_{A\backslash B}(X) \rightarrow T_B([a]X)\} \cup \{T_A([a]) \rightarrow \epsilon\}$
Corresponding natural deduction rule:

$$\dfrac{\begin{array}{c}b : B\\ \vdots\\ a + \gamma : C\end{array}}{\gamma : B\backslash C} I\backslash$$

case 2 $\lfloor A/B \rfloor = \lfloor A \rfloor \cup \lceil B \rceil \cup \{T_{A/B}(X) \rightarrow T_A(X[b])\} \cup \{T_B([b]) \rightarrow \epsilon\}$
Corresponding natural deduction rule:

$$\dfrac{\begin{array}{c}c : C\\ \vdots\\ \gamma + c : B\end{array}}{\gamma : B/C} I/$$

case 3 $\lfloor A \bullet B \rfloor = \lfloor A \rfloor \cup \lfloor B \rfloor \cup \{T_{A \bullet B} \rightarrow T_A(X), T_B(Y)\}$
Corresponding natural deduction rule:

$$\frac{\begin{array}{cc} \vdots & \vdots \\ \alpha : B & \gamma : C \end{array}}{\alpha + \gamma : B \bullet C} I\bullet$$

case 4 $\lfloor A \downarrow B \rfloor = \lceil A \rceil \cup \lfloor B \rfloor \cup \{T_{A \downarrow B}(X) \rightarrow T_B([a1]X[a2])\} \cup \{T_A([a1]\#[a2]) \rightarrow \epsilon\}$
Corresponding natural deduction rule:

$$\frac{\begin{array}{c} (b1, b2) : B \\ \vdots \\ b1 + \alpha + b2 : C \end{array}}{\alpha : B \downarrow C} I\downarrow$$

case 5 $\lfloor A \uparrow B \rfloor = \lceil B \rceil \cup \lfloor A \rfloor \cup \{T_{A \uparrow B}(X\#Y) \rightarrow T_A(X[b]Z)\} \cup \{T_B([b]) \rightarrow \epsilon\}$
Corresponding natural deduction rule:

$$\frac{\begin{array}{c} c : C \\ \vdots \\ \alpha 1 + c + \alpha 2 : B \end{array}}{(\alpha 1, \alpha 2) : B \uparrow C} I\uparrow$$

case 6 $\lfloor A \odot B \rfloor = \lfloor A \rfloor \cup \lfloor B \rfloor \cup \{T_{A \odot B}(XYZ) \rightarrow T_A(X\#Z), T_B(Y)\}$
Corresponding natural deduction rule:

$$\frac{\begin{array}{cc} \vdots & \vdots \\ (\alpha 1, \alpha 2) : C & \gamma : B \end{array}}{\alpha 1 + \gamma + \alpha 2 : C \odot B} I\odot$$

Types with J:
If there are types containing J, we add the axiom rule:
$T_J(\#) \rightarrow \epsilon$
This rule is simply the equivalent of the rule 1: J in natural deduction. We are essentially acting as if 1: J was in the displacement grammars lexicon, which it might as well have been.
Types with I:
When adding a hypothetical item of type I, we could add a $[i]$ to the string, and later make appropriate clauses that deal with it, for example:
$T_A(X[i]) \rightarrow T_A(X)$
However, we will instead make use of the fact that $A \bullet I = A$ for any type A, and $\alpha + \epsilon = \alpha$ for any string α. Therefore instead of using a hypothetical item $[i]$ for I, we will simply use ϵ for simplicity.

4.3 Hypothetical items

When items need to be added that are not in the lexicon, we add a hypothetical item (for example $[n]$ for a hypothetical item with type N). The same rules that apply in **D** for the types of these items, should apply to our PRCG G_T so that it generates the same strings and remains strongly equivalent. Therefore, after our initial transformation of each lexical item, we treat each added hypothetical item *as if it were an item in the lexicon* (or, one could say we add this to a virtual lexicon and apply the transformation to the virtual lexicon). We then apply the normal transformation procedure to these items, which means we might then add more hypothetical items, and so on. Important to note here is that the total amount of hypothetical items that will have to be dealt with in this manner will always be finite, since with each new clause the number of total connectives that would be left in the natural deduction proof would be reduced by one (remember each clause corresponds to one step in a natural deduction proof).

We are now ready to give a formal definition of the transformation function.
$G_S = (\Sigma, \delta, S)$
$G_T = (N, T, V, T_S, P)$ where:
$P = \bigcup_{(a,A)\in\delta} \{T_A(a) \to \epsilon\} \cup \lceil A \rceil \cup pcheck(A, S)$

$\lceil A_{atomic} \rceil = \emptyset$
$\lceil A\backslash B \rceil = \lceil B \rceil \cup \lfloor A \rfloor \cup \{T_B(XY) \to T_A(X), T_{A\backslash B}(Y)\}$
$\lceil A/B \rceil = \lceil A \rceil \cup \lfloor B \rfloor \cup \{T_A(XY) \to T_{A/B}(X), T_B(Y)\}$
$\lceil A \bullet B \rceil = \{S(XYZ) \to T_{A\bullet B}(Y), S(X[a][b]Z) | c \in \Sigma \wedge \delta(c, A \bullet B)\} \cup \{T_A([a]) \to \epsilon\} \cup \{T_B([b]) \to \epsilon\} \cup \lceil A \rceil \cup \lceil B \rceil$
$\lceil A \downarrow B \rceil = \lceil B \rceil \cup \lfloor A \rfloor \cup \{T_B(XYZ) \to T_{A\downarrow B}(Y), T_A(X\#Z)\}$
$\lceil A \uparrow B \rceil = \lceil A \rceil \cup \lfloor B \rfloor \cup \{T_A(XYZ) \to T_{A\uparrow B}(X\#Z), T_B(Y)\}$
$\lceil A \odot B \rceil = \{S(XYZ) \to T_{A\odot B}(Y), S(X[a1][b][a2]Z) | c \in \Sigma \wedge \delta(c, A \odot B)\} \cup \{T_A([a1]\#[a2]) \to \epsilon\} \cup \{T_B([b]) \to \epsilon\} \cup \lceil A \rceil \cup \lceil B \rceil$
$\lfloor A_{atomic} \rfloor = \emptyset$
$\lfloor A\backslash B \rfloor = \lfloor B \rfloor \cup \lceil A \rceil \cup \{T_{A\backslash B}(X) \to T_B([a]X)\} \cup \{T_A([a]) \to \epsilon\}$
$\lfloor A/B \rfloor = \lfloor A \rfloor \cup \lceil B \rceil \cup \{T_{A/B}(X) \to T_A(X[b])\} \cup \{T_B([b]) \to \epsilon\}$
$\lfloor A \bullet B \rfloor = \lfloor A \rfloor \cup \lfloor B \rfloor \cup \{T_{A\bullet B} \to T_A(X), T_B(Y)\}$
$\lfloor A \downarrow B \rfloor = \lceil A \rceil \cup \lfloor B \rfloor \cup \{T_{A\downarrow B}(X) \to T_B([a1]X[a2])\} \cup \{T_A([a1]\#[a2]) \to \epsilon\}$
$\lfloor A \uparrow B \rfloor = \lceil B \rceil \cup \lfloor A \rfloor \cup \{T_{A\uparrow B}(X\#Y) \to T_A(X[b]Z)\} \cup \{T_B([b]) \to \epsilon\}$
$\lfloor A \odot B \rfloor = \lfloor A \rfloor \cup \lfloor B \rfloor \cup \{T_{A\odot B}(XYZ) \to T_A(X\#Z), T_B(Y)\}$

$V = \{X, Y, Z\}$
$T = \{a | \exists a(\delta(a, A)) \vee (\{T_A(X) \to \epsilon\} \subseteq P \wedge \{a\} \subseteq \{X\})\}$
$N = \{T_A | (\{T_A(X_0 ... X_n) \to ...\} \subseteq P) \wedge (\{X_0 ... X_n\} \subset (T \cup V)^*)\}$

$$\begin{aligned}
pcheck(A/B, C) &= pcheck(A, B)\\
pcheck(A\backslash B, C) &= pcheck(B, A)\\
pcheck(A \bullet B, C) &= pcheck(A, C) \cup pcheck(B, C) \cup T_C(XYZ) \to T_{A\bullet B}(Y), T_C(X[a][b]Z)\\
pcheck(A \downarrow B, C) &= pcheck(B, A)\\
pcheck(A \uparrow B, C) &= pcheck(A, B)\\
pcheck(A \odot B, C) &= pcheck(A, C) \cup pcheck(B, C) \cup T_C(XYZ) \to T_C(X[a1][b][a2]Y), T_{A\odot B}(Y)\\
pcheck(A_{atomic}, C) &= \emptyset
\end{aligned}$$

$$\textit{a pair of types A and C} \longrightarrow \textit{set of clauses}$$

Graphical representation of the function pcheck, that deals with the (discontinuous) product units in (all the subtypes of) some type A, using C to keep track of what the goal type would be.*

4.4 Multiple separators

So far, we have been working with displacement grammars containing up to one separator. However, the transformation could work with any number of separators. All we would need to do is assign a number to each of the added gaps, linking them to a perticular separator:
$\lceil A \downarrow_n B \rceil = \lceil B \rceil \cup \lfloor A \rfloor \cup \{T_B(XYZ) \to T_{A\downarrow_n B}(Y), T_A(X\#_n Z)\}$
This can be applied to each of the rules above. However, since it does not seem common in natural language to have multiple separators, it is not a main focus of this paper.

4.5 Examples

What follows is a few examples of how the transformation works.

First, we present an example of three counting dependencies, this is a TAL [2].
$\{a^n b^n c^n | n > 0\}$
Lexicon:

$$\begin{aligned}
b &: J\backslash B\\
b &: J\backslash(A \downarrow B)\\
c &: B\backslash C\\
a &: A/C\\
\textit{Distinguished type} &: A \odot I
\end{aligned}$$

Transforming the 4 items in lexicon and the distinguished type yields the following clauses:

$$
\begin{aligned}
T_{J\backslash B}(b) &\to \epsilon \\
T_B(XY) &\to T_J(X), T_{J\backslash B}(Y) \\
T_{J\backslash(A\downarrow B)}(b) &\to \epsilon \\
T_{A\downarrow B}(XY) &\to T_J(X), T_{J\backslash(A\downarrow B)}(Y) \\
T_B(XYZ) &\to T_{A\downarrow B}(Y), T_A(X\#Z) \\
T_{B\backslash C}(c) &\to \epsilon \\
T_C(XY) &\to T_B(X), T_{B\backslash C}(Y) \\
T_{A/C}(a) &\to \epsilon \\
T_A(XY) &\to T_{A/C}(X), T_C(T) \\
T_J(\#) &\to \epsilon \\
T'_A(XY) &\to T_A(X\#Y)
\end{aligned}
$$

Next, we present an example using the copy-language. This is another example of a language that can be generated by a TAG, however it does have cross-serial dependencies:
$\{ww|w \in \{a,b\}^+\}$
Lexicon:

$$
\begin{aligned}
&a : J\backslash(A\backslash S) \\
&a : J\backslash(S \downarrow (A\backslash S)) \\
&a : A \\
&b : J\backslash(B\backslash S) \\
&b : J\backslash(S \downarrow (B\backslash S)) \\
&b : B \textit{Distinguished type} \qquad : S \odot I
\end{aligned}
$$

Transforming this grammar yields the following clauses:

$$
\begin{aligned}
T_{J\backslash(A\backslash S)}(a) &\to \epsilon \\
T_{A\backslash S}(XY) &\to T_J(X), T_{J\backslash(A\backslash S)}(Y) \\
T_S(XY) &\to T_A(X), T_{A\backslash S}(Y) \\
T_{J\backslash(S\downarrow(A\backslash S))}(a) &\to \epsilon \\
T_{S\downarrow(A\backslash S)}(XY) &\to T_J(X), T_{J\backslash(S\downarrow(A\backslash S))} \\
T_{A\backslash S}(XYZ) &\to T_{S\downarrow(A\backslash S)}(Y), T_S(X\#Z) \\
T_A(a) &\to \epsilon \\
T_{J\backslash(B\backslash S)}(b) &\to \epsilon \\
T_{B\backslash S}(XY) &\to T_J(X), T_{J\backslash(B\backslash S)}(Y) \\
T_S(XY) &\to T_B(X), T_{B\backslash S}(Y) \\
T_{J\backslash(S\downarrow(B\backslash S))}(b) &\to \epsilon \\
T_{S\downarrow(B\backslash S)}(XY) &\to T_J(X), T_{J\backslash(S\downarrow(B\backslash S))} \\
T_{B\backslash S}(XYZ) &\to T_{S\downarrow(B\backslash S)}(Y), T_S(X\#Z) \\
T_B(b) &\to \epsilon \\
T_J(\#) &\to \epsilon \\
T'_S(XY) &\to T_S(X\#Y)
\end{aligned}
$$

Next, we will transform MIX_3. This language has interesting properties (free word order). This language is not a TAG [9].
$MIX_3 = \{w \in \{a, b, c\}^+ | \ \ |w|_a = |w|_b = |w|_c\}$
Lexicon:

$$
\begin{aligned}
a &: A \\
a &: (S \uparrow I) \downarrow A \\
b &: (A \uparrow I) \downarrow B \\
c &: (B \uparrow I) \downarrow S \\
\mathit{Distinguished\ type} &: S
\end{aligned}
$$

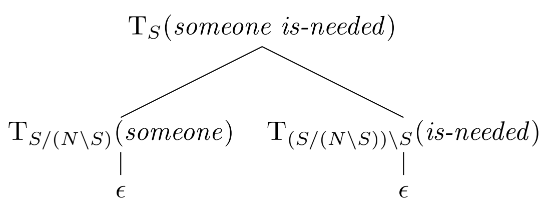
Transforming this grammar yields the following clauses:

$$
\begin{aligned}
T_A(a) &\to \epsilon \\
T_{(S\uparrow I)\downarrow A}(a) &\to \epsilon \\
T_A(XYZ) &\to T_{(S\uparrow I)\downarrow A}(Y), T_{S\uparrow I}(X\#Z) \\
T_{S\uparrow I}(X\#Z) &\to T_S(X\epsilon Z) \\
T_{(A\uparrow I)\downarrow B}(b) &\to \epsilon \\
T_B(XYZ) &\to T_{(A\uparrow I)\downarrow B}(Y), T_{A\uparrow I}(X\#Z) \\
T_{A\uparrow I}(X\#Z) &\to T_A(X\epsilon Z) \\
T_{(B\uparrow I)\downarrow S}(c) &\to \epsilon \\
T_S(XYZ) &\to T_{(B\uparrow I)\downarrow S}(Y), T_{B\uparrow I}(X\#Z) \\
T_{B\uparrow I}(X\#Z) &\to T_B(X\epsilon Z)
\end{aligned}
$$

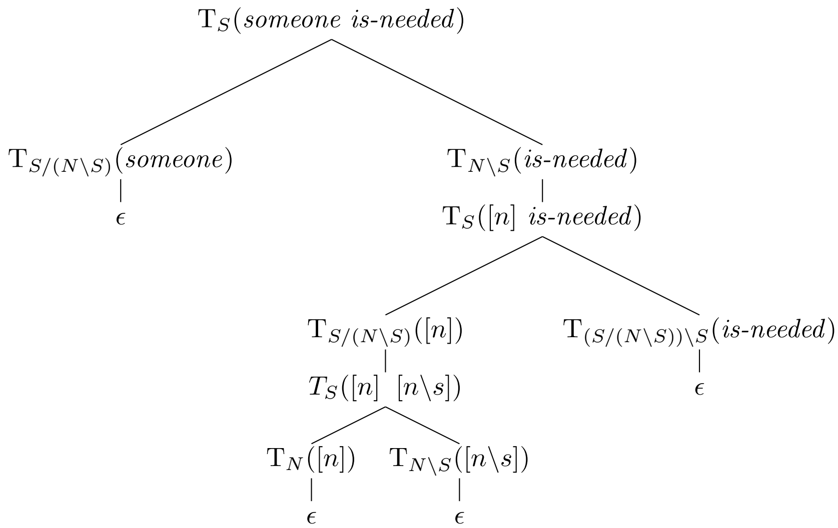
Next, we present an example that shows how ambiguity is preserved after the transformation using our toy grammar for *someone is-needed*.
Lexicon:
$someone : S/(N\backslash S)$
$is - needed : (S/(N\backslash S))\backslash S$
Distinguished type: S (sort 0)
Transforming this grammar yields the following clauses:

$$
\begin{aligned}
T_{S/(N\backslash S)}(someone) &\to \epsilon \\
T_S(XY) &\to T_{S/(N\backslash S)}(X), T_{N\backslash S}(Y) \\
T_{N\backslash S}(X) &\to T_S([n]X) \\
T_{(S/(N\backslash S))\backslash S}(\textit{is-needed}) &\to \epsilon \\
T_S(XY) &\to T_{S/(N\backslash S)}(X), T_{(S/(N\backslash S))\backslash S}(Y) \\
T_S(XY) &\to T_N(X), T_{N\backslash S}(Y) \\
T_{S/(N\backslash S)}(X) &\to T_S(X[n\backslash s]) \\
T_{N\backslash S}([n\backslash s]) &\to \epsilon \\
T_N([n]) &\to \epsilon
\end{aligned}
$$

One way to derive *someone is-needed* in G is:



Another way would be:



5 Conclusions

We have shown that displacement grammars can be transformed into Range Concatenation Grammars, using the transformation described in this paper. The target RCG (G_T) requires combinatory clauses. However, each clause mimics the actions of a natural deduction rules as used in ordinary displacement grammars. This means resource sensitivity has been preserved. This means that either these grammars could be transformed into SRCGs somehow, or that there is some sort of "middle ground" where combinatory clauses *are* allowed, and yet the grammar as a whole remains resource sensitive. We also believe the translation proposed here can be made even more direct by transforming the input and output side of a statement at the initial state in each natural deduction rule, into the first and second block of the left side of an RCG clause. The input and output at the final state of the natural deduction rule could then be transformed into the first and second block of the right side of the RCG clause. In this manner, a 2-RCG very close to a *simple* 2-RCG could be used for G_T. The only difference would be that we would need to allow certain combinatory clauses (only those that add the separator symbol #). This further supports the idea that resource sensitivity could be maintained while still allowing combinatory clauses.

References

[1] Chomsky, Noam, *Three models for the description of language.* Information Theory, IRE Trans. on Information Theory 2 (1956), p113-124.

[2] Kallmeyer, Laura, *Parsing Beyond Context-Free Grammars*. Cognitive Technologies 1st edition (2010).

[3] Kuhlmann, Marco, *Dependency Structures and Lexicalized Grammars: An Algebraic Approach*. Springer (1998).

[4] Morrill, Glyn & Valentin, Oriol, *On Calculus of Displacement* TAG+10, Proceedings of TAG+Related Formalisms 2010, University of Yale (2010).

[5] Boullier, Pierre, *Proposal for a Natural Language Processing Syntactic Backbone*. Rapport de recherche n 3442, (1998). ISSN 0249-6399.

[6] Morrill, Glyn & Valentin, Oriol & Fadda, Mario, *The Displacement Calculus*. Journal of Logic, Language and Information, Volume 20 Issue 1, January 2011. Pages 1-48.

[7] Wijnholds, Gijs, *Investigations into Categorial Grammar: Symmetric Pregroup Grammar and Displacement Calculus*. Bachelor Thesis 2011, University of Utrecht.
http://igitur-archive.library.uu.nl/student-theses/2011-0809-200850/UUindex.html

[8] Morrill, Glyn & Merenciano, Josep-Maria, *Generalising Discontinuity*. T.A.L., volume 37, n 2, p.119-143.

[9] Kanazawa, Makoto & Salvati, Sylvain, *MIX is not a tree-adjoining language*. Proceedings of the 50th Annual Meeting of the Assiociation for Computational Linguistics.