# Computing Semantic Representation: Towards ACG Abstract Terms as Derivation Trees

Sylvain Pogodalla

# Computing Semantic Representation: Towards ACG Abstract Terms as Derivation Trees

**Sylvain Pogodalla**
LORIA - Campus Scientifique
BP239
F-54602 Vandœuvre-lès-Nancy – France
Sylvain.Pogodalla@loria.fr

## Abstract

This paper proposes a process to build semantic representation for Tree Adjoining Grammars (TAGs) analysis. Being in the derivation tree tradition, it proposes to reconsider derivation trees as abstract terms ($\lambda$-terms) of Abstract Categorial Grammars (ACGs). The latter offers a flexible tool for expliciting compositionality and semantic combination. The chosen semantic representation language here is an underspecified one. The ACG framework allows to deal both with the semantic language and the derived tree language in an equivalent way: as concrete realizations of the abstract terms. Then, in the semantic part, we can model linguistic phenomena usually considered as difficult for the derivation tree approach.

## Introduction

When dealing with the computation of semantic representation for TAG analysis, two main approaches are usually considered. The first one gives the derivation trees a central role for the computation (Schabes and Shieber, 1994; Candito and Kahane, 1998; Kallmeyer, 2002; Joshi et al., 2003), and the second one relies on a direct computation on the derived tree (Frank and van Genabith, 2001; Gardent and Kallmeyer, 2003).

The present article wants to explore the intuition that the two approaches are indeed bound: derivation trees are a specification of the operations that are to be processed, but the derived trees hold the precise descriptions of these operations. We propose to exhibit those operations by separating them from the syntactic trees. Then, under the specifications given by the derivation trees, we show how to build the semantic representations.

The tools we use for this purpose are Abstract Categorial Grammars (ACGs) (de Groote, 2001). The main feature of an ACG is to generate two languages: an *abstract language* and an *object language*. Whereas the abstract language may appear as a set of grammatical or parse structures, the object language may appear as its realization, or the concrete language it generates. For instance, (de Groote, 2002) proposes as object language the tree language of TAGs (encoded in linear $\lambda$-terms) and, as abstract language, a tree language (also encoded in linear $\lambda$-terms) and *very close to the derivation tree language*. In this paper, we use the same abstract language, and, as object language, $\lambda$-terms that encode underspecified semantic representation as in (Bos, 1995; Blackburn and Bos, 2003). Thus, we realize our program to separate the computation specification and the operation definition. As for Montague's semantics, missing information is represented by bound $\lambda$-variables and replacement and variable catching by application instead of unification (as in (Frank and van Genabith, 2001; Gardent and Kallmeyer, 2003)).

The next section briefly describes the underlying principles of ACGs. Then we show how syntactic parts of TAGs are modelled and how we translate, through the abstract terms (our derivation trees), the combination of intial and auxiliary trees to their semantic representations by means of some examples.

## 1 ACG Principles

An ACG $\mathcal{G}$ defines:

1. two sets of typed $\lambda$-terms: $\Lambda_1$ (based on the typed constant set $C_1$) and $\Lambda_2$ (based on the typed constant set $C_2$);
2. a morphism $\mathcal{L} : \Lambda_1 \rightarrow \Lambda_2$;
3. a distinguished type $\mathbf{S}$.

(de Groote, 2001) defines both $\Lambda_1$ and $\Lambda_2$ as sets of *linear* $\lambda$-terms. In this paper, we use simply typed $\lambda$-terms for $\Lambda_2$, using the translation of intuitionnistic logic into

*TAG+7: Seventh International Workshop on Tree Adjoining Grammar and Related Formalisms.*
*May 20-22, 2004, Vancouver, BC, CA.*
*Pages 64-71.*

linear logic $A \to B = (!A) \multimap B$ (Girard, 1987; Danos and Cosmo, 1992). We don't elaborate on that subject in this paper, but it does not change the main properties of ACGs[1]. Then the abstract language $\mathcal{A}(\mathcal{G})$ and the object language $\mathcal{O}(\mathcal{G})$ are defined as follows:

$$\mathcal{A}(\mathcal{G}) = \{t \in \Lambda_1 | t : \mathbf{S}\}$$
$$\mathcal{O}(\mathcal{G}) = \{t \in \Lambda_2 | \exists u \in \mathcal{A}(\mathcal{G})\ t = \mathcal{L}(u)\}$$

Note that $\mathcal{L}$ binds the parse structures in $\mathcal{A}(\mathcal{G})$ to the concrete expressions of $\mathcal{O}(\mathcal{G})$. Depending on the choice of $\Lambda_1$, $\Lambda_2$ and $\mathcal{L}$, it can map for instance derivation trees and derived trees for TAGs (de Groote, 2002), derivation trees of context-free grammars and strings of the generated language (de Groote, 2001), derivation trees of $m$-linear context-free rewriting systems and strings of the generated language (de Groote and Pogodalla, 2003). Of course, this link between an abstract and a concrete structure can apply not only to syntactical formalisms, but also to semantic formalisms.

The main point here is that ACGs can be mixed in different ways: in a transversal way, were two ACGs use the same abstract language, or in a compositional way, were the abstract language of an ACG is the object language of an other one. In this paper, as described in figure 1, we use different ACGs and some composition with $\mathcal{G}$: $\Lambda_2$ is the tree language of TAGs, $\Lambda_1$ the tree language of our *derivation trees*. For $\mathcal{G}'$, we have the same abstract language and $\Lambda'_2$ is the underspecified representation language. In dotted lines is a composition presented in (de Groote, 2001) between strings and derivation trees we do not use here.

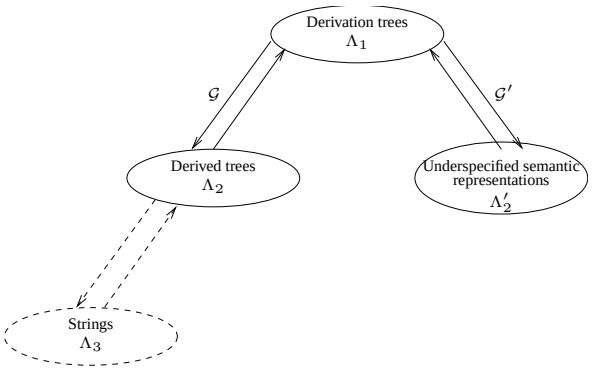


Figure 1: Moving from an object language to another

[1]In particular, this means that, provided there is no vacuous abstraction in $\mathcal{L}(C_1)$ and every $c \in \mathcal{L}(C_1)$ is such that it has $t \in C_2$ as subterm, we can decide if, for $u \in \Lambda_2$ if $u \in \mathcal{O}(\mathcal{G})$ and what is (are) the antecedent(s) (Pogodalla, 2004).

## 2 TAGs as ACGs

This section refers to (de Groote, 2002), which proposes to encode TAGs into ACGs. Given a TAG $G$, $\Lambda_1$ is build as follows:

- for every non-termninal symbol $X$, there are two types $X_S$ and $X_A$ standing for places where substitution and adjunction can occur respectively;
- for every elementary tree $\gamma$, there is a constant $c_\gamma \in C_1$. Moreover, for every non-terminal symbol $X$, there is a constant $I_X : X_A$.

For instance, given the trees of table 1, we have the constants and their types (for concision, we suppress parameters that are not used in the next examples of this paper, namely nodes where no adjunction occur[2]):

$$\begin{aligned} c_{\textit{every}} &: \mathbf{N}_A \\ c_{\textit{dog}} &: \mathbf{N}_A \multimap \mathbf{N}_S \\ c_{\textit{chases}} &: \mathbf{S}_A \multimap \mathbf{VP}_A \multimap \mathbf{N}_S \multimap \mathbf{N}_S \multimap \mathbf{S}_S \\ c_{\textit{usually}} &: \mathbf{VP}_A \multimap \mathbf{VP}_A \end{aligned}$$

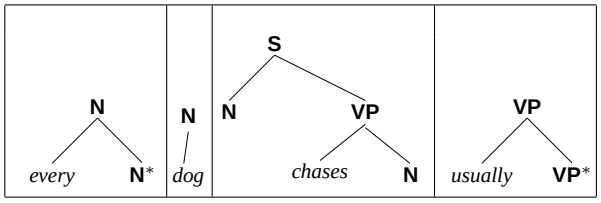


Table 1: Examples of elementary trees

To completely define the ACG $\mathcal{G}$, we need to define $\Lambda_2$ and $\mathcal{L}$. The types of $\Lambda_2$ are made of the single type $\tau$, representing the type of trees. For any non-terminal symbol $X$, there are constants $X_0, \ldots, X_i$ where $i$ is the maximal number of children of the $X$ nodes in the elementary trees. For any terminal symbol $X$ in $G$, there is a constant $X : \tau \in C_2$. Then $\mathcal{L}$ is defined by sending any $X_S$ type to the type $\tau$, and any $X_A$ types to the type $\tau \multimap \tau$. Corresponding to the trees of table 1, we have for instance:

$$\begin{aligned} \mathcal{L}(I_X) &= \lambda x.x : \tau \multimap \tau \\ \mathcal{L}(c_{\textit{every}}) &= \lambda x.\mathbf{N_2}(\textbf{every}\ x) : \tau \multimap \tau \\ \mathcal{L}(c_{\textit{dog}}) &= \lambda N.N(\mathbf{N_1}\textbf{dog}) : (\tau \multimap \tau) \multimap \tau \\ \mathcal{L}(c_{\textit{chases}}) &= \lambda SV.\lambda x.\lambda y.S(\mathbf{S_2}x(V \\ &\quad (\mathbf{VP_2}\textbf{chases}\ y))) \\ &\quad : (\tau \multimap \tau) \multimap (\tau \multimap \tau) \multimap \tau \multimap \tau \multimap \tau \end{aligned}$$

Note that in the adjunction operation, the auxiliary tree is a parameter. But it also has a higher-order type, that

[2]For instance, the type of $c_{\textit{every}}$ should be $\mathbf{Det}_A \multimap \mathbf{N}_A \multimap \mathbf{N}_A$.

is a function from trees to trees. We let the reader check that $\mathcal{L}(c_{chases} I_{\mathbf{S}} I_{\mathbf{VP}}(c_{dog} c_{every})(c_{cat} c_{some}))$ correspond the derived tree associated to *every dog chases some cat* of figure 2.

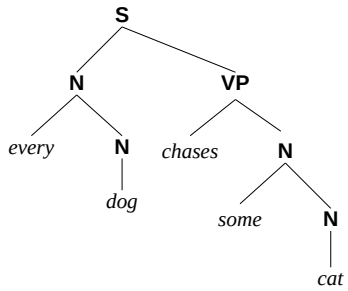


Figure 2: $\mathcal{L}(c_{chases} I_{VP}(c_{dog} c_{every})(c_{cat} c_{some})) = \mathbf{S}_2(\mathbf{N}_2\ \textit{every}\ (\mathbf{N}_1\ \textit{dog}\ ))(\mathbf{VP}_2\ \textit{chases}\ (\mathbf{N}_2\ \textit{some}\ (\mathbf{N}_1\ \textit{cat}\ )))$

We note two important things. First, the abstract terms, as $c_{chases} I_{\mathbf{S}} I_{\mathbf{VP}}(c_{dog} c_{every})(c_{cat} c_{some})$ can be represented by a tree structure where the children of a node are its arguments. Then erasing the $I_X$ arguments, and directing the edges downward if the argument is of type $X_S$ and upward if the argument is of type $X_A$, we get the usual notion of derivation tree. Second, the auxiliary trees are modelled as higher-order function. We use the same approach in our semantic modelling, getting some type raising, as in Montague's semantics. But let us precise the ACG we use for the semantic representation.

## 3 Semantic representation for TAGs as ACGs

The semantic representation language we use is an underspecified one presented in (Bos, 1995; Blackburn and Bos, 2003): the predicate logic "unplugged". The aim of this language, the *underspecified representation language* (URL) is to specify in a single formula the possible formulas (of the *semantic representation language* (SRL)) associated to an ambiguous expression. For instance, the expression *every dog chases a cat* has the two possible meanings:

$$\forall x(\mathbf{dog}(x) \Rightarrow \exists y(\mathbf{cat}(y) \wedge \mathbf{chases}(x,y)))$$
$$\exists y(\mathbf{cat}(y) \wedge \forall x(\mathbf{dog}(x) \Rightarrow \mathbf{chases}(x,y)))$$

To mark the difference between the SRL and the URL, both being first order languages, we translate the usual first order logic symbols of SRL. This translation is straightforward, using boldface symbols (e.g, **All**, **And**, **Imp**, etc.). In SRL, the two previous formulas are restated as follows:

$$\mathbf{All}(x, \mathbf{Imp}(\mathbf{dog}(x), \mathbf{Some}(y, \mathbf{And}(\mathbf{cat}(y), \mathbf{chases}(x,y)))))$$
$$\mathbf{Some}(y, \mathbf{And}(\mathbf{cat}(y), \mathbf{All}(x, \mathbf{Imp}(\mathbf{dog}(x), \mathbf{chases}(x,y)))))$$

Both these formulas have the property that:

- they have at leat two subformulas: one quantified by **All**, one quantified by **Some**;
- $\mathbf{chases}(x,y)$ is a subformulas of the two quantified subformulas.

The URL relies on the speficication of subformula constraints that the SRL formulas have to satisfy, and the two SRL formulas above can be described by the following URL formula:

$$\exists h_0 h_1 h_2 l_1 l_2 l_3 l_4 l_5 l_6 l_7 xy(l_1 : \mathbf{All}(x, l_2)$$
$$\wedge l_2 : \mathbf{Imp}(l_3, h_1) \wedge l_3 : \mathbf{dog}(x) \wedge l_4 : \mathbf{Some}(y, l_5)$$
$$\wedge l_5 : \mathbf{And}(l_6, h_2) \wedge l_6 : \mathbf{cat}(y)$$
$$\wedge l_7 : \mathbf{chases}(x,y) \wedge h_1 \geq l_7 \wedge h_2 \geq l_7 \wedge h_0 \geq l_1$$
$$\wedge h_0 \geq l_4)$$

illustrated in figure 3. The syntax of URL is basically the same that first-order logic, except that if atomic formulas remain the same, formulas are built from *holes* and *labels*, the latter being used as place holder for logical formulas in the underspecified representation language. We use the usual logical symbols ($\exists$, $\wedge$), an infix predicate $\geq$ to specify the constraints and an infix operateur : for URL. The symbol $h \geq l$ imposes the constraint for a formula that is associated to $l$ to be a subformula of the one associated to $h$. $l : p$ indicates that a predicate $p$ of SRL is labelled in URL by $l$.

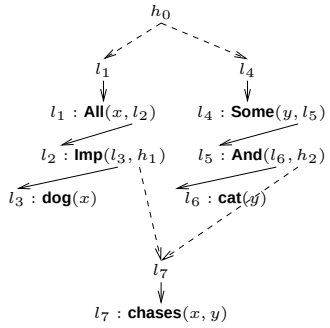


Figure 3: URL formula for *every dog chases a cat*

We want to underline the difference between URL and SRL because our concern in this paper is not to build and manage SRL formulas, but only URL formulas, that is underspecified representations. So that the object language of the ACG we are designing is URL.

Coming back to the figure 1, we established in the previous section the $\mathcal{G}$ ACG to encode TAGs. We know want to rely on the common abstract language, $\Lambda_1$, the one of derivation trees, to build the $\mathcal{G}'$ ACG that model the semantic behaviour, with URL as $\Lambda'_2$. So let us now define $\mathcal{G}'$.

First is $\Lambda'_2$:

- the types we use are $e, h, l, p, t$ where $e$ stands for entities, $h$ for holes, $l$ for labels, $p$ for predicate of the logical language and $t$ for truth values;
- the constants are $\geq$, :, $\exists_l$, $\exists_e$, $\exists_h$, $\wedge$, **Imp**, **And**, **Some**, **All** and the set of the predicate symbols of the logical language (**dog**, **chases**, etc. in the examples). Their types are described in table 2.

Note we have three existential quantifiers $\exists_l$, $\exists_h$ and $\exists_e$, but we usually note them only $\exists$. Moreover, to keep with the usual logical notation we write $\exists x\, P$ instead of $\exists(\lambda x.P)$ where $x$ is a free variable of $P$.

Finally, to define the ACG $\mathcal{G}'$, we need the lexicon $\mathcal{L}'$. It transforms the types from $\Lambda_1$ as follows:

$$\begin{aligned}
\mathcal{L}'(\mathbf{N}_S) &= (e \to h \to l \to t) \multimap (h \to l \to t)\\
\mathcal{L}'(\mathbf{N}_A) &= (e \to h \to l \to t)\\
&\quad \multimap (e \to h \to l \to t) \multimap (h \to l \to t)\\
\mathcal{L}'(\mathbf{S}_S) &= h \to l \to t\\
\mathcal{L}'(\mathbf{S}_A) &= (e \to h \to l \to t)\\
&\quad \multimap (e \to h \to l \to t)\\
\mathcal{L}'(\mathbf{VP}_A) &= (h \to l \to t) \multimap (h \to l \to t)
\end{aligned}$$

Contrary to $\Lambda_2$, that model derived trees and generates linear terms, we use in $\Lambda_2'$ non-linear terms, as the intuitionnistic $\to$ shows. The definition of $\mathcal{L}'$ on the terms justifies it. We shall introduce this definition in the next sections, illustrating different linguistic phenomena.

### 3.1 Quantification

We start with the classical example of quantitification. When dealing with quantifiers as adjunct (Abeillé, 1993), where quantifier is adjoined to the noun, quantifiers are separated from the verb by the noun in the derivation trees. Then the problem of the proposition coming from the **VP** to be part of the scope of the quantifiers arises. (Kallmeyer, 2002) proposes to enrich the derivation trees with additional links to take this kind of linking into account.

We propose to deal with this kind of problems following the Montague's approach of quantification (Montague, 1974): the subject is an argument of the verb, but it is also a higher order function which has the verb predicate as argument. So the lexicon for the ACG $\mathcal{G}'$ could define :

$$\begin{aligned}
\mathcal{L}'(c_{dog}) &= \lambda q.q(\lambda x h l.h \geq l \wedge l : \mathbf{dog}(x))\\
\mathcal{L}'(c_{cat}) &= \lambda q.q(\lambda x h l.h \geq l \wedge l : \mathbf{cat}(x))\\
\mathcal{L}'(c_{chases}) &= \lambda b a s o.s(b(\lambda x.a(o(\lambda y h' l'.h' \geq l'\\
&\quad \wedge l' : \mathbf{chases}(x, y)))))\\
\mathcal{L}'(c_{every}) &= \lambda r p.\lambda h l.\exists h_1 l_1 l_2 l_3 v_1 (h \geq l_2\\
&\quad \wedge l_2 : \mathbf{All}(v_1, l_3) \wedge l_3 : \mathbf{Imp}(l_1, h_1)\\
&\quad \wedge h_1 \geq l \wedge r\, v_1\, h\, l_1 \wedge p\, v_1\, h\, l)\\
\mathcal{L}'(c_{some}) &= \lambda r p.\lambda h' l'.\exists h_1' l_1' l_2' l_3' v_1' (h' \geq l_2'\\
&\quad \wedge l_2' : \mathbf{Ex}(v_1', l_3') \wedge l_3' : \mathbf{And}(l_1', h_1')\\
&\quad \wedge h_1' \geq l' \wedge r\, v_1'\, h'\, l_1' \wedge p\, v_1'\, h'\, l')
\end{aligned}$$

It's easy to check that the translation from the abstract term, or the derivation tree in our sense, $t = c_{chases} I_{\mathbf{S}} I_{\mathbf{VP}} (c_{dog} c_{every})(c_{cat} c_{some})$ by $\mathcal{L}'$ has the expected form:

$$\begin{aligned}
\mathcal{L}'(c_{dog} c_{every}) &= \lambda p.\lambda h l.\exists h_1 l_1 l_2 l_3 v_1 (h \geq l_2\\
&\quad \wedge l_2 : \mathbf{All}(v_1, l_3) \wedge l_3 : \mathbf{Imp}(l_1, h_1)\\
&\quad \wedge h_1 \geq l \wedge h \geq l_1 \wedge l_1 : \mathbf{dog}(v_1)\\
&\quad \wedge p\, v_1\, h\, l)\\
\mathcal{L}'(c_{cat} c_{some}) &= \lambda p.\lambda h' l'.\exists h_1' l_1' l_2' l_3' v_1' (h' \geq l_2'\\
&\quad \wedge l_2' : \mathbf{Ex}(v_1', l_3') \wedge l_3' : \mathbf{And}(l_1', h_1')\\
&\quad \wedge h_1' \geq l' \wedge h' \geq l_1' \wedge l_1' : \mathbf{cat}(v_1')\\
&\quad \wedge p\, v_1'\, h'\, l')\\
\mathcal{L}'(c_{chases} I_{\mathbf{S}} I_{\mathbf{VP}}) = &\quad \lambda s o.s(\lambda x.o(\lambda y h' l'.h' \geq l'\\
&\quad \wedge l' : \mathbf{chases}(x, y)))
\end{aligned}$$

Hence for $\mathcal{L}'(t)$ we have:

$$\begin{aligned}
&\lambda h l.\exists h_1 l_1 l_2 l_3 v_1 (h \geq l_2 \wedge l_2 : \mathbf{All}(v_1, l_3)\\
&\wedge l_3 : \mathbf{Imp}(l_1, h_1) \wedge h_1 \geq l \wedge h \geq l_1 \wedge l_1 : \mathbf{dog}(v_1)\\
&\wedge \exists h_1' l_1' l_2' l_3' v_1' (h \geq l_2' \wedge l_2' : \mathbf{Ex}(v_1', l_3')\\
&\wedge l_3' : \mathbf{And}(l_1', h_1') \wedge h_1' \geq l \wedge h \geq l_1' \wedge l_1' : \mathbf{cat}(v_1')\\
&\quad \wedge h \geq l \wedge l : \mathbf{chases}(v_1, v_1')))
\end{aligned}$$

recovering the one from the figure 3 (modulo variable renaming). To deal with quantification in this example, we don't add any extra-link to the derivation tree (or abstract term) ones, contrary to (Kallmeyer, 2002). Both the subject (the $s$ variable in $\mathcal{L}'(c_{chases})$) and the object parameter (the $o$ variable) are considered as the real functors, applyed to the relation **chases** as in $s(\cdots(o(\cdots \mathbf{chases}(x, y)\cdots)))$. This implies that **N**s and **NP**s have higher-order types (see also the semantic term associated to entities in section 3.4). This is reminiscent to Montague's approach (Montague, 1974).

A term like $\mathcal{L}'(c_{chases})$ also shows the exact contribution of every node. For instance, the $b$ variable stands for the semantic contribution of the **S** node, whereas the $a$ variable stands for the semantic contribution of the **VP**. That is the former can act both on the predicate and its argument (see the type of $\mathcal{L}'(\mathbf{S}_A)$), whereas the latter can only modify the whole relation. The next sections illustrate this point, with adverbs and raising verbs. Then, modelling verbs with phrasal arguments, we show how the $b$ variable can act.

In the sequel of the paper, whenever we introduce a new term which has a similar constrution to a previous one, we don't give its explicit definition (e.g. *loves*, similar to *chases*).

### 3.2 Adverbs

In the semantic representation we associate to $c_{chases}$ in the previous section, we see, between the subject $s$ and the "**VP** relation", an argument $a$. Its type ($\mathcal{L}'(\mathbf{VP}_A) = (h \to l \to t) \multimap (h \to l \to t)$) shows it is a verb modifier. So let us introduce a new consant $c_{usually} : \mathbf{VP}_A \multimap$

| | | |
|---|---|---|
| $\geq$ | $: h \to l \to t$ | specifies the underspecification constraints |
| $:$ | $: l \to p \multimap t$ | labels the logical predicates |
| $\wedge$ | $: t \multimap t \multimap t$ | conjunct of descriptions |
| $\exists_l$ | $: (l \to t) \multimap t$ | existential quantifier on labels |
| $\exists_h$ | $: (l \to t) \multimap t$ | existential quantifier on holes |
| $\exists_e$ | $: (e \to t) \multimap t$ | existential quantifier on entities |
| **And**, **Imp** | $: l \to h \to p$ | conjonction and implication in the embedded logical language |
| **dog**, **cat** | $: e \to p$ | predicates in the embedded logical language |
| **chases** | $: e \to e \to p$ | predicate in the embedded logical language |

Table 2: Typing of constants of $\Lambda'_2$

$\mathbf{VP}_A \in C_1$. We can associate it, with $\mathcal{L}'$, to the term:

$$\lambda a.\lambda r.\lambda hl.\exists h_1 l_1 (r\, h\, l \wedge h \geq l_1 \wedge l_1 : \mathbf{U}(h_1) \\ \wedge h_1 \geq l \wedge a(\lambda h' l'.h' \geq l') h\, l_1)$$

Its first argument, $a$, correspond to the verb modifier that could also be adjoined to this node (for instance an other adverb *alledgelly*). The second argument, $r$, corresponds to the verb predicate it modifies. Here, it is $l$ that the adverb **U** should also dominate ($h_1 \geq l$). Then, to express that *usually* is an opaque modifier is just indicating that the label $l_1$ of **U** has to be the lowest point in the modification induced by $a$. That is $l_1$ is also the label argument of $a$.

So $c_{usually}(c_{alledgedly} I_{\mathbf{VP}})$ is mapped to

$$\lambda r.\lambda hl.\exists h_1 l_1 (r\, h\, l \wedge h \geq l_1 \wedge l_1 : \mathbf{U}(h_1) \\ \wedge h_1 \geq l \wedge \exists h'_1 l'_1 (h \geq l_1 \wedge h \geq l'_1 \wedge l'_1 : \mathbf{A}(h'_1) \\ \wedge h'_1 \geq l_1))$$

where every subformula of $h'_1$ is a subformula of **A**. Since $h'_1$ dominates $l_1$ which is the label of **U**, $\mathbf{U}(h_1)$ is always a subformula of **A**.

As mentionned in (Gardent and Kallmeyer, 2003), there are adverbs that would not have this opaque behaviour and rather pass the label of the verb predicate to other possibles modifiers. In this case, the argument of $a$ is not $l_1$, but simply $l$. We illustrate it in the next example, even if not on adverbs.

### 3.3 Raising Verbs

Raising verbs like *seems* have been modelled in TAGs as adverbs. We can use exactly the same semantic encoding as for adverbs, except that this time it is not considered as opaque. Hence its associated term in $\Lambda'_2$ is:

$$\lambda a.\lambda r.\lambda hl.\exists h_1 l_1 (r\, h\, l \wedge h \geq l_1 \wedge l_1 : \mathbf{seems}(h_1) \\ \wedge h_1 \geq l \wedge a(\lambda h' l'.h' \geq l') h\, l)$$

### 3.4 Verbs with Phrasal Arguments

Going upward in the syntactic tree, we can now try to model expressions that act on **S** nodes like *claims* (see table 3). Coming back to our modelling of *chases*, we had a $b$ argument of type $\mathcal{L}'(\mathbf{S}_A) = (e \to h \to l \to t) \multimap (e \to h \to l \to t)$. So we can associate to a term $c_{claims} : \mathbf{N}_S \multimap \mathbf{S}_A \multimap \mathbf{S}_A \in \Lambda_1$ a term in $\Lambda'_2$:

$$\lambda spr.\lambda y.p(s(\lambda x hl.\exists l_1 h_1 (h \geq l_1 \wedge \\ l_1 : \mathbf{claims}(x, h_1) \wedge r y h_1 l)))$$

which specifies that $x$ claims something, the latter being dominated by $h_1$ (hence **claims**).

So for instance, an expression *Paul claims John loves Mary* would give the abstract term $c_{loves}(c_{claims} c_{Paul} I_{\mathbf{S}}) I_{\mathbf{VP}} c_{John} c_{Mary}$ and its underspecified representation ($\mathcal{L}'(c_{Paul}) = \lambda P.P\mathbf{p}$):

$$\lambda hl.\exists l_1 h_1 (h \geq l_1 \wedge l_1 : \mathbf{claims}(\mathbf{p}, h_1) \\ \wedge h_1 \geq l \wedge l : \mathbf{loves}(\mathbf{j}, \mathbf{m})))$$

because

$$\begin{aligned} \mathcal{L}'(c_{claims} c_{Paul}) &= \lambda r.\lambda y.\lambda hl.\exists l_1 h_1 (h \geq l_1 \\ &\quad \wedge l_1 : \mathbf{claims}(\mathbf{p}, h_1) \wedge r y h_1 l) \\ \mathcal{L}'(c_{loves} t I_{\mathbf{VP}} \\ c_{John} c_{Mary}) &= (\lambda P.P\mathbf{j})(t(\lambda x.(\lambda Q.Q\mathbf{m}) \\ &\quad (\lambda y h' l'.h' \geq l' \wedge \\ &\quad l' : \mathbf{loves}(x, y)))) \\ &= (\lambda P.P\mathbf{j})(t(\lambda x.\lambda h' l'.h' \geq l' \\ &\quad \wedge l' : \mathbf{loves}(x, \mathbf{m}))) \end{aligned}$$

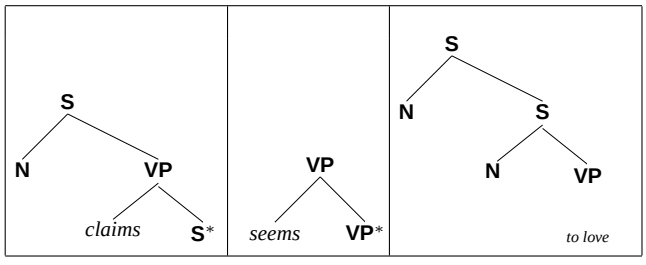


Table 3: Few more trees

Let us now illustrate the long distance dependancy behaviour, together with phrasal arguments. We can see that

if the syntactic properties of the infinitive *to love* (see table 3) really differs from the ones of *loves*, their semantic counterpart only differs in the order of argument (and an extra $\mathcal{L}'(\mathbf{S}_A)$ whose role should be precised). We can naturally associate to $\mathcal{L}'(c_{\textit{to love}})$ the term:

$$\lambda baos.s(b(\lambda x.a(o(\lambda yh'l'.h' \geq l' \wedge l' : \mathbf{loves}(x, y)))))$$

Then analyzing a long distance dependency *Mary Paul claims John seems to love* is the same as analyzing the previous example, except that the $I_{\mathbf{VP}}$ term is replaced by $c_{\textit{seems}}$ and the order of the other arguments is exchanged: $c_{\textit{loves}}(c_{\textit{claims}}c_{\textit{Paul}})(c_{\textit{seems}}I_{\mathbf{VP}})c_{\textit{Mary}}c_{\textit{John}}$. The contribution of $\mathcal{L}'(c_{\textit{seems}}I_{\mathbf{VP}})$ to $\mathcal{L}'(c_{\textit{love}})$ is just adding the conjonction of (modulo the variable renaming) $\exists h_2 l_2(h_1 \geq l \wedge l : \mathbf{loves}(\mathbf{j}, \mathbf{m}) \wedge h_1 \geq l_2 \wedge l_2 : \mathbf{seems}(h_2) \wedge h_2 \geq l)$ instead of only $h_1 \geq l \wedge l : \mathbf{loves}(\mathbf{j}, \mathbf{m}))$ so that we finally have:

$$\begin{array}{l}\lambda hl.\exists l_1 h_1(h \geq l_1 \wedge l_1 : \mathbf{claims}(\mathbf{p}, h_1)\\ \wedge \exists h_2 l_2(h_1 \geq l \wedge l : \mathbf{loves}(\mathbf{j}, \mathbf{m}) \wedge h_1 \geq l_2\\ \wedge l_2 : \mathbf{seems}(h_2) \wedge h_2 \geq l)\end{array}$$

which is the expected result.

### 3.5 Wh-questions

This section provides an example of an adjunction occurring on the root node of an auxiliary tree which is itself adjoined to a third tree. The expression *who does Paul think John said Bill liked,* can be analyzed with the constants $c_{\textit{who}} : \mathbf{WH}_S \in \Lambda_1$ and $c_{\textit{liked}} : \mathbf{S}_A \multimap \mathbf{VP}_A \multimap \mathbf{WH}_S \multimap \mathbf{N}_S \multimap \mathbf{S}_S \in \Lambda_1$, that correspond to the trees of figure 4. The two other constants $c_{\textit{does think}}$ and $c_{\textit{said}}$, corresponds to the auxiliary trees of the same figure and the derivation tree is $c_{\textit{liked}}(c_{\textit{said}}c_{\textit{John}}(c_{\textit{does think}}c_{\textit{Paul}}I_{\mathbf{S}}))I_{\mathbf{VP}}c_{\textit{who}}c_{\textit{Bill}}$.

Then, we can extend $\mathcal{L}'$ as follows:

$$\begin{array}{rl}\mathcal{L}'(c_{\textit{who}}) = & \lambda phl.\exists v_1 h_1'' l_1''(h \geq l_1''\\ & \wedge l_1'' : \mathbf{W}(v_1, h_1'') \wedge h_1'' \geq l \wedge pv_1 h_1'' l)\\ \mathcal{L}'(c_{\textit{liked}}) = & \lambda baos.o(b(\lambda y.a(s(\lambda xh'l'.h' \geq l'\\ & \wedge l' : \mathbf{liked}(x, y)))))\\ \mathcal{L}'(c_{\textit{said}}) = & \lambda sbr.b(\lambda y.s(\lambda xhl.\exists h_1 l_1(h \geq l_1\\ & \wedge l_1 : \mathbf{S}(x, h_1) \wedge h_1 \geq l \wedge ryh_1 l)))\\ \mathcal{L}'(c_{\textit{does think}}) = & \lambda sbr'.b(\lambda y.s(\lambda xhl.\exists h_1' l_1'(h \geq l_1'\\ & \wedge l_1' : \mathbf{T}(x, h_1') \wedge h_1' \geq l \wedge r'yh_1' l)))\end{array}$$

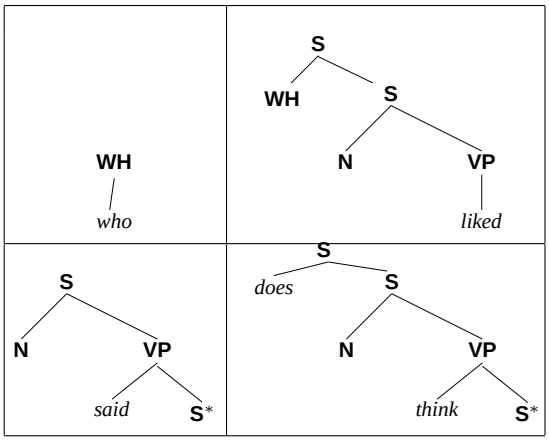


Figure 4: Wh-question example

Then, we have :

$$\begin{array}{rl}\mathcal{L}'(c_{\textit{does think}}c_{\textit{Paul}}I_{\mathbf{S}}) = & \mathcal{L}'(t_0)\\ = & \lambda r'\lambda yhl.\exists l_1' h_1'(h \geq l_1'\\ & \wedge l_1' : \mathbf{T}(\mathbf{p}, h_1') \wedge h_1' \geq l\\ & \wedge r'yh_1' l)\\ \mathcal{L}'(c_{\textit{said}}c_{\textit{John}}t_0 I_{\mathbf{S}}) = & \mathcal{L}'(t_1)\\ = & \lambda r.\lambda yhl.\exists l_1' h_1'(h \geq l_1'\\ & \wedge l_1' : \mathbf{T}(\mathbf{p}, h_1') \wedge h_1' \geq l\\ & \wedge \exists h_1 l_1(h_1' \geq l_1 \wedge l_1 : \mathbf{S}(\mathbf{j}, h_1)\\ & \wedge h_1 \geq l \wedge ryh_1 l))\end{array}$$

This yields the following result:

$$\begin{array}{rl}\mathcal{L}'(c_{\textit{liked}}t_1 I_{\mathbf{VP}}c_{\textit{who}}c_{\textit{Bill}}) = & (\lambda o.o(\lambda yhl.\exists l_1' h_1'(h \geq l_1'\\ & \wedge l_1' : \mathbf{T}(\mathbf{p}, h_1') \wedge h_1' \geq l\\ & \wedge \exists h_1 l_1(h_1' \geq l_1 \wedge l_1 : \mathbf{S}(\mathbf{j}, h_1)\\ & \wedge h_1 \geq l \wedge h_1 \geq l\\ & \wedge l : \mathbf{liked}(\mathbf{b}, y)))))\mathcal{L}'(c_{\textit{who}})\\ = & \lambda hl.\exists v_1 h_1'' l_1''(h \geq l_1''\\ & \wedge l_1'' : \mathbf{W}(v_1, h_1'') \wedge h_1'' \geq l\\ & \wedge \exists l_1' h_1'(h_1'' \geq l_1'\\ & \wedge l_1' : \mathbf{T}(\mathbf{p}, h_1') \wedge h_1' \geq l\\ & \wedge \exists h_1 l_1(h_1' \geq l_1 \wedge l_1 : \mathbf{S}(\mathbf{j}, h_1)\\ & \wedge h_1 \geq l \wedge h_1 \geq l\\ & \wedge l : \mathbf{liked}(\mathbf{b}, v_1))))\end{array}$$

which is the expected one, with $\mathbf{W}$ binding the variable $v_1$ and dominating $\mathbf{T}$, itself dominating $\mathbf{S}$, itself dominating $\mathbf{liked}(\mathbf{b}, v_1)$.

### 3.6 Control Verbs

Control verbs, as presented in (Gardent and Kallmeyer, 2003) or (Frank and van Genabith, 2001), with adjunc-

tion on a **S** node (see table 4) to produce an expression like *John tries to sleep*, with the adjunction of *tries to* on *sleep*, is a problem for our approach.

Indeed, it is build from the term $c_{sleep}(c_{tries\ to} I_{\mathbf{VP}} c_{John} I_{\mathbf{S}}) I_{\mathbf{VP}}$ and the typing discipline makes $t = c_{tries\ to} I_{\mathbf{VP}} c_{John} I_{\mathbf{S}}$ of type $\mathbf{S}_A$, hence $\mathcal{L}'(t)$ of type $(e \to h \to l \to t) \multimap e \to h \to l \to t$. If it is clear that the first argument of type $(e \to h \to l \to t)$ concerns the **sleep** predicate (with something like $\lambda xhl.h \geq l \wedge l : \mathbf{sleep}(x)$), the result should not have any $e$ possible argument (it has been filled with **j**).

In other worlds, if we look at adjunctions on **S** nodes in previous sections, the subtrees always lack an **N** (*John seems to love* $x$, or *John said Bill liked* $x$) and are always transformed into a subtree lacking an **N** too (*Paul claims John seems to love* $x$, or *does Paul think John said Bill liked* $x$). This is not the case anymore with control verbs where the subtree for $x$ *sleep* turns into *John tries to sleep*.

So control verbs cannot be dealt with directly that way with our techniques. We need for instance to differentiate the $\mathbf{S}_A$ type into the usual one $(e \to h \to l \to t) \multimap e \to h \to l \to t$ and another one $(e \to h \to l \to t) \multimap h \to l \to t$. This could be done with a special $\mathbf{S}_{\text{Pro}}$ node, or with an extended type system (for instance additives of linear logic to manage disjunctive types). But this requires further investigation and goes beyond this article.

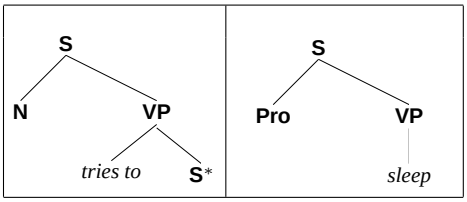


Table 4: Derived trees for control verbs

## Conclusion

We propose to reconsider semantic representation computation for TAG from the derivation trees. But derivation trees here are understood as abstract terms of ACGs, even if the informations born by each of the formalism are essentially the same. Whereas they hold the specification of how trees should combine, the locality of computing the meanings held by the different nodes is described in the object vocabulary. It obviates the addition of extra links to manage scoping and shows that derivation trees, by themselves, are enough, even if further investigation are required to handle control verbs.

It also clearly defines the compositional aspects of building semantic representations with a clear and modular distinction between syntax and semantics. The latter point lacks in the derived tree approaches. Moreover, the mathemetical primitives we use are very simple (if expression not always are) and are the same both on the syntactic and the semantic side, and no external principles need to be added.

So, from the ACG point of view, both syntax and semantics are dealt with in an equivalent way: as object languages of the same abstract language. This is interesting because the computation engine to go from the object language to the abstract language in an ACG does not depend on the object language. So the underlying process remains the same for all that cases:

- to compute a derived tree, then a derivation tree, from a string;
- to compute a derivation tree from a URL formula;
- to compute a derived tree, then a string, from a derivation tree;
- to compute an URL formula from a derivation tree.

So that going from one to the other (parsing or generation, in the usual sense) is as difficult (or as easy) as going the other way. Of course, on the semantic side, it means the initial point is an URL formula, and it gives no hint on how to build it from an SRL formula, nor on how to deal with the logical equivalence (be it on the SRL or on the URL level).

Finally, it underlines the interesting feature of ACG to transport or transmit structures from one language to anoher, illustrated between a syntactic formalism and a semantic formalism for TAGs. As suggested by an anonymous referee, the same approach could be used to provide semantic representations to expressions belonging to $m$-linear context-free languages, since abstract terms have already been proposed for them (de Groote and Pogodalla, 2003).

## References

Anne Abeillé. 1993. *Les nouvelles syntaxes*. Armand Colin Éditeur, Paris.

Patrick Blackburn and Johan Bos. 2003. Computational semantics for natural language. `http://www.iccs.informatics.ed.ac.uk/~jbos/comsem/book1.html`. Course Notes for NASSLLI 2003.

Johan Bos. 1995. Predicate logic unplugged. In *Proceedings of the Tenth Amsterdam Colloquium*.

Marie-Hélène Candito and Sylvain Kahane. 1998. Can the tag derivation tree represent a semantic graph? an answer in the light of meaning-text theory. In *Proceedings of the Fourth International Workshop on Tree Adjoining Grammars and Related Framework (TAG+4)*, volume 98-12 of *IRCS Technical Report Series*.

Vincent Danos and Roberto Di Cosmo. 1992. The linear logic primer. http://www.pps.jussieu.fr/~dicosmo/CourseNotes/LinLog/. An introductory course on Linear Logic.

Philippe de Groote and Sylvain Pogodalla. 2003. $m$-linear context-free rewriting systems as abstract categorial grammars. In Richard Oehrle and James Rogers, editors, *MOL 8, proceedings of the eighth Mathematics of Language Conference*.

Philippe de Groote. 2001. 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.

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

Anette Frank and Josef van Genabith. 2001. Glue tag: Linear logic based semantics construction for ltag - and what it teaches us about the relation between lfg and ltag. In Miriam Butt and Tracy Holloway King, editors, *Proceedings of the LFG '01 Conference*, Online Proceedings. CSLI Publications. http://cslipublications.stanford.edu/LFG/6/lfg01.html.

Claire Gardent and Laura Kallmeyer. 2003. Semantic construction in feature-based tag. In *Proceedings of the 10th Meeting of the European Chapter of the Association for Computational Linguistics (EACL)*.

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science*, 50:1–102.

Aravind K. Joshi, Laura Kallmeyer, and Maribel Romero. 2003. Flexible composition in ltag: Quantifier scope and inverse linking. In Harry Bunt, Ielka van der Sluis, and Roser Morante, editors, *Proceedings of the Fifth International Workshop on Computational Semantics IWCS-5*.

Laura Kallmeyer. 2002. Using an enriched tag derivation structure as basis for semantics. In *Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+6)*.

Richard Montague, 1974. *The Proper Treatment of Quantification in Ordinary English*, chapter 1. In Portner and Partee (Portner and Partee, 2002).

Sylvain Pogodalla. 2004. Using and extending ACG technology: Endowing categorial grammars with an underspecified semantic representation. In *Proceedings of Categorial Grammars 2004, Montpellier*, June.

Paul Portner and Barbara H. Partee, editors. 2002. *Formal Semantics: The Essential Readings*. Blackwell Publishers.

Yves Schabes and Stuart M. Shieber. 1994. An alternative conception of tree-adjoining derivation. *Computational Linguistics*, 20(1):91–124.