Parsing in second order ACGs

Clovis Eberhart

Contents

Introduction

Natural Language Processing (NLP) is a very wide field of study that has to do with natural language, how it can be understood or generated, how information can be dug from a text in a book, how grammatical structures turn into concrete sentences...

In NLP, there exists many formalisms to describe grammatical structures and how these structures are transformed into actual sentences and a meaning can be infered from them.

In 2001, Philippe de Groote [1] introduced a new formalism called Abstract Categorial Grammars that aims at unifying most of the other formalisms under a general and very simple formalism to describe grammatical structures and how they are transformed into syntactical structures.

The problem that we face here is the problem of parsing in second-order (a certain class of) Abstract Categorial Grammars and implementation of the corresponding algorithm. Parsing is the problem of having a concrete sentence and trying to find the grammatical structure behind it.

This problem has been solved by Makoto Kanazawa [3], and it reduces the problem of parsing in a second-order ACG to the problem of solving a query in *Datalog*, which is decidable. What I did was implement this reduction to solve this problem of parsing.

We will first inroduce the notion of Abstract Categorial Grammars and give an example, then study in detail parsing in second-order Abstract Categorial Grammars and how it is reduced to *Datalog* solving, and finally we will see the limits of this algorithm and a possible application.

Chapter 1

Abstract Categorial Grammars

1.1 Definition

Abstract Categorial Grammars (ACGs) is a formalism that describes the interface between grammatical structures and syntax in natural language. It is based on linear logic and the associated linear λ-calculus.

ACGs are based on linear implicative types on a set of atomic types A, defined by:

$$\mathcal{I}(A) ::= A \mid \mathcal{I}(A) \multimap \mathcal{I}(A)$$

The rules to deduce a formula in linear logic are:

$$\frac{}{A \vdash A}\ (Ax) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ (Abs) \qquad \frac{\Gamma \vdash A \quad \Delta \vdash A \multimap B}{\Gamma, \Delta \vdash B}\ (App)$$

Note that there is no weakening rule, i.e. $\Gamma \vdash A$ does not imply $\Gamma, \Delta \vdash A$, and that the sign $\multimap$ denotes linear implication. This can be seen as *resources-wary* logics as every hypothesis has to be used once and only once.

A higher-order signature Σ is a triple $(A,\ C,\ \tau)$ where A is a finite set of atomic types, C is a finite set of constants and τ is a mapping from C to $\mathcal{I}(A)$.

The intuition behind the definition of a higher order signature is that, on a grammatical point of view, constants in C correspond to basic grammatical constructions and their type $\tau(C)$ corresponds to how they interact. We want to describe the interaction between words by use of linear logic, a sentence is well-structured when one is able to deduce S (sentence) by use of linear logic on the types of the words in the sentence. Here is a concrete example of a higher order signature:

We have a higher-order signature $\Sigma_1 = (A_1, C_1, \tau_1)$, where $A_1 = \{N, NP, S\}$, $C_1 = \{Jean, mange, une, pomme\}$ and $\tau_1 = \{Jean \mapsto NP, mange \mapsto NP \multimap NP \multimap S, une \mapsto N \multimap NP, pomme \mapsto N\}$. N stands for noun, NP for noun phrase and S for sentence. *Jean* is a constant that correspond to the French name 'Jean', which is indeed a noun phrase, *mange* is a constant that corresponds to the French verb form 'mange', which can be seen as a function that takes two noun phrases (its subject and object) and returns a sentence, hence the type $NP \multimap S$, *une* represents 'une', which can be seen as a function that takes a noun and returns a noun phrase, and *pomme* represents 'pomme', which is a noun.

If X is an infinite set of λ-variables, the set $\Lambda(\Sigma)$ of linear λ-terms built on $\Sigma = (A, C, \tau)$ is defined by:

- $x \in X$, then $x \in \Lambda(\Sigma)$
- if $c \in C$, then $c \in \Lambda(\Sigma)$
- if $x \in X$ occurs free in $t \in \Lambda(\Sigma)$ exactly once, then $\lambda x \cdot t \in \Lambda(\Sigma)$
- if $t, u \in \Lambda(\Sigma)$ and free variables of t and u are disjoint, then $tu \in \Lambda(\Sigma)$

The rules to type linear λ-terms built on Σ are:

$$\frac{c \in C}{\vdash_\Sigma c : \tau(c)}\ (cons) \qquad \frac{x \in X}{x : \alpha \vdash_\Sigma x : \alpha}\ (var)$$

$$\frac{\Gamma, x : \alpha \vdash_\Sigma t : \beta}{\Gamma \vdash_\Sigma \lambda x \cdot t : \alpha \multimap \beta}\ (abs) \qquad \frac{\Gamma \vdash_\Sigma t : \alpha \multimap \beta \quad \Delta \vdash_\Sigma u : \alpha}{\Gamma, \Delta \vdash_\Sigma tu : \beta}\ (app)$$

A lexicon $\mathcal{L}$ from a signature $\Sigma_1 = (A_1, C_1, \tau_1)$ to a signature $\Sigma_2 = (A_2, C_2, \tau_2)$ is a couple (F, G) where:

- F is a function from A_1 to $\mathcal{I}(A_2)$
- G is a function from C_1 to $\Lambda(\Sigma_2)$
- F and G are compatible with the typing relation, i.e.:
  $\forall c \in C_1, \vdash_{\Sigma_2} G(c) : \hat{F}(\tau_1(c))$, where $\hat{F}$ is the only homomorphism that extends F.

From here on, $\mathcal{L}$ will also denote F and G in contexts where the meaning is clear.

An ACG a quadruple $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, s)$ where $\mathcal{L}$ is a lexicon from Σ_1 to Σ_2 and $s \in \Sigma_1$ is a distinguished type. Σ_1 is called the *abstract vocabulary* and represents the abstract sturcture of the grammar, Σ_2 is called the *object vocabulary* and represents the syntax of a language. An ACG produces two languages :

- the *abstract language* $\mathcal{A}(\mathcal{G}) = \{t \in \Lambda(\Sigma_1) | \vdash_{\Sigma_1} t : s\}$ that represents all valid grammatical strucutures of a language
- the *object language* $\mathcal{O}(\mathcal{G}) = \mathcal{L}(\mathcal{A}(\mathcal{G}))$ that represents the concrete realisation of these abstract structures

1.2 Example

Here is a concrete example of an ACG for a very tiny part of French grammar to show how to check that a sentence is valid (meaninig that it's part of the object language):

Σ_1 is defined as in the example above, and $\Sigma_2 = (A_2, C_2, \tau_2)$ is defined by $A_2 = \{string\}$, $C_2 = \{+, \mathsf{Jean}, \mathsf{mange}, \mathsf{une}, \mathsf{pomme}\}$ and $\tau_2 = \{+ \mapsto string \multimap string \multimap string, \mathsf{Jean} \mapsto string, \mathsf{mange} \mapsto string, \mathsf{une} \mapsto string, \mathsf{pomme} \mapsto string\}$ (where $+$ represents the concatenation operator, and will be used with an infix notation for clarity). We define $\mathcal{L} = \{F = \{NP \mapsto string, N \mapsto string, S \mapsto string\}, G = \{Jean \mapsto \mathsf{Jean}, mange \mapsto \lambda x, y \cdot x + \mathsf{mange} + y, une \mapsto \lambda x \cdot \mathsf{une} + x, pomme \mapsto \mathsf{pomme}\}\}$. The distinguished type is S.

The linear λ-term *mange Jean* (*une pomme*) is a valid grammatical structure, since *mange* has type $NP \multimap NP \multimap S$, *Jean* has type NP, hence $\vdash_{\Sigma_1} mange\ Jean : NP \multimap S$, *une* has type $N \multimap NP$, *pomme* has type N, hence $\vdash_{\Sigma_1} une\ pomme : NP$ and $\vdash_{\Sigma_1} mange\ Jean\ (une\ pomme) : S$. The concrete structure corresponding to it is $\mathcal{L}(mange\ Jean\ (une\ pomme)) = (\lambda x, y \cdot x + \mathsf{mange} + y)\ \mathsf{Jean}\ ((\lambda x \cdot \mathsf{une} + x)\ \mathsf{pomme}) \rightarrow_\beta^* \mathsf{Jean} + \mathsf{mange} + \mathsf{une} + \mathsf{pomme}$. Therefore, $\mathsf{Jean} + \mathsf{mange} + \mathsf{une} + \mathsf{pomme}$ is a valid sentence in French.

On the other hand, the following sentences are not valid in French:

- $\mathsf{une} + \mathsf{Jean} + \mathsf{pomme} + \mathsf{mange}$, because there is no linear λ-term t such that $\mathcal{L}(t) \rightarrow_\beta^* \mathsf{une} + \mathsf{Jean} + \mathsf{pomme} + \mathsf{mange}$

- mange, because any λ-term t such that $\mathcal{L}(t) \rightarrow_\beta^* \mathsf{mange}$ is of type $NP \multimap NP \multimap S$

Although the natural idea for the object signature is one that describes the realisation of a grammatical structure in a given language, it can also be the realisation of that grammatical structure as a logical proposition that describes the meaning of the sentence associated to it. It is a very powerful feature of ACGs to be able to treat a grammatical structure's syntax and its semantics the exact same way. Therefore, computing the meaning of a grammatical structure is nearly as easy as computing the syntactic struxture that corresponds to it. Here is an example of a second ACG that correspond to the previous one, but whose object language is the meaning of the grammatical structure (I do not pretend I understand the semantics, I just give an example to show this is handled the very same way):

Σ_1 is as above, $\Sigma_2 = (A_2, C_2, \tau_2)$, with $A_2 = \{e, t\}$, $C_2 = \{John, eat, apple\}$ and $\tau_2 = \{John \mapsto e, eat \mapsto e \multimap e \multimap t, apple \mapsto e \multimap t\}$. We define $\mathcal{L} = \{F = \{NP \mapsto (e \multimap t) \multimap t, N \mapsto e \multimap t, S \mapsto t\}, G = \{Jean \mapsto \lambda P \cdot P\ John, mange \mapsto \lambda P, Q \cdot Q\ (\lambda y \cdot P\ (\lambda x \cdot eat\ x\ y)), une \mapsto \lambda P, Q \cdot \exists x \cdot P\ x \wedge Q\ x, pomme \mapsto \lambda x \cdot apple\ x\}\}$.

Then we have $\mathcal{L}(mange\ Jean\ (une\ pomme)) = (\lambda P, Q \cdot Q\ (\lambda y \cdot P\ (\lambda x \cdot eat\ x\ y)))(\lambda P \cdot P\ John)((\lambda P, Q \cdot \exists x \cdot P\ x \wedge Q\ x)(\lambda x \cdot apple\ x)) \rightarrow^* \exists x \cdot apple\ x \wedge eat\ John\ x$

1.3 Other features

A pleasant and useful feature of ACGs is that they form a category, with signatures as objects and lexica as arrows. This provides, for example, a way to compose ACGs, which can be useful in order to restrict the abstract language of an ACG. Indeed, if too many sentences are valid in an ACG $\mathcal{G}_{12} = (\Sigma_1, \Sigma_2, \mathcal{L}_{12}, s_1)$, it may be useful to find an ACG $\mathcal{G}_{01} = (\Sigma_0, \Sigma_1, \mathcal{L}_{01}, s_0)$ to restrict the abstract (and therefore object) language of $\mathcal{G}_{12}$, and to study $\mathcal{G}_{02} = (\Sigma_0, \Sigma_2, \mathcal{L}_{12} \circ \mathcal{L}_{01}, s_0)$.

An important notion in the ACG theory is the notion of *order* of an ACG, which describes the expressive power of a class of ACGs. It is particularily important to us, as we will only study second-order ACGs. The order of a linear inductive type is defined by induction by:

- $ord(A) = 1$ if A is an atomic type
- $ord(A \multimap B) = \max\{ord(A) + 1, ord(B)\}$

The order of an ACG is defined by $ord(\mathcal{G}) = \max\{ord(\tau(c)) \mid c \in C_1\}$.

First order ACGs are pretty useless, as their abstract languages is defined by the portion of their abstract vocabulary with type s.

Second order ACGs have a much greater expressive power and can be used for some simple grammars. One may define transitive verbs with type $NP \multimap NP \multimap S$, where the first NP is the subject and the second one is the object. For example, *loves* would have this type and be associated with the λ-term $\lambda x, y \cdot x + \mathsf{aime} + y$ in French grammar. $\mathcal{L}(\textit{loves John Mary})$ would then reduce to $\mathsf{Jean} + \mathsf{aime} + \mathsf{Marie}$.

Third order ACGs are really useful, as they can be used to define relative clauses. *who*, for example would have type $(NP \multimap S) \multimap NP \multimap NP$ and be associated with the λ-term $\lambda x, y \cdot x\ (y + \mathsf{qui})$ in French grammar. The following example aims at giving some intuition: $\mathcal{L}(\textit{is John } (\textit{who } (\textit{a man})(\lambda x \cdot \textit{loves } x \textit{ Mary})))$ would then reduce to $\mathsf{Jean} + \mathsf{est} + \mathsf{un} + \mathsf{homme} + \mathsf{qui} + \mathsf{aime} + \mathsf{Marie}$.

Chapter 2

Parsing in ACGs

2.1 Reduction to *Datalog*

The general problem we face here is the problem of parsing in an ACG. Going from the abstract stucture to the object structure (applicative paradigm) is very easy, as we only need to apply a lexicon and β-normalize the result. The problem of parsing is the other way around: we have a term u built on the object signature, and we would like to know whether there exists a term t built on the abstract signature such that $\mathcal{L}(t) \rightarrow_\beta^* u$ (deductive paradigm). This corresponds to trying to inverse a lexicon. This, however, is not easy in the general case, it is not even known whether it is decidable.

In our case, we only consider second-order ACGs, and the problem becomes decidable, as an algorithm has been found by Makoto Kanazawa [4]. It reduces a problem of parsing in second-order ACGs to a problem of trying to infer a fact with *Datalog*, which is decidable with a good complexity in some cases.

Datalog derives facts in first-order logic through the use of Horn clauses. It uses a *Datalog program* and an *extensive database*. The *extensive database* is a set of facts that are true. The *Datalog program* is a set of rules (Horn clauses) used to derive new facts. Such a rule is written $p(x_1, ..., x_n) \coloneq p_1(x_{1,1}, ..., x_{1,n_1}), ..., p_m(x_{m,1}, ..., x_{m,n_m})$, which means that if $p_1(x_{1,1}, ..., x_{1,n_1}), ..., p_m(x_{m,1}, ..., x_{m,n_m})$ are true, then $p(x_1, ..., x_n)$ is also true. There is a Horn clause that corresponds to this statement:

$p(x_1, ..., x_n) \vee \neg p_1(x_{1,1}, ..., x_{1,n_1}) \vee ... \vee \neg p_m(x_{m,1}, ..., x_{m,n_m})$

If $p(x_1, ..., x_n)$ is derivable for a *Datalog* program P and an database D, then we write $P \cup D \vdash p(x_1, ..., x_n)$. The set of predicates that are at the head of a rule is called the *intensional database*.

The sequence of constants in a λ-term $M \in \Lambda(\Sigma)$ when it's read from left to right is noted $\overrightarrow{Con}(M)$. The sequence of its free variables when it's read from left to right is noted $\overrightarrow{FV}(M)$. If M is close and $\overrightarrow{Con}(M) = (c_1, ..., c_n)$, then we note $\widehat{M}[x_1, ..., x_n]$ the λ-term with M where c_i has been replaced by x_i. A typing of M is a derivable judgment $\Gamma \vdash_\Sigma M : \alpha$. M is said to be in *η-long β-normal form relative to* $\Gamma \Rightarrow \alpha$ if it's β-normal and there is a deduction of $\Gamma \vdash_\Sigma M : \alpha$ that is η-long.

To a second-order ACG $\mathcal{G} = (\Sigma, \Sigma', \mathcal{L}, S)$, we associate a *Datalog* program `program`$(\mathcal{G})$, whose intensional database is A and extensional database is C'. We assume that for every

$c \in C, \mathcal{L}(c)$ is in *η-long β-normal form relative to* $\mathcal{L}(\tau(c))$. The arity of $p \in A$ is the number of atomic type occurences in $\mathcal{L}(p)$. The arity of $d \in C'$ is the number of atomic type occurences in $\tau'(d)$. Each constant $c \in C$ gives birth to a single rule ρ_c as follows:

if $\tau(c) = p_1 \to ... \to p_n \to p_0$, $\overrightarrow{Con}(\mathcal{L}(c)) = (d_1, ..., d_m)$ and $y_1 : \beta_1, ..., y_m : \beta_m \vdash \widehat{\mathcal{L}(c)}[y_1, ..., y_m] : \alpha_1 \to ... \to \alpha_n \to \alpha_0$ is a principal typing of $\widehat{\mathcal{L}(c)}[y_1, ..., y_m]$ then:

$\rho_c = p_0(\overline{\alpha_0}) :- p_1(\overline{\alpha_1}), ..., p_n(\overline{\alpha_n}), d_1(\overline{\beta_1}), ..., d_m(\overline{\beta_m})$ where $\overline{\alpha}$ is the sequence of atomic types in α when read from left to right. These atomic types are considered here as *Datalog* variables.

The *Datalog* program associated to $\mathcal{G}$ is $\texttt{program}(\mathcal{G}) = \{\rho_c \mid c \in C\}$.

Let $p \in A$ and $M \in \Lambda(\Sigma')$ be in *η-long β-normal form relative to* $\mathcal{L}(p)$. If $\overrightarrow{Con}(M) = (d_1, ..., d_m)$ and $y_1 : \beta_1, ..., y_m : \beta_m \vdash \widehat{M}[y_1, ..., y_m] : \alpha$ is a principal typing of $\widehat{M}[y_1, ..., y_m]$, then we define:

$\texttt{database}(M) = \{d_i(\overline{\beta_i}) \mid 1 \leqslant i \leqslant m\}$
$\texttt{query}_p(M) = \{p(\overline{\alpha})\}$

Theorem 1 (Kanzawa) *Let $\mathcal{G} = (\Sigma, \Sigma', \mathcal{L}, S)$ be a second-order ACG and $p \in A$. Suppose that $N \in \Lambda(\Sigma')$ is in eta-long beta-normal form relative to $\mathcal{L}(p)$. Then the following are equivalent:*

- *there is $P \in \Lambda(\Sigma)$ such that $\vdash_\Sigma P : p$ and $\mathcal{L}(P) \to^*_\beta N$*
- $\texttt{program}(\mathcal{G}) \cup \texttt{database}(N) \vdash \texttt{query}_p(N)$

2.2 Implementation

The work I had to do during this stage was to implement the reduction of second-order ACGs to *Datalog* programs in *OCaml*. At the beginning of my stage, the prototype for ACGs already included modules to manage λ-terms, higher-order signatures and lexica, modules to manage *Datalog* programs and signatures, and a *Datalog* solver. The modules I had to implement were a module for type inference and a module to transform an ACG into a *Datalog* program.

The difficult parts of this work were to understand the notion of ACGs, to understand the global structure of the code and how all the modules interacted, then to understand an unpublished article by Kanazawa [3] that explained the reduction in a very detailed way and then difficulties about writing code.

In the reduction, the only really difficult part to implement was the type inference algorithm. Hopefully, I had already written one, so I knew how to do it, though the structure of λ-terms was a little complicated (use of de Bruijn indexes).

Another difficult task, though it is not directly related to the fact that I manipulated ACGs, is that I had to change a data structure (how λ-terms were described in the ACG modules) to another structure (how *Datalog* programs are handled) without having written any of these structures.

Although I would have liked to parse a French sentence to have its grammatical structure and then use this structure to infer the sentence's semantics with another ACG, I could not. The reason is that the *Datalog* solver did not return enough information to recontruct the abstract structure corresponding to the sentence (the solver awnsered whether the fact was derivable but did not give its derivation).

I would also have liked to try to parse a logical proposition to find the grammatical structure behind it and then compute the corresponding sentence in French. It was impossible for the same reason as above.

I did obtain some results though. Consider the ACG $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$, where $\Sigma_1 = \{\{S\}, \{R_1, R_2\}, \{R_1 \mapsto S, R_2 \mapsto S \multimap S\}\}$, $\Sigma_2 = \{\{o\}, \{a, b\}, \{a \mapsto o \multimap o, b \mapsto o \multimap o\}\}$ and $\mathcal{L} = \{\{S \mapsto o \multimap o\}, \{R_1 \mapsto \lambda x \cdot x, R_2 \mapsto \lambda x \cdot a + x + b\}\}$ (it is the reduction of $S \rightarrow aSb \mid \epsilon$ to a second-order ACG and $o \multimap o$ represents the string type). I applied the reduction to it and obtained the *Datalog* program:

$\texttt{c.S}(i, i)$.

$\texttt{c.S}(i, j) :- \texttt{t.+}(k, l, m, n, i, j), \texttt{t.a}(k, l), \texttt{t.+}(o, p, q, r, m, n), \texttt{c.S}(o, p), \texttt{t.b}(q, r)$.

which is the reduction of this ACG. The databases and queries I got when applying my implementation of the reduction also corresponded to what the reduction should give.

2.3 Other parsing problems

In the general case, it is not known whether parsing is decidable in an ACG. Sylvain Salvati [5] showed that it is equivalent to decidability of proof research in MELL.

In some other cases, parsing in an ACG remains decidable. For example, if the ACG is lexicalised (the image of every abstract constant through the lexicon contains an object constant), then parsing is decidable. There exists an algorithm, found by Sylvain Salvati [5], that solves the problem. Its complexity, however, is unknown.

One thing that could be very nice would be to have a parser for ACGs of bound order (third order would already be pretty good). One may think of trying to adapt the technique used for second order ACGs for third order or greater ones. Yet, the reduction uses the simple form of skeletons of second order types, which is not the case anymore in third or greater order. Therefore, it seems very unlikely that this technique would adapt to third or greater order ACGs.

Chapter 3

Applications

3.1 Expressive power of second order ACGs

To understand the extent of possible applications of the parser of second order ACGs, we need to know the expressive power of second order ACGs.

The general expressive power of ACGs is unkown. Yet, a number of grammars can be encoded in ACG formalism, thus giving a 'lower bound' of ACGs' expressive power. Here are some grammars that second-order ACGs can encode, the encodings are detailed in [2].

- context-free grammars, whose rules have the form $X \rightarrow w$ where X is a non-terminal symbol and w is a word made of terminal and non terminal symbols. Such a rule can be written in another way, as in the following example : $S \rightarrow aBBS$ can be rewritten $S(aXYZ) :- B(X), B(Y), S(Z)$. The idea here is that you deduce $S(aXYZ)$ from $B(X), B(Y), S(Z)$ (if you can produce X with B, Y with B and Z with S, then you can produce $aXYZ$ with S), a word w is in the language when one can deduce $S(w)$. The first writing is a top-down view, the second one is a bottom-up view.

- multiple context-free grammars, whose rules have the form $X(u_1, ..., u_n) :- X_1(u_{1,1}, ..., u_{1,n_1}), ..., X_m(u_{m,1}, ..., u_{m,n_m})$. These grammars are more powerful than context-free grammars, as the grammar made of the following rules: $S(x_1y_1x_2y_2) :- A(x_1, x_2), B(y_1, y_2)$ $\quad A(\epsilon, \epsilon)$ $\quad A(ax_1, cx_2) :- A(x_1, x_2)$ $\quad B(\epsilon, \epsilon)$ $B(by_1, dy_2) :- B(y_1, y_2)$ generates the language $a^n b^m c^n d^m$.

- tree adjoining grammars is a formalism that has been studied a lot in NLP because it can parse natural language well with two simple operations. This formalism is further discussed in the next section.

- ACGs can also encode linear context-free tree grammars. Here is an example of such a grammar : $S(Y(e, e)) :- A(Y)$ $\quad A(aY(b\square, b\square)) :- A(Y)$ $\quad A(f(\square, \square))$ genereates the language $a^n f(b^n e, b^n e)$ (where parentheses have not been marked when there is a single child and $\square$ is a hole).

- multiple linear context-free grammars are to linear context-free tree grammars what multiple context-free grammars are to context-free grammars. Here is an example of such a grammar : $S(f(X_1, X_2))$:− $A(X_1, X_2)$ $A(g(a, X_1, b), g(c, X_2, d))$:− $A(X_1, X_2)$ $A(e, e)$ generates a language whose leaf language is $a^n eb^n c^n ed^n$.

We have seen that second-order ACGs can encode a lot of different and powerful formalisms, but, in terms of strings, they cannot encode more than this, as states *Salvati's theorem.*

Theorem 2 (Salvati) *The expressive power of second-order string ACGs is the same as that of multiple context-free grammars.*

3.2 Tree-adjoining grammars

Tree-adjoining grammars have been widely studied and used in NLP as they are conjectured to be able to produce natural language while being easily parsable in the general case.

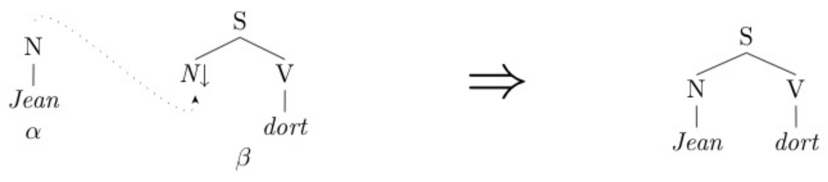
A tree-adjoining grammar is defined by a set of terminal symbols, a set of non-terminal symbols, a set of *initial trees* and a set of *auxiliary trees.* The union of initial trees and auxiliary trees is called *elementary trees.* They must verify the following:

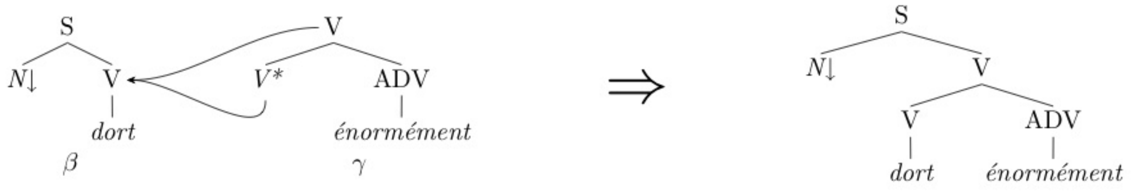
- the leaves of initial trees are terminal symbols or non-terminal symbols, which are then called *substitution nodes*,
- the leaves of auxiliary trees are terminal symbols or non-terminal symbols, which are then substitution nodes, except for a single non-terminal leaf, that has the same symbol as the root and is called the *foot node*

There are two operations possible with tree-adojoining grammars. The set of trees derived by these operations is called *derived trees.*

- *Substitution*: this operation is very easy to understand. It replaces a substitution node of a tree α by an initial or derived tree β if the substitution node and β's root node share the same symbol.
- *Adjunction*: this operation is a little trickier. The intution behind this operation is trying to insert a tree into another tree. To adjoin a tree β to a tree γ at node n, the sub-tree δ of γ with root n is removed from γ and substituted to β's foot node, the result is then substituted in γ at node n.

The following figures show examples of substitution and adjunction.





In these pictures, the usual TAG notations are used, that is that substitution nodes are noted with a downwards arrow and foot nodes are noted with a star.

3.3 Accurate automatic translation

As we have seen, a very powerful aspect of ACGs is that this formalism uses the same tools to generate sentences in a given language and a logical proposition to describe the meaning of a grammatical structure. This can be used to build an accurate automatic translator. The idea is that instead of trying to translate a sentence, the automatic translator is going to 'translate' the sentence into a logical proposition that will then be 'translated' into another language. More precisely, it will transduce (parse in an ACG an apply in a second ACG) the sentence in a logical proposition, and then transduce this proposition into a sentance again.

For example, to translate from French to German the sentence Jean+dort, the sentence is going to be parsed in an ACG that corresponds to French language, which will return *dort Jean*, this is then transformed in a second ACG to find its meaning, which will return *sleep John*, this is then parsed in a third ACG that is symetrical to the second one, but in German, it will return *schläft Johan*, which will ultimately be tranformed by a fourth ACG (symetrical to the first one, but in Greman), which will give Johan+schläft.

Conclusion

I think the part I wrote in the ACG prototype was a fundamental one, even though all basic features were already written. Actually, without parsing, ACGs have no real interest with respect to NLP, as it is what is most important when one wants to dig information from text. The next step would be to improve the *Datalog* solver, so that it returns the derivation of the query, so that the grammatical structure may be constructed from the object realisation through parsing. That would already allow transduction and therefore automatic translation. Another possible improvement would be to implement a parser for the lexicalized case as an algorithm exists and it would allow to parse more complex sentences, with relative clauses and the likes.

This stage, although a little short, made me experience a lot of things that were completely new to me. First of all, even though I had been programming in *CamlLight* and *OCaml* for four years, I had never used modules when writing code, which is impossible when projects get bigger. I also learned to use Emacs in a very basic way, allowing me to write and compile code much faster than before. I also learned to use Subversion, read and modify Makefiles and configuration files, among other things.

I would like to thank all the people I met when I was in Nancy, as they were all nice to me. Of course, Sylvain Pogodalla, who directed me for seven weeks and was always highly responsive even when he did not have much time. All the researchers, PhD students, engineers, interns and the assistant at team Calligramme, and the people I met in LORIA in general, who were always nice and lively, who gave me good pieces of advice and spent time with me.

Bibliography

[1] Philippe de Groote. Towards abstract categorial grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pages 148–155, 2001.

[2] Philippe de Groote and Sylvain Pogodalla. On the expressive power of abstract categorial grammars: Representing context-free formalisms. *Journal of Logic, Language and Information*, 13(4):421–438, 2004.

[3] Makoto Kanazawa. Second-order acgs as datalog programs. *unpublished*, August 2006.

[4] Makoto Kanazawa. Parsing and generation as datalog queries. In *Association for Computational Linguistics, 45th Annual Meeting, Proceedings of the Conference*, pages 176–183, 2007.

[5] Sylvain Salvati. *Problemes de Filtrage et Problemes d'Analyse pour les Grammaires Categorielles Abstraites.* PhD thesis, Institut National Polythechnique de Lorraine, 2005.