Encoding m-Multiple Context-free Grammars to Abstract Categorial Grammars

Michiel de Winter

In this paper I will give a clear explanation of the encoding of an MCFG expressing the shuffle language L2s into an ACG. Step by step I explain what happens, and try to give an idea of why this is done. With this I hope to give more clarity than the original papers on this topic.

Introduction

Abstract Categorial Grammars (ACG) were introduced by De Groote in 2001 as a new categorical formalism, based on linear logic (de Groote, 2001). ACG's have the property of generating two languages, the abstract and the object language. The first gives a set of grammatical structures and the latter a set of concrete forms. A lexicon gives the encoding from abstract to object language. The formalism was not introduced to compete with existing formalisms but rather to give a framework in which they could be encoded (de Groote, 2002), thus enabling us to show similarities and differences between formalisms and their expressivity.

This encoding of formalisms has already been done for TAG (de Groote, 2002) and m-LCFRS (de Groote and Pogodalla, 2003). These papers are very technical, though, and hard to read. With this paper, I will give an example encoding of an m-MCFG, which is in fact equivalent to the m-LCFRS into an ACG step by step to show more clearly how this is done.

Shuffle language L_{2s}

Our example will be the shuffle language L_{s2} as introduced in (Dost, 2011). This language consists of strings of brackets of two types (e.g. "( )" and "[ ]"), that can be shuffled as long as a type only has closing brackets after enough opening brackets. For reasons of convenience we will use "d b" and "q p" respectively to represent the two kinds of brackets. Formally, our language will look like this:

$L_{s2} = \{ \{d,b,q,p\}^*$ such that $|w|_d = |w|_b$, for each prefix w' of w $|w'|_d \geq |w'|_b$
$|w|_q = |w|_p$, for each prefix w' of w $|w'|_q \geq |w'|_p \}$

Possible derivations of this language are dbqp and qdbp but not pq, qppq and qdppbq.

Abstract Categorial Grammars

I will give a general idea here of how ACG's work. For full formal definitions, see (de Groote, 2001), (de Groote, 2002) and (de Groote and Pogodalla, 2003). As said above, an ACG consists of an abstract language, an object language and a lexicon. These languages are built by higher-order linear signatures, being triples $\Sigma = \langle A,C,t \rangle$, where A is the finite set of atomic types, C the finite set of constants and t the function that assigns a linear implicative type to each constant in C. Given these two languages $\Sigma_1 = \langle A_1,C_1,t_1 \rangle$ and $\Sigma_2 = \langle A_2,C_2,t_2 \rangle$, a lexicon $L : \Sigma_1 \rightarrow \Sigma_2$ is the interpretation of the abstract language into the object language. This lexicon forms a pair $L = \langle F,G \rangle$ so that $F: A_1 \rightarrow T(A_2)$ is the interpretation of atomic types of Σ_1 as types built upon A_2, and $G: C_1 \rightarrow \Lambda(\Sigma_2)$ the interpretation of the constants of Σ_1 as λ-terms built upon Σ_2.

An abstract categorial grammar itself now forms a quadruple $G = \langle \Sigma_1,\Sigma_2,L,s \rangle$, with Σ_1 and Σ_2 being the mentioned abstract and object vocabulary respectively, *L* the lexicon from the first to the latter and s an atomic type from the abstract vocabulary, as a start symbol.

m-Multiple Context-free Grammars

We will now take a look at m-MCFG's and also give an 2-MCFG for our example language L_{s2}. An MCFG is defined as a 5-tuple $G = \langle N,T,F,R,S \rangle$ where:

- N is the finite set of non-terminals
- T is the finite set of terminals
- F is the set of mcf-functions, with the restriction that each component on the right hand side may not appear more than once.
- P is a finite set of rules
- $S \in N$ is the start symbol.

Other than normal context-free grammars, an multiple context-free grammars deals with tuples of strings, rather than single strings. Because these tuples do not necessarily have to be well-nested, MCFG's are capable of handling crossed dependencies, which are, as we will see, the case with L_{2s}. An MCFG has dimension m if it has a maximum predicate dimension of m. In other words, predicated of an 2-MCFG can have no more arguments than 2. For our example we will use the implementation given by (Dost, 2011), written as a simple Range Concatenation Grammar. This formalism is equivalent to MCFG, but its notation is closer related to that of the Prolog programming language, making it easier to implement.

Our MCFG will now be the following:

N = { S, A, Y }
T = { d, b, q, p }

r0: S (x1 x2) → A (x1 , x2)
r1: Y (d, b) → ε
r2: Y (q, p) → ε
r3: A (ε, ε) → ε
r4: A (x1 x2 , y1 y2) → Y (x1 , x2) , A (y1 , y2)
r5: A (x1 y1 , x2 y2) → Y (x1 , x2) , A (y1 , y2)
r6: A (x1 y1 , y2 x2) → Y (x1 , x2) , A (y1 , y2)
r7: A (x1 , x2 y1 y2) → Y (x1 , x2) , A (y1 , y2)
r8: A (x1 , y1 x2 y2) → Y (x1 , x2) , A (y1 , y2)
r9: A (x1 , y1 y2 x2) → Y (x1 , x2) , A (y1 , y2)
r10: A (x1 x2 y1 , y2) → Y (x1 , x2) , A (y1 , y2)
r11: A (x1 y1 x2 , y2) → Y (x1 , x2) , A (y1 , y2)
r12: A (x1 y1 y2 , x2) → Y (x1 , x2) , A (y1 , y2)

As we can see, r0 concatenates two strings given by A to be the final output. Predicate A can be either empty, or an concatenation of two tuples, of which one is recursive. Predicate Y puts in the constant symbols. The use of this predicate Y avoids redundancy, because without it we would have to duplicate rules r4 – r12 for the cases of d/b and q/p. Furthermore, the three different separations of the strings on the left hand side are needed to enable more complex bracketing around other substrings.

Encoding L_{2s} into ACG

Now that we have the basics of the Abstract Categorial Grammar and an 2-MCFG expressing L_{2s} we can start with encoding. We start with the abstract language, $\Sigma_1 = \langle A_1, C_1, t_1 \rangle$:

A_1 : The atomic symbols of the abstract language consist of the non-terminal symbols of the MCFG. (de Groote and Pogodalla, 2003) suggest to add an extra start symbol S', to get from higher-order functions back to strings. In our example, though, this is unnecessary because there are no recursive rules for S, and all strings are concatenated by this predicate. We can therefore say A_1 = { S, A, Y }

C_1 : The constants of the object language consist of a one-to-one correspondence of R, which gives C_1 = { r0, … , r12 }

t_1` : t assigns a linear implicative type to each constant c in C_1. These being the rules of the MCFG, t_1 gives the actual syntactic structure of the language. Therefore t_1 is such that:

t_1(r0) = A -o S
t_1(r1) = Y
t_1(r2) = Y
t_1(r3) = A
t_1(r4-12) = Y -o (A -o A)

The object language now is fairly simple. It mostly gives the notation we will have as output, and some semantic interpretation. $\Sigma_2 = \langle A_2, C_2, t_2 \rangle$ will be as follows:

A_2 : {σ}
C_2 : the constants of the object language are the actual strings, and therefore the same as the set T of terminal symbols from the MCFG: { d, b, q, p }
t_2 : the function t of the object language assigns type σ to every constant c in C_2.

Finally we have to define the lexicon *L*. First we assign types to the atomic types of Σ_1. We know that S only produces a single string. Furthermore, we know that both Y and A take two strings, from which we can abstract, to produce a string. Given the atomic type σ of Σ_2, we have:

L(S) = σ (single string)
L(A) = (σ --o σ --o σ) --o σ (take two strings, and abstract to produce a string)
L(Y) = (σ --o σ --o σ) --o σ (Idem)

This assignment is not very well explained, though, and remains hard to understand. After this, we build lambda-terms such that they represent the rules of the MCFG, and thus follow the structure given in t_1:

L(r0) = $\lambda P.\ P(\lambda x_1 x_2.\ (x_1 + x_2))$
L(r1) = $\lambda f.\ f(d, b)$
L(r2) = $\lambda f.\ f(q, p)$
L(r3) = $\lambda f.\ f(\varepsilon, \varepsilon)$
L(r4) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1 + x_2)\ (y_1 + y_2)))$
L(r5) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1 + y_1)\ (x_2 + y_2)))$
L(r6) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1 + y_1)\ (y_2 + x_2)))$
L(r7) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1)\ (x_2 + y_1 + y_2)))$
L(r8) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1)\ (y_1 + x_2 + y_2)))$
L(r9) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1)\ (y_1 + y_2 + x_2)))$
L(r10) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1 + x_2 + y_1)\ (y_2)))$
L(11) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1 + y_1 + x_2)\ (y_2)))$
L(r12) = $\lambda PQ.\ P(\lambda x_1 x_2.\ Q(y_1 y_2.\ (x_1 + y_1 + y_2)\ (x_2)))$

We can now see that *L*(r0) has one predicate containing two strings that are concatenated (+) to become one single string. *L*(r1,r2,r3) give functions that put symbols (empty or non-empty) in. *L*(r4-r12) show two predicates, both containing two strings. These four strings are concatenated into two longer strings, in the nine variants argued to be necessary earlier.

With the ACG given above, we can now compute output of the shuffle language. Based on t_1 we know which rules we can apply on each other. For example we can now computer the following program:

$$
\begin{aligned}
L(\text{r0}(\text{r4}(\text{r2 r11 (r1 r4(r2 r3)))))}) &= L(\text{r0}(\text{r4}(\text{r2 r11 (r1 r4}((q,p), \varepsilon))))) \\
&= L(\text{r0}(\text{r4}(\text{r2 r11 ((d, b) (q+p)}, \varepsilon)))) \\
&= L(\text{r0}(\text{r4}((q,p), ((d+(q+p)+b), \varepsilon)))) \\
&= L(\text{r0}((q+p), ((d+(q+p)+b) + \varepsilon))) \\
&= L((q+p)+(d+(q+p)+b)+\varepsilon) \\
&= q+p+d+q+p+b
\end{aligned}
$$

Conclusion

We have seen step by step a method to encode an 2-MCFG into an ACG. Although we did not go into much detail of formal definitions, a clear view on how to do this encoding should be done should now be given. A difficult to understand part remains the assigning of types, especially that of the atomic types of the abstract language.

References

- Dost, S. (2011): Shuffle language in Minimalist Grammar, *paper for course "Computationele Grammatica's", Universiteit Utrecht*
- de Groote, P. (2001): Towards Abstract Categorial Grammars, in *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, p. 148-155
- de Groote, P. (2002): Tree-adjoining grammars as abstract categorial grammars. In *TAG+6, Proceedings of the sixth International Workshop on Tree AdjoiningGrammars and Related Frameworks*, p. 145–150
- de Groote, P. and Pogodalla, S. (2003): m-linear CF Rewriting Systems as Abstract CGs, In *Proceedings of Mathematics of Language 8, R.T. Oehrle & J. Rogers (editors)*, chapter 8
- Kallmeyer, L. (2010): Parsing beyond Context-free Grammars, *Springer-Verlag, Berlin Heidelberg*
- Kuhlmann, M. (2010): Dependency Structures and Lexicalized Grammars: an algebraic approach,, *Springer-verlag, Berlin Heidelberg*