Parsing TAG with Abstract Categorial Grammar.

Sylvain Salvati

Parsing TAG with Abstract Categorial Grammar

Sylvain Salvati
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku,
Tokyo 101-8430, JAPAN
salvati@nii.ac.jp

Abstract

This paper presents informally an Earley algorithm for TAG which behaves as the algorithm given by (Schabes and Joshi, 1988). This algorithm is a specialization to TAG of a more general algorithm dedicated to second order ACGs. As second order ACGs allows to encode Linear Context Free Rewriting Systems (LCFRS) (de Groote and Pogodalla, 2004), the presentation of this algorithm gives a rough presentation of the formal tools which can be used to design efficient algorithms for LCFRS. Furthermore, as these tools allow to parse linear λ-terms, they can be used as a basis for developping algorithms for generation.

1 Introduction

The algorithm we present is a specialization to TAGs of a more general one dedicated to second order Abstract Categorial Grammars (ACGs) (de Groote, 2001). Our aim is to give here an informal presentation of tools that can be used to design efficient parsing algorithms for formalisms more expressive than TAG. Therefore, we only give a representation of TAGs with linear λ-terms together with simple derivation rules; we do not give in complete details the technical relation with ACGs. For some more information about ACGs and their relation to TAGs, one may read (de Groote, 2001) and (de Groote, 2002).

The advantage of using ACG is that they are defined with very few primitives, but can encode many formalisms. Thus they are well suited to study from a general perspective a full class of formalisms. In particular, a special class of ACGs (second order ACGs) embeds LCFRS (de Groote and Pogodalla, 2004), *i.e.* mildly context sensitive languages. Therefore, the study of second order ACGs leads to insights on mildly context sensitive languages. Having a general framework to describe parsing algorithms for mildly context sensitive languages may give some help to transfer some interesting parsing technique from one formalism to another. It can be, for example, a good mean to obtain prefix-valid algorithms, LC algorithms, LR algorithms... for the full class of mildly context sensitive languages.

The class of languages described by second order ACGs is wider than mildly context sensitive languages. They can encode tree languages, and more generally languages of linear λ-terms. As Montague style semantics (Montague, 1974) is based on λ-calculus, being able to parse linear λ-term is a first step towards generation algorithms seen as parsing algorithm. Furthermore, since this parsing algorithm is a generalization of algorithms *à la Earley* for CFGs and TAGs, the more general algorithm that can be used for generation (when semantic formulae are linear) can be considered as efficient.

The paper is organized as follows: section two gives basic defintions and tools concerning the linear λ-calculus. Section three explains how the indices usually used by parsers are represented for the linear λ-calculus. Section four gives a rough explaination of the encoding of TAGs within a compiled representation of second order ACGs. Section five explains the parsing algorithm and we conclude with section six.

2 The linear λ-calculus

We begin by giving a brief definition of linear types and linear λ-terms together with some standard notations. We assume that the reader is familiar with the usual notions related to λ-calculus (β-conversion, free variables, capture-avoiding substitutions...); for more details about λ-calculus, one may consult (Barendregt, 1984).

Definition 1 *The set of linear types, $\mathcal{T}$, is the smallest set containing $\{*\}$ and such that if $\alpha, \beta \in \mathcal{T}$ then $(\alpha \multimap \beta) \in \mathcal{T}$.*

Given a type $(\alpha_1 \multimap (\cdots (\alpha_n \multimap *) \cdots))$, we write it $(\alpha_1, \ldots, \alpha_n) \multimap *$.

Definition 2 *Given a infinite enumerable set of variables, $\mathcal{X}$, and an alphabet Σ, we define the set of linear λ-terms of type $\alpha \in \mathcal{T}$, Λ^α, as the smallest set satisfying the following properties:*

1. $x \in \mathcal{X} \Rightarrow x^\alpha \in \Lambda^\alpha$
2. $t \in \Lambda^\alpha \wedge x^\beta \in FV(t) \Rightarrow \lambda x^\beta.t \in \Lambda^{\beta \multimap \alpha}$
3. $a \in \Sigma \Rightarrow a \in \Lambda^{* \multimap *}$
4. $t_1 \in \Lambda^{\beta \multimap \alpha} \wedge t_2 \in \Lambda^\beta \wedge FV(t_1) \cap FV(t_2) \Rightarrow (t_1 t_2) \in \Lambda^\alpha$

In general, we write $\lambda x_1 \ldots x_n.t$ for $\lambda x_1 . \ldots \lambda x_n.t$ and we write $t_0 t_1 \ldots t_n$ for $(\ldots (t_0 t_1) \ldots t_n)$. Strings are represented by closed linear λ-terms of type $str = * \multimap *$. Given a string $abcde$, it is represented by the following linear λ-term: $\lambda y^*.a(b(c(d(e\, y^*))))$; $/w/$ represents the set of terms which are β-convertible to the λ-term representing the string w. Concatenation is represented by $+ = \lambda x_1^{str} x_2^{str} y^*.x_1^{str}(x_2^{str} y^*)$, and $(+w_1)w_2$ will be written $w_1 + w_2$. The concatenation is moreover associative, we may thus write $w_1 + \cdots + w_n$.

For the description of our algorithm, we rely on contexts:

Definition 3 *A context is a λ-term with a hole. Contexts are defined by the following grammar:*

$$\mathbf{C} = [] \mid \Lambda \mathbf{C} \mid \mathbf{C} \Lambda \mid \lambda \mathcal{V}.\mathbf{C}$$

The insertion of a term within a context is done the obvious way. One has nevertheless to remark that when a term t is inserted in a context $C[]$, the context $C[]$ can bind variables free in t. For example, if $C[] = \lambda x.[]$ and $t = x$ then $C[t] = \lambda x.x$ and x which was free in t is not free anymore in $C[t]$.

3 Indices as syntactic descriptions

Usually the items of Earley algorithms use indices to represent positions in the input string. The algorithm we describe is a particular instance of a more general one which parses linear λ-terms rather than strings. In that case, one cannot describe in a simple way positions by means of indices. Instead of indices, positions in a term t will be represented with *zippers* ((Huet, 1997)), *i.e.* a pair $(C[], v)$ of a context and a term such that $C[v] = t$. Figure 1 explicits the correspondence between indices and zippers via an example.

The items of Earley algorithms for TAGs use pairs of indices to describe portions of the input string. In our algorithm, this role is played by linear types built upon zippers; the parsing process can be seen as a type-checking process in a particular type system. We will not present this system here, but we will give a flavor of the meaning of those types called *syntactic descriptions* (Salvati, 2006). In order to represent the portion of a string between the indices i and j, we use the zippers $(C_i[], v_i)$ and $(C_j[], v_j)$ which respectively represent the position i and j in the string. The portion of string is represented by the syntactic description $(C_j[], v_j) \multimap (C_i[], v_i)$; this syntactic description can be used to type functions which take v_j as argument and return v_i as a result. For example, given the syntactic description: $(\lambda x.a(b(c[])), d(e\, x)) \multimap (\lambda x.a[], b(c(d(e\, x))))$, it represents the set of functions that result in terms that are β-convertible to $b(c(d(e\, x)))$ when they take $d(e\, x)$ as an argument; this set is exactly $/bc/$. Our algorithm uses representations of string contexts with syntactic descriptions such as $\mathbf{d} = ((C_1[], v_1) \multimap (C_2[], v_2)) \multimap (C_3[], v_3) \multimap (C_4[], v_4)$ (in the following we write $((C_1[], v_1) \multimap (C_2[], v_2), (C_3[], v_3)) \multimap (C_4[], v_4)$ for such syntactic descriptions). Assume that $(C_1[], v_1) \multimap (C_2[], v_2)$ represents $/bc/$ and that $(C_3[], v_3) \multimap (C_4[], v_4)$ represents $/abcde/$, then $\mathbf{d}$ describes the terms which give a result in $/abcde/$ when they are applied to an element of $/bc/$. Thus, $\mathbf{d}$ describes the set of terms β-convertible to $\lambda f y.a(f(d(e\, y)))$, the set of terms representing the string context $a[\,]de$.

Some of the syntactic descriptions we use may contain *variables* denoting *non-specified syntactic descriptions* that may be instanciated during parsing. In particular, the syntactic description variable F will always be used as a non-specified syntactic description representing strings (*i.e.* F may only be substituted by a syntactic description of the form $(C_1[], v_1) \multimap (C_2[], v_2)$), such syntactic descriptions will represent the foot of an auxiliary tree. We will also use Y to represent a non-specifed point in the input sentence (*i.e.* Y may only be substituted by syntactic descriptions of the form $(C[], v)$), such syntactic descriptions will represent the end of an elementary tree.

As syntactic desccriptions are types for the linear λ-calculus, we introduce the notion of typing context for syntactic descriptions.

Definition 4 *A typing context Γ (context for short), is a set of pairs of the form $x : d$ where x is a variable and d is a syntactic description such that $x : d \in \Gamma$ and $x : e \in \Gamma$ iff $d = e$.*

If $x : d \in \Gamma$, then we say that x is declared with type d in Γ.

Typing contexts Γ must not be confused with con-

0	$(\lambda x.[], a(b(c(d(e\,x)))))$	1	$(\lambda x.a[], b(c(d(e\,x))))$
2	$(\lambda x.a(b[]), c(d(e\,x)))$	3	$(\lambda x.a(b(c[])), d(e\,x))$
4	$(\lambda x.a(b(c(d[]), e\,x)$	5	$(\lambda x.a(b(c(d(e[])))), x)$

$_0a_1b_2c_3d_4e_5$

Figure 1: Correspondence indices/zippers for the string $abcde$

texts $C[]$. If a typing context Γ is the set $\{x_1 : d_1; \ldots; x_n : d_n\}$ then we will write if by $x_1 : d_1, \ldots, x_n : d_n$. In the present paper, typing contexts may declare at most two variables.

4 Representing TAG with second order ACGs

We cannot give here a detailed definition of second order ACGs here. We therefore directly explain how to transform TAGs into lexical entries representing a second order ACG that can be directly used by the algorithm.

We represent a TAG $\mathbf{G}$ by a *set of lexical entries* $\mathcal{L}_\mathbf{G}$. *Lexical entries* are triples (Γ, t, α) where Γ is a typing context, t is a linear λ-term and α is either N_a, N_s or $N_a.1$ if N is a non-terminal of the considered TAG. Without loss of generality, we consider that the adjunction at an interior node of an elementary tree is either mandatory or forbidden[1]. We adopt the convention of representing adjunction nodes labeled with N by the variable $x_{N_a}^{str \multimap str}$, the substitution nodes labeled with $N\downarrow$ by the variable $x_{N_s}^{str}$, the foot node of an auxiliary tree labeled with N^* by the variable $f_{N_a.1}^{str}$ and the variable y^* will represent the end of strings. When necessary, in order to respect the linearity constraints of the λ-terms, indices are used to distinguish those variables. This convention being settled, the type annotation on variables is not necessary anymore, thus we will write x_{N_a}, x_{N_s}, $f_{N_a.1}$ and y. To translate the TAG, we use the function ϕ defined by figure 2. Given an initial tree T whose root is labeled by N and t the normal form of $\phi(T)$, $(\ , t, N_s)^2$ is the lexical entry associated to T; if T is an auxiliary tree whose root is labeled by N and t is the normal form of $\phi(T)$ then $(\ , \lambda f_{N_a.1}.t, N_a)^2$ is the lexical entry associated to T. A TAG $\mathbf{G}$ is represented by $\mathcal{L}_\mathbf{G}$ the smallest set verifying:

1. if T is an elementary tree of $\mathbf{G}$ then the lexical entry associated to T is in $\mathcal{L}_\mathbf{G}$.
2. if $(\ , t, \alpha) \in \mathcal{L}_\mathbf{G}$, with α equals to N_a or N_s, and $t = C[x_{N_a} t_1 t_2]$ then $(\Gamma, t_1, N_a.1) \in \mathcal{L}_\mathbf{G}$ where $\Gamma = f_{M_a.1} : F$ if $f_{M_a.1} \in FV(t_1)$ otherwise Γ is the empty typing context.

Given a term t such that $x_\alpha \in FV(t)$, and $(\Gamma, t', \alpha) \in \mathcal{L}_\mathbf{G}$, then we say that t is rewritten as $t[x_\alpha := t']$, $t \Rightarrow t[x_\alpha := t']$. Furthermore if x_α is the leftmost variable we write $t \Rightarrow_l t[x_\alpha := t']$. It is easy to check that if $t \overset{*}{\Rightarrow} t'$ with $FV(t') = \emptyset$, then $t \overset{*}{\Rightarrow}_l t'$. A string w is generated by a $\mathcal{L}_\mathbf{G}$ whenever $x_{S_s} \overset{*}{\Rightarrow} t$ and $t \in /w/$ (S being the start symbol of $\mathbf{G}$). Straightforwardly, the set of strings generated by $\mathcal{L}_\mathbf{G}$ is exactly the language of $\mathbf{G}$.

5 The algorithm

As we want to emphasize the fact that the algorithm we propose borrows much to type checking, we use sequents in the items the algorithm manipulates. Sequents are objects of the form $\Gamma \vdash t : \mathbf{d}$ where Γ is a typing context, t is a linear λ-term, and $\mathbf{d}$ is a syntactic description.

The algorithm uses two kinds of items; either items of the form $(\alpha; \Gamma \vdash t : \mathbf{d}; L)$ (where L is a list of sequents, the subgoals, here L contains either zero or one element) or items of the form $[N_a.1; \Gamma; t; (C_1[], v_1) \multimap (C_2[], v_2)]$. All the possible instances of the items are given by figure 3. The algorithm is a recognizer but can easily be extended into a parser[3]. It fills iteratively a chart until a fixed-point is reached. Elements are added to the chart by means of inference rules given by figure 4, in a deductive parsing fashion (Shieber et al., 1995). Inference rules contain two parts: the first part is a set of premises which state conditions on elements that are already in the chart. The second part gives the new element to add to the chart if it is not already present. For the more general algorithm, the rules are not much more numerous as they can be abstracted into more general schemes.

An item of the form $(\alpha; \Gamma_1 \vdash t_1 : \mathbf{d}; \Gamma_2 \vdash t_2 : (C_1[], v_1))$ verifies:

1. $(\Gamma_1', t_1, \alpha) \in \mathcal{L}_\mathbf{G}$ where $\Gamma_1' = f_{N_a.1} : F$ if $\Gamma_1 = f_{N_a.1} : \mathbf{e}$ or $\Gamma_1' = \Gamma_1$ otherwise.

[1] We do not treat here the case of optional adjunction, but our method can be straightforwardly extended to cope with it, following ideas from (de Groote, 2002). It only modifies the way we encode a TAG with a set of lexical entries, the algorithm remains unchanged.

[2] In that case the typing context is empty.

[3] Actually, if it is extended into a parser, it will ouput the shared forest of the derivation trees; (de Groote, 2002) explains how to obtain the derived trees from the derivation trees in the framework of ACGs

$\phi\left(\begin{array}{ccc} & N & \\ \diagup & & \diagdown \\ T_1 & \dots & T_n \end{array}\right)$	$\longrightarrow$	$\lambda y.x_{N_a}(\phi(T_1)+\dots+\phi(T_n))y \quad x_{N_a}$ and y are fresh
$\phi\left(\begin{array}{ccc} & N_{NA} & \\ \diagup & & \diagdown \\ T_1 & \dots & T_n \end{array}\right)$	$\longrightarrow$	$\phi(T_1)+\dots+\phi(T_n)$
$\phi(N^*)$	$\longrightarrow$	$\lambda y.x_{N_a}(\lambda y.f_{N.1}y)y$
$\phi(N^*_{NA})$	$\longrightarrow$	$\lambda y.f_{N.1}y$
$\phi(N\downarrow)$	$\longrightarrow$	$\lambda y.x_{N_s}y$
$\phi(a)$	$\longrightarrow$	$\lambda y.ay$
$\phi(\epsilon)$	$\longrightarrow$	$\lambda y.y$

Figure 2: Translating TAG into ACG: definition of ϕ

General items
$(N_a\ ;\ \vdash \lambda f_{N_a.1}y.t_1 : (F,Y) \multimap (C_1[],v_1)\ ;\ f_{N_a.1} : F, y : Y \vdash t_2 : (C_2[],v_2))$
$(N_a\ ;\ \vdash \lambda f_{N_a.1}y.t : ((C_1[],v_1) \multimap (C_2[],v_2),Y) \multimap (C_3[],v_3)\ ;\ y : Y \vdash t_2 : (C_4[],v_4))$
$(N_a\ ;\ \vdash \lambda f_{N_a.1}y.t : ((C_1[],v_1) \multimap (C_2[],v_2),(C_3[],v_3)) \multimap (C_4[],v_4)\ ;\ )$
$(\alpha\ ;\ \vdash \lambda y.t_1 : Y \multimap (C_1[],v_1)\ ;\ y : Y \vdash t_2 : (C_2[],v_2))$
$(\alpha\ ;\ \vdash \lambda y.t : (C_1[],v_1) \multimap (C_2[],v_2)\ ;\ )$
$(N_a.1\ ;\ f_{M_a.1} : F \vdash \lambda y.t : Y \multimap (C[],v)\ ;\ f_{M_a.1} : F, y : Y \vdash t_2 : (C_2[],v_2)$
$(N_a.1\ ;\ f_{M_a.1} : (C_1[],v_1) \multimap (C_2[],v_2) \vdash \lambda y.t : Y \multimap (C_3[],v_3)\ ;\ y : Y \vdash t_2 : (C_4[],v_4))$
$(N_a.1\ ;\ f_{M_a.1} : (C_1[],v_1) \multimap (C_2[],v_2) \vdash \lambda y.t : (C_3[],v_3) \multimap (C_4[],v_4)\ ;\ )$
Wrapped subtrees
$[N_a.1\ ;\ \ ;\ t\ ;\ (C_1[],v_1) \multimap (C_2[],v_2)]$
$[N_a.1\ ;\ f_{M_a.1} : (C_1[],v_1) \multimap (C_2[],v_2)\ ;\ t\ ;\ (C_3[],v_3) \multimap (C_4[],v_4)]$

Figure 3: Possible items

2. there is a context $C[]$ such that $t_1 = C[t_2]$ and if $\mathbf{d}$ is of the form $(\mathbf{d}_1,\dots,\mathbf{d}_n) \multimap (C_2[],v_2)$ (n must be 1, or 2) then $C[y] \stackrel{*}{\Rightarrow}_l t'$ so that t' is described by $(C_1[],v_1) \multimap (C_2[],v_2)$.

3. if $\Gamma_1 = f_{N_a.1} : (C_3[],v_3) \multimap (C_4[],v_4)$ or if $\mathbf{d} = ((C_3[],v_3) \multimap (C_4[],v_4),Y) \multimap (C_2[],v_2)$ and $t_1 = \lambda f_{N_a.1}y.v$ then $f_{N_a.1} \Rightarrow_l t''$ and t'' is described by $(C_3[],v_3) \multimap (C_4[],v_4)$

An item of the form $(\alpha;\Gamma \vdash t : \mathbf{d};)$ verifies:

1. $(\Gamma',t,\alpha) \in \mathcal{L}_{\mathbf{G}}$ where $\Gamma' = f_{N_a.1} : F$ if $\Gamma = f_{N_a.1} : \mathbf{e}$ or $\Gamma' = \Gamma$ otherwise

2. $\mathbf{d}$ does not contain non-specified syntactic descriptions[4].

3. $t \stackrel{*}{\Rightarrow}_l t'$ and t' is described by $\mathbf{d}$ ($\mathbf{d}$ may either represent a string context or a string).

4. if $\Gamma = f_{N_a.1} : (C_3[],v_3) \multimap (C_4[],v_4)$ or if $\mathbf{d} = ((C_3[],v_3) \multimap (C_4[],v_4),(C_1[],v_1)) \multimap (C_2[],v_2)$ and $t_1 = \lambda f_{N_a.1}y.t'$ then $f_{M_a.1} \stackrel{*}{\Rightarrow}_l t''$ and t'' is described by $(C_3[],v_3) \multimap (C_4[],v_4)$

[4]There is no occurence of F or Y in $\mathbf{d}$.

Finally an item of the form $[N_a.1;\Gamma;t;(C_1[],v_1) \multimap (C_2[],v_2)]$ implies the existence of t', $(C_3[],v_3)$ and $(C_4[],v_4)$ such that $(N_a;\vdash t' : ((C_3[],v_3) \multimap (C_4[],v_4),(C_1[],v_1)) \multimap (C_2[],v_2);)$ and $(N_a.1;\Gamma \vdash t : (C_3[],v_3) \multimap (C_4[],v_4));)$ are in the chart.

An input $\lambda y.C[y]$ is recognized iff when the fixed-point is reached, the chart contains an item of the form $(S_s;\ \vdash t : (\lambda y.C[],y) \multimap (\lambda y.[],C[y]);\ )$ (where S is the start symbol of the TAG $\mathbf{G}$.

6 Conclusion and perspective

In this paper, we have illustrated the use for TAGs of general and abstract tools, syntactic descriptions, which can be used to parse linear λ-terms. Even though ACGs are very general in their definition, the algorithm we describe shows that this generality is not a source of unefficiency. Indeed, this algorithm, a special instance of a general one which can parse any second order ACG and it behaves exactly the same way as the algorithm given by (Schabes and Joshi, 1988) so that it parses a second order ACG encoding a TAG in $\mathcal{O}(n^6)$.

The technique used enables to see generation as parsing. In the framework of second order ACG, the

The initializer

$$\frac{(\lambda y.t, S_s) \in \mathcal{L}_{\mathbf{G}}}{(S_s;\ \vdash \lambda y.t : Y \multimap (\lambda y.[], u); y : Y \vdash t : (\lambda y.[], u))}$$

The scanner

$$\frac{(\alpha; \Gamma_1 \vdash t_1 : \mathbf{d}; \Gamma_2 \vdash at_2 : (C[], av))}{(\alpha; \Gamma_1 \vdash t_1 : \mathbf{d}; \Gamma_2 \vdash t_2 : (C[a[]], v))} \qquad \frac{(\alpha; \Gamma \vdash t : \mathbf{d}; y : Y \vdash y : (C[], v)) \quad \sigma = [Y := (C[], v)]}{(\alpha; \Gamma \vdash t : \mathbf{d}.\sigma;)}$$

The predictor

$$\frac{(\alpha; \Gamma_1 \vdash t_1 : \mathbf{d}; \Gamma_2 \vdash x_{N_a} t_2 t_3 : (C[], v)) \quad (\ , \lambda f_{N_a.1} y.t, N_a) \in \mathcal{L}_{\mathbf{G}}}{(N_a;\ \vdash \lambda f_{N_a.1} y.t : (F, Y) \multimap (C[], v); f_{N_a.1} : F, y : Y \vdash t : (C[], v))}$$

$$\frac{(\alpha; \Gamma_1 \vdash t_1 : \mathbf{d}; \Gamma_2 \vdash x_{N_s} t_2 : (C[], v)) \quad (\ , \lambda y.t, N_s) \in \mathcal{L}_{\mathbf{G}}}{(N_s;\ \vdash \lambda y.t : Y \multimap (C[], v); y : Y \vdash t : (C[], v))} \qquad \frac{\begin{array}{l}(\alpha; \Gamma_1 \vdash t_1 : \mathbf{d}; \Gamma_2 \vdash f_{N_a.1} t_2 : (C_2[], v_2)) \\ (\Gamma_3, \lambda y.t_3, N_a.1) \in \mathcal{L}_{\mathbf{G}}\end{array}}{(N_a.1; \Gamma_3 \vdash \lambda y.t_3 : Y \multimap (C_2[], v_2); \Gamma_3, y : Y \vdash t_3 : (C_2[], v_2))}$$

The completer

$$\frac{\begin{array}{l}(N_a;\ \vdash t_1 : ((C_1[], v_1) \multimap (C_2[], v_2), (C_3[], v_3)) \multimap (C_4[], v_4);) \\ (N_a.1; \Gamma_2; t_2 : (C_1[], v_1) \multimap (C_2[], v_2);)\end{array}}{[N_a.1; \Gamma_2; t_2; (C_3[], v_3) \multimap (C_4[], v_4)]} \qquad \frac{\begin{array}{l}(\alpha; \Gamma_1 \vdash t_1 : \mathbf{d}; y : Y, \Gamma_2' \vdash x_{N_a} t_2 t_3 : (C_1[], v_1)) \\ [N_a.1; \Gamma_2; t_2; (C_2[], v_2) \multimap (C_1[], v_1)] \\ \text{if } \Gamma_2 = f_{M_a.1} : \mathbf{f} \text{ then } \sigma = [F := \mathbf{f}] \text{ else } \sigma = Id\end{array}}{(\alpha; \Gamma_1.\sigma \vdash t_1 : \mathbf{d}.\sigma; \Gamma_2 \vdash t_3 : (C_2[], v_2))}$$

$$\frac{\begin{array}{l}(\alpha; \Gamma_1 \vdash t_1 : \mathbf{d}; f_{N_a.1} : F, y : Y \vdash f_{N_a.1} t_2 : (C_1[], v_1)) \\ (N_a.1; \Gamma_2 \vdash t_2 : (C_2[], v_2) \multimap (C_1[], v_1);) \\ \sigma = [F := (C_2[], v_2) \multimap (C_1[], v_1)]\end{array}}{(\alpha; \Gamma_1.\sigma \vdash t_1 : \mathbf{d}.\sigma; y : Y \vdash t_2 : (C_2[], v_2))} \qquad \frac{\begin{array}{l}(\alpha; \Gamma_1 \vdash t_1 : d; \Gamma_2 \vdash x_{N_s} t_2 : (C_1[], v_1)) \\ (N_s;\ \vdash t_2 : (C_2[], v_2) \multimap (C_1[], v_1);)\end{array}}{(\alpha; \Gamma_1 \vdash t_1 : d; \Gamma_2 \vdash t_2 : (C_2[], v_2))}$$

Figure 4: The rules of the algorithm

logical formulae on which generation is performed are bound to be obtained from semantic recipies coded with linear λ-terms and are therefore not really adapted to Montague semantics. Nevertheless, syntactic descriptions can be extended with intersection types (Dezani-Ciancaglini et al., 2005) in order to cope with simply typed λ-calculus. With this extension, it seems possible to extend the algorithm for second order ACGs so that it can deal with simply typed λ-terms and without loosing its efficiency in the linear case.

References

Henk P. Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics*, volume 103. Studies in Logic and the Foundations of Mathematics, North-Holland Amsterdam. revised edition.

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

Philippe de Groote. 2001. Towards abstract categorial grammars. In Association for Computational Linguistic, editor, *Proceedings 39th Annual Meeting and 10th Conference of the European Chapter*, pages 148–155. Morgan Kaufmann Publishers.

Philippe de Groote. 2002. Tree-adjoining grammars as abstract categorial grammars. *TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks*, pages 145–150.

Mariangiola Dezani-Ciancaglini, Furio Honsell, and Yoko Motohama. 2005. Compositional Characterization of λ-terms using Intersection Types. *Theoret. Comput. Sci.*, 340(3):459–495.

Gérard Huet. 1997. The zipper. *Journal of Functional Programming*, 7(5):549–554.

Richard Montague. 1974. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, CT.

Sylvain Salvati. 2006. Syntactic descriptions: a type system for solving matching equations in the linear λ-calculus. In *to be published in the proceedings of the 17th International Conference on Rewriting Techniques and Applications*.

Yves Schabes and Aravind K. Joshi. 1988. An earley-type parsing algorithm for tree adjoining grammars. In *Proceedings of the 26th annual meeting on Association for Computational Linguistics*, pages 258–269, Morristown, NJ, USA. Association for Computational Linguistics.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, July–August. Also available as cmp-lg/9404008.