HAL

archives-ouvertes.fr

Advances in Abstract Categorial Grammars: Language Theory and Linguistic Modeling. ESSLLI 2009 Lecture Notes, Part II

Sylvain Pogodalla

▶ To cite this version:

Sylvain Pogodalla. Advances in Abstract Categorial Grammars: Language Theory and Linguistic Modeling. ESSLLI 2009 Lecture Notes, Part II. 2009. hal-00749297

HAL Id: hal-00749297
https://hal.inria.fr/hal-00749297

Preprint submitted on 7 Nov 2012

Advances in Abstract Categorial Grammars: Language Theory and Linguistic Modeling

Lecture Notes, Part II[1]

The Syntax-Semantics Interface: an ACG Perspective

Sylvain Pogodalla
LORIA/INRIA Nancy – Grand Est
France
http://www.loria.fr/~pogodall/

Contents

Foreword

These are the lecture notes of the ESSLLI 2009[1] second week course[2] on Abstract Categorial Grammar. This was an advanced course, while an introductory course was given the first week: Introduction to Abstract Categorial Grammars: Foundations and main properties[3], delivered by Philippe de Groote and Sylvain Salvati.

An up-to-date version of these notes, possible errata, and generally ACG papers, can be found at the ACG homepage: http://calligramme.loria.fr/acg/. At this URL, the ACG Development Toolkit is also available and downloadable as free software. It might be useful to run some of the examples given in these notes. In particular, some of the latter are gathered in a special file.

The Abstract Categorial Grammar (ACG) (de Groote, 2001), a grammar formalism based on the typed lambda calculus, elegantly generalizes and unifies a variety of grammar formalisms that have been proposed for the description of formal and natural languages. The first part of the course investigated formal-language-theoretic properties of "second-order" ACGs, a subclass of ACGs that have "context-free" derivations. Their generative capacity was precisely characterized. Lecture notes for this part are available at http://research.nii.ac.jp/~kanazawa/publications/esslli2009_lectures.pdf. An efficient Earley-style algorithm along the lines of (Kanazawa, 2007), suitable both for parsing and generation, was then presented.

The second part of the course (these present notes) presents linguistic applications of ACGs and gives various illustrations of how ACGs provide flexible and explicit ways to model the syntax-semantics interface of natural language.

Part of these notes are new, some others are almost directly taken from published papers.

---

[1] http://esslli2009.labri.fr/
[2] http://esslli2009.labri.fr/course.php?id=52
[3] http://esslli2009.labri.fr/course.php?id=108

Outline

In these notes, after a brief reminders of the main ACG definitions (Chapter 1), we explore the different ways ACG can be composed. The aim is twofold:

1. Offering a unified view on various formalisms (TAG family in Chapter 2, Categorial Grammar in Chapter 3 and Convergent Grammar in Chapter 4), we may compare them and possibly import some solutions for the syntax-semantics interface from one to another. For instance, we can import the CG and CVG approach to scope ambiguity to TAG.

2. Modularizing the different contributions of the grammatical formalism, we can for instance:

   - separate what is relevant for computing semantics to what is relevant for controlling derivations (hence the generated languages) in TAG;
   - highlight and explicit the role of "syntax" in the CG formalism and in the CVG one;
   - modularize the encoding of some linguistic phenomena.

Chapter 2 gives an ACG account of TAG and MCTAG. The proposed architecture relates TAG with the formal-language-theoretic properties of ACG. In particular, it makes explicit how generative capacity can grow and where some control on the derivations has to be added. In particular, the fact that TAG and (set-local) MCTAG are mildly context-sensitive appears in the overall control of 2nd order ACG.

Then Chapter 3 presents an ACG view on (classical) Categorial Grammar. The aim in this chapter is to redistribute the syntactic and the semantic contributions of CG typing. In particular, it shows that the syntactic ambiguities that is required for semantic ambiguities in CG is a "compiled" point of view of the syntax-semantic interface.

Chapter 4 makes this point of view even more explicit when encoding CVG into ACG. In particular, the modelling of overt and covert movement makes explicit what are the respective contributions to syntax and semantics of such phenomena, and how these contributions can be shared. The recently pointed out notions of CVG such as "simple syntax" and "weak syntactocentrism" (Culicover and Jackendoff, 2005; Jackendoff, 2002) are expressed in the ACG framework and related to the ACG architecture for CG.

Chapter 1

Abstract Categorial Grammar: a (Very Brief) Reminder

1.1 Definitions

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. This general picture can of course be adapted to the need of the modeling. In order to be able to model non linearity (this is useful for semantics), we use an extension of the ACG with both *linear* and *non-linear* implication but the principles follow de Groote, 2001's definitions.[1]

Definition 1.1. *Let A be a set of atomic types. The set $\mathscr{T}(A)$ of* implicative types *build upon A is defined with the following grammar:*

$$\mathscr{T}(A) ::= A \,|\, \mathscr{T}(A) \multimap \mathscr{T}(A) \,|\, \mathscr{T}(A) \to \mathscr{T}(A)$$

Definition 1.2. *A* higher-order signature Σ *is a triple $\Sigma = \langle A, C, \tau \rangle$ where:*

- *A is a finite set of atomic types;*

- *C is a finite set of constants;*

- *$\tau : C \to \mathscr{T}(A)$ is a function assigning a types to constants.*

Definition 1.3. *Let X be an infinite countable set of λ-variables. The set $\Lambda(\Sigma)$ of λ-terms built upon a higher-order signature $\Sigma = \langle A, C, \tau \rangle$ is inductively defined as follows:*

- *if $c \in C$ then $c \in \Lambda(\Sigma)$;*

- *if $x \in X$ then $x \in \Lambda(\Sigma)$;*

- *if $x \in X$ and $t \in \Lambda(\Sigma)$ and x occurs free in t exactly once, then $\lambda^{o} x.t \in \Lambda(\Sigma)$;*

- *if $x \in X$ and $t \in \Lambda(\Sigma)$ and x occurs free in t, then $\lambda x.t \in \Lambda(\Sigma)$;*

- *if $t, u \in \Lambda(\Sigma)$ and the set of free variables of u and t are disjoint then $(t\,u) \in \Lambda(\Sigma)$.*

---

[1]Formal properties of this extension, as expressiveness and the computational properties, are beyond the scope of this chapter.

Note there is a linear λ-abstraction and a (usual) intuitionistic λ-abstraction. There also are the usual notion of α-conversion and β-reduction.

Given a higher-order signature Σ, the typing rules are given with an inference system whose judgments are of the following form: $\Gamma; \Delta \vdash_\Sigma t : \alpha$ where:

- Γ is a finite set of non-linear variable typing declaration;

- Δ is a finite set of linear variable typing declaration.

Both Γ and Δ may be empty. Here are the typing rules:

$$\frac{}{\Gamma; \vdash_\Sigma c : \tau(c)} \text{ (const.)}$$

$$\frac{}{\Gamma; x : \alpha \vdash_\Sigma x : \alpha} \text{ (lin. var.)} \qquad \frac{}{\Gamma, x : \alpha; \vdash_\Sigma x : \alpha} \text{ (var.)}$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda^o x.t : \alpha \multimap \beta} \text{ (lin. abs.)}$$

$$\frac{\Gamma, x : \alpha; \Delta \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda x.t : \alpha \to \beta} \text{ (abs.)}$$

$$\frac{\Gamma; \Delta_1 \vdash_\Sigma t : \alpha \multimap \beta \qquad \Gamma; \Delta_2 \vdash_\Sigma u : \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash_\Sigma (t\,u) : \beta} \text{ (lin. app.)}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma t : \alpha \to \beta \qquad \Gamma; \vdash_\Sigma u : \alpha}{\Gamma; \Delta \vdash_\Sigma (t\,u) : \beta} \text{ (app.)}$$

Definition 1.4. *Let $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ be two higher-order signatures, a lexicon $\mathscr{L} = \langle F, G \rangle$ from Σ_1 to Σ_2 is such that:*

- $F : A_1 \to \mathscr{T}(A_2)$. *We also note* $F : \mathscr{T}(A_1) \to \mathscr{T}(A_2)$ *its homomorphic extension*[2];

- $G : C_1 \to \Lambda(\Sigma_2)$. *We also note* $G : \Lambda(\Sigma_1) \to \Lambda(\Sigma_2)$ *its homomorphic extension;*

- F *and* G *are such that for all* $c \in C_1$, $\vdash_{\Sigma_2} G(c) : F(\tau_1(c))$ *is provable.*

We also use $\mathscr{L}$ instead of F or G.

Definition 1.5. *An* abstract categorial grammar *is a quadruple $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, S \rangle$ where:*

- $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ *and* $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ *are two higher-order signatures;*

- $\mathscr{L} : \Sigma_1 \to \Sigma_2$ *is a lexicon;*

- $S \in \mathscr{T}(A_1)$ *is the* distinguished type *of the grammar.*

---

[2]such that $F(\alpha \multimap \beta) = F(\alpha) \multimap F(\beta)$ and $F(\alpha \to \beta) = F(\alpha) \to F(\beta)$

1.2   Generated Languages and Orders

Definition 1.6. *Given a set of atomic type A, the* order *$o(\alpha)$ of a type α of $\mathcal{T}(A)$ is inductively defined as:*

- *$o(\alpha) = 1$ if $\alpha \in A$;*

- *$o(\alpha \rightarrow \beta) = \max o(\beta, \alpha + 1)$*

- *$o(\alpha \multimap \beta) = \max o(\beta, \alpha + 1)$*

The order of a typed term is the order of its type.

Definition 1.7. *The* order *of an ACG is the maximum of the order of its abstract constants.*
The order of the lexicon of an ACG is the maximum of the order of the realizations of its atomic types.

Definition 1.8. *Given an ACG $\mathscr{G}$, the* abstract language *is defined by*

$$\mathcal{A}(\mathscr{G}) = \{t \in \Lambda(\Sigma_1) \mid \vdash_{\Sigma_1} t : S \text{ is derivable}\}$$

The object language *is defined by*

$$\mathcal{O}(\mathscr{G}) = \{u \in \Lambda(\Sigma_2) \mid \exists t \in \mathcal{A}(\mathscr{G}) \text{ s.t. } u = \mathscr{L}(t)\}$$

Note that $\mathscr{L}$ binds the parse structures of $\mathcal{A}(\mathscr{G})$ to the concrete expressions of $\mathcal{O}(\mathscr{G})$. Depending on the choice of Σ_1, Σ_2 and $\mathscr{L}$, it can map for instance derivation trees and derived trees for TAG 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. Moreover, this link between an abstract and a concrete structure can apply not only to syntactical formalisms, but also to semantic formalisms Pogodalla, 2004.

A crucial point is that ACG can be mixed in different ways: in a transversal way, were two ACG 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 (figure 1.1 illustrates these composition between three ACG). The next chapters exemplify both of these usages.
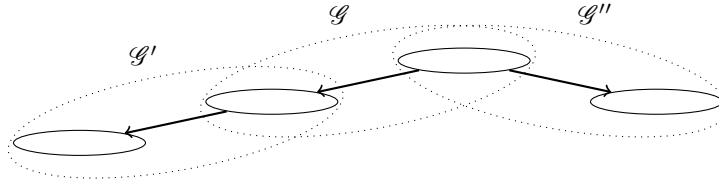
Figure 1.1: Ways of combining ACG

Chapter 2

TAG Formalisms as ACG

2.1 A Functional View on TAG

This section exemplifies the control ACGs provide on the derivation structures, hence on the object language, by defining the abstract language. It also shows the modular capabilities (function) composition between ACG gives. We choose to illustrate these features on the well-known framework of TAG (Joshi and Schabes, 1997; Joshi, Levy, and Takahashi, 1975).

Let's consider the following auxiliary tree[1]:   [tree: VP → apparently, VP*]   . When adjoined to another tree t at node VP_0[2], this auxiliary tree replace its own VP^* node by the subtree of t rooted at VP_0. If we call x the latter subtree, then we can consider the auxiliary tree as function associating a subtree x to a new tree   [tree: VP → apparently, x]   . Using the λ-calculus notation, the following term represents this tree:

$$\gamma'_{apparently} = \lambda^{\circ} x. \quad \text{[tree: VP → apparently, x]}$$

Let's now consider an initial tree such as: [tree: S → NP, VP → likes, NP]   . This tree can accept an adjunction at node VP. In this case, the argument it provides to the auxiliary tree (since the latter can be described as a function from trees to trees) is the subtree: [tree: VP → likes, NP]

Then we can describe the initial tree as a function whose parameter is an auxiliary. In the λ-calculus

---

[1] Here and in the rest of this paper, for sake of conciseness, we drop the category of the non terminal and the associated node in the tree.

[2] The 0 subscript only helps to remember at what node in the tree the adjunction took place.

notation, it gives:

$$\gamma'_{likes} = \lambda^{\mathrm{o}} a.\ \overset{\displaystyle S}{\underset{\displaystyle NP}{\diagup}}\ a\left(\begin{array}{c} VP \\ \diagup\ \diagdown \\ likes \qquad NP \end{array}\right)$$

The adjunction operation that yields the tree

$$\begin{array}{c} S \\ \diagup\ \diagdown \\ NP \qquad VP \\ \diagup\quad \diagdown \\ apparently \qquad VP \\ \diagup\ \diagdown \\ likes \qquad NP \end{array}$$

is then faithfully described by applying the term γ'_{likes} to $\gamma'_{apparently}$. Indeed:

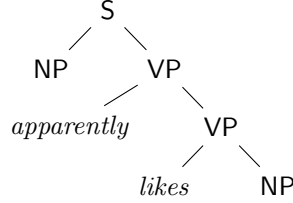$$\gamma'_{likes}\,\gamma'_{apparently} = \left(\lambda^{\mathrm{o}} a.\ \overset{S}{\underset{NP}{\diagup}}\ a\left(\begin{array}{c} VP \\ \diagup\ \diagdown \\ likes \quad NP \end{array}\right)\right)\left(\lambda^{\mathrm{o}} x.\ \begin{array}{c} VP \\ \diagup\ \diagdown \\ apparently \quad x \end{array}\right)$$

$$\rightarrow_{\beta}\ \overset{S}{\underset{NP}{\diagup}}\ \left(\left(\lambda^{\mathrm{o}} x.\ \begin{array}{c} VP \\ \diagup\ \diagdown \\ apparently \quad x \end{array}\right)\left(\begin{array}{c} VP \\ \diagup\ \diagdown \\ likes \quad NP \end{array}\right)\right)$$

$$\rightarrow_{\beta}\ \begin{array}{c} S \\ \diagup\ \diagdown \\ NP \qquad VP \\ \diagup\quad \diagdown \\ apparently \qquad VP \\ \diagup\ \diagdown \\ likes \qquad NP \end{array}$$

Of course, the tree for *apparently* itself should accept some adjunction at its VP node. So it is rather described with the term:

$$\gamma_{apparently} = \lambda^{\mathrm{o}} ax.a\left(\begin{array}{c} VP \\ \diagup\ \diagdown \\ apparently \quad x \end{array}\right)$$

In order to use $\gamma_{apparently}$ without performing any adjunction, we can use the identity $I = \lambda^{\mathrm{o}} x.x$ as auxiliary tree, so we have: $\gamma'_{apparently} = \gamma_{apparently} I$

Let's now consider the substitution operation. Actually, the tree

$$\begin{array}{c} S \\ \diagup\ \diagdown \\ NP \qquad VP \\ \diagup\ \diagdown \\ likes \quad NP \end{array}$$
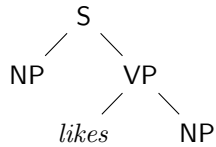
not only accept adjunctions on its VP node, but also substitutions[3] on the two NP nodes. It means it has two additional arguments s and o, that also are trees, that are substituted at the two NP nodes. This tree is then more faithfully represented by the following term[4]:

$$\gamma_{likes} = \lambda^{\mathrm{o}} aso.\ \overset{\displaystyle S}{\underset{\displaystyle s}{\diagup}}\ a\left(\begin{array}{c} VP \\ \diagup\ \diagdown \\ likes \quad o \end{array}\right)$$

---

[3]Substitution nodes are usually represented with a $\downarrow$. We drop this notation here.

[4]We use the convention for the argument list that auxiliary tree parameter, *i.e.* function from tree to tree, come in the first place, then substituted tree, *i.e.* simple trees, come in the second place

Let's assume we also have the following terms: $\gamma_{John} = \begin{smallmatrix} \mathsf{NP} \\ | \\ John \end{smallmatrix}$ and $\gamma_{Mary} = \begin{smallmatrix} \mathsf{NP} \\ | \\ Mary \end{smallmatrix}$. Note that being constant terms, because they don't have any parameter, they forbid any adjunction at their NP node.

Then we can build for instance:

The attentive reader may have notice that γ_{likes} does not provide a parameter for the S node (the root). It means that no adjunction can occur here. The mechanism is now known: just put another λ^o, and make the whole tree a parameter of this new variable. However, in the rest of this paper, for sake of legibility, we usually skip additional parameters when they are irrelevant for the given examples.

Collecting the previously introduced terms, together with a type τ as the type of trees, we have the constants of Table 2.1. Note that in this representation, leaves are either terminal symbols or variables.

$$
\begin{array}{llll}
\gamma_{John} & : \tau & \gamma_{likes} & : (\tau \multimap \tau) \multimap \tau \multimap \tau \multimap \tau \\
\gamma_{Mary} & : \tau & \gamma_{apparently} & : (\tau \multimap \tau) \multimap \tau \multimap \tau \\
I & : \tau \multimap \tau & &
\end{array}
$$
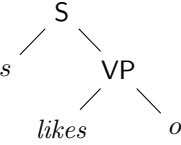
Table 2.1: Typing of the constants describing derived trees

2.2   TAG as ACG

We need another ingredient before we can encode TAG into ACG. We need to explicit the coding of trees using λ-terms build on a signature Σ_{trees}. Indeed, there is no such things like edges in terms. However, the encoding is straightforward:

- First, there is only one atomic type, the type of tree τ.

- Second, for any terminal w, we introduce a constant w of type τ.

- Then, for any non-terminal X of arity n of the ranked alphabet used to define trees, we introduce a constant X_n of type $\underbrace{\tau \multimap \cdots \multimap \tau}_{n \text{ times}} \multimap \tau$.

2.2.1 Overview

The encoding of

```
        S
       / \
      s   VP
         /  \
      likes   o
```

is then the term of $\Lambda(\Sigma_{trees})$: $\mathsf{S}_2\, s\, (\mathsf{VP}_2\, (\mathsf{V}_1\, likes)\, o)$. Then we can define the terms we introduced in the previous as in table 2.2.

$$
\begin{array}{llll}
\gamma_{\textit{John}} & = \mathsf{NP}_1\, \textit{John} & : \tau & \\
\gamma_{\textit{Mary}} & = \mathsf{NP}_1\, \textit{Mary} & : \tau & \\
I & = \lambda^{\circ} x.x & : \tau \multimap \tau &
\end{array}
\qquad
\begin{array}{lll}
\gamma_{\textit{likes}} & = \lambda^{\circ} aso.\mathsf{S}_2\, s\, (a(\mathsf{VP}_2\, likes\, o)) & : (\tau \multimap \tau) \multimap \tau \multimap \tau \multimap \tau \\
\gamma_{\textit{apparently}} & = \lambda^{\circ} ax.a\, (\mathsf{VP}_2\, apparently\, x) & : (\tau \multimap \tau) \multimap \tau \multimap \tau
\end{array}
$$

Table 2.2: Definition of the terms representing derived trees
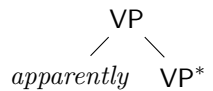
Remark 2.1. *The previous terms enable the construction of terms that the corresponding TAG would accept. For instance, the term $\gamma_{\textit{apparently}}\, I\, \gamma_{\textit{John}}$ is a well-formed term of $\Lambda(\Sigma_{trees})$. However, the corresponding tree*

```
        VP
       /  \
apparently  NP
            |
          John
```

would not be produced by the TAG. Indeed, we need to provide further control on the way terms of $\Lambda(\Sigma_{trees})$ are formed. This is the role of the ACG $\mathscr{G}_{typed\ trees}$, whose object vocabulary is Σ_{trees} and whose abstract vocabulary $\Sigma_{derivations}$ is to be introduced next.

In the encoding of trees we gave, using only the atomic type τ, we lost the typing information on trees that would prevent the attachment of a tree rooted by an NP at a location were we expect a VP.

We re-introduce the typing control using an ACG. Remember that the object language generated by an ACG is the image of the abstract language by the lexicon. So the solution here is to have the "correct" trees in the domain of the lexicon, and the "incorrect" trees in its complement in $\Lambda(\Sigma_{trees})$. So we re-introduce the types (the syntactic categories) at the abstract level defining $\Sigma_{derivations}$ with :

- The atomic types are the non-terminals

- For each tree in the TAG we introduce a constant.

Let's take for instance the constant $c_{\textit{apparently}}$ corresponding to the auxiliary tree for *apparently*:

```
         VP
        /  \
 apparently  VP*
```

. Just as in Section 2.1, we consider it as a function taking a VP as argument, and returning a VP. And if we want to integrate the fact that some adjunction can take place at the root node, we also specify that it has a $(\mathsf{VP} \multimap \mathsf{VP})$ parameter. So we come with the following type:
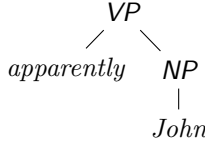
$$c_{\textit{apparently}} : (\mathsf{VP} \multimap \mathsf{VP}) \multimap \mathsf{VP} \multimap \mathsf{VP}$$

Note that it strongly relates to the type we gave to $\gamma_{\textit{apparently}}$.

On the other hand, we cannot use anymore the same fake auxiliary tree I, since it should depend on the category of the node it is adjoined to. So we need a term I_X for any atomic type X.

Applying the same approach to the terms we defined until now, we can type the constants of $\Sigma_{derivations}$ as in table 2.3.

While $\gamma_{\textit{apparently}}\, I\, \gamma_{\textit{John}}$ is a well-typed term, it is not anymore the case of the term $c_{\textit{apparently}}\, I_{\mathsf{VP}}\, c_{\textit{John}}$. This gives an idea of the way the object language, the one of TAG derived trees, is controlled by the abstract language.

We know have the abstract and the object vocabulary of $\mathscr{G}_{typed\ trees}$. We still need to define its lexicon $\mathscr{L}_{typed\ trees}$. First at the level of types:

$$
\begin{array}{ll}
c_{John} & : \mathsf{NP} \\
c_{Mary} & : \mathsf{NP} \\
I_X & : X \multimap X \text{ for every atomic type } X
\end{array}
\qquad
\begin{array}{ll}
c_{likes} & : (\mathsf{VP} \multimap \mathsf{VP}) \multimap \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S} \\
c_{apparently} & : (\mathsf{VP} \multimap \mathsf{VP}) \multimap \mathsf{VP} \multimap \mathsf{VP}
\end{array}
$$

Table 2.3: Typing of the constants of $\Sigma_{derivations}$

- For every atomic type X of $\Sigma_{derivations}$, $\mathscr{L}_{typed\ trees}(X) = \tau$ (we also note it $[\![X]\!]_{typedtrees} = \tau$, or $X :=_{typed\ trees} \tau$, or even $X := \tau$ if the ACG it relates to is obvious from the context).

- Then, for every constant $c_w \in \Sigma_{derivations}$, $c_w :=_{typed\ trees} \gamma_w$. Note that $\mathscr{L}_{typed\ trees}$ satisfies the typing criterion (the type of the image of a constant is the image of the type of this constant).

- Finally, for every constant $I_X : X \multimap X$, $I_X := I$.

Now $\mathscr{G}_{typed\ trees} = \langle \Sigma_{derivations}, \Sigma_{trees}, \mathscr{L}_{typed\ trees}, \mathsf{S} \rangle$ is completely defined.

For instance, $c_{likes}(c_{apparently}\ I_{\mathsf{VP}})\ c_{John}\ c_{Mary}$ is of type S, hence it belongs to $\mathcal{A}(\mathscr{G}_{typed\ trees})$, and its image by $\mathscr{L}_{typed\ trees}$ belongs to $\mathcal{O}(\mathscr{G}_{typed\ trees})$.
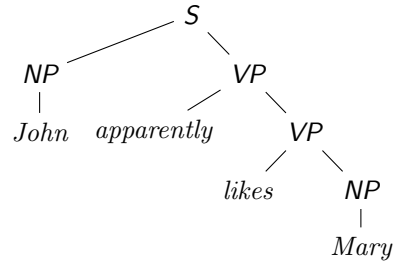
Exercice 2.1. *Check that* $c_{likes}(c_{apparently}\ I_{\mathsf{VP}})\ c_{John}\ c_{Mary} :=_{typed\ trees}$

Remark 2.2. *This presentation is actually very close to the one of (Vijay-Shanker, 1992) that introduces TAG as description of trees. The $(X \multimap X)$ types expresses that between the two X, there is a dominance that may not be immediate.*

2.2.2 Enriching the Lexicon

We're now in position of providing a richer lexicon, given in Table 2.4. When two entries differ only on the anchor (the terminal symbol, for instance for two adjectives, or two transitive verbs, etc.), we put only one of them.

Remark 2.3. *In this example, we use the modeling of the determiner with an auxiliary tree as in (Abeillé, 2002; XTAG Research Group, 2001). Note however that we encode the distinction between the N category with the feature det $= +$ and the one where it is det $= -$ as NP for the former and N for the latter. We could have used N[+] and N[−] as atomic types instead.*

This has several consequences:

1. *We distinguish two possible adjunction at node N for nouns: one for modifiers ($\mathsf{N} \multimap \mathsf{N}$) and one for determiners ($\mathsf{N} \multimap \mathsf{NP}$).*

2. *To force the adjunction of a determiner, we don't provide in the lexicon some identity constant of type ($\mathsf{N} \multimap \mathsf{NP}$). Hence, in order to use a N in a NP, the using a determiner is required. This corresponds to the OA (obligatory adjunction) described by features as in (Vijay-Shanker, 1992).*

3. *We should provide an additional entry of type ($\mathsf{N} \multimap \mathsf{N}$) $\multimap \mathsf{N}$ for the nouns in order to use them in constructs that don't require an NP. While this can seem awkward, it translates the two TAG categories N[det $= -$] and N[det $= +$] into N and NP resp.*

However, it does not change in essence the modelling.

We let the reader check[5] that this lexicon allows us to build the terms of $\mathcal{A}(\mathscr{G}_{typed\ trees})$ and get their interpretation as trees as described in Table 2.5.

If we look at the abstract terms (for instance $c_{chases}\ I_\mathsf{S}\ I_\mathsf{VP}\,(c_{dog}\ c_{every}\ I_\mathsf{N})\,(c_{cat}\ c_a\ I_\mathsf{N})$ of the second row of Table 2.5), since it uses only *application* (and not *abstraction*), we can represent it the tree of Figure 2.1(a). Dropping the fake adjunction of I_X terms yield a tree (Figure 2.1(b)) that looks pretty much the same as the traditional *derivation tree* of TAG. Indeed, the type of the parameters of an abstract constant plays the same role as the addresses of a node in an elementary tree in the definition of the derivation tree of (Weir, 1988) (plus an order on the addresses).

Since we now also have the derivation tree, as an abstract term, we can use it to build the semantic representation using the standard ACG architecture for the syntax-semantics interface.

(a) Abstract term (b) Derivation tree
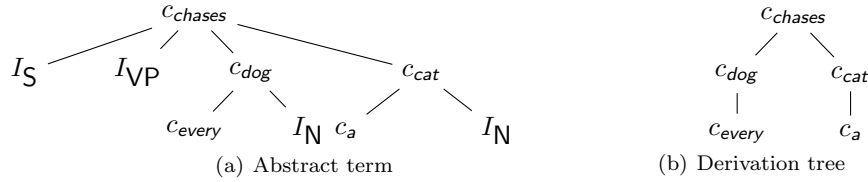
Figure 2.1: Representation of the derivation tree

2.2.3 Computing the Semantic Representation

In the standard ACG architecture for modelling the syntax-semantics interface, two ACGs share the same abstract level. Here we have a first ACG, $\mathscr{G}_{typed\ trees}$ whose abstract vocabulary is $\Sigma_{derivations}$, that encodes TAG derivations and TAG derived trees, and we need another $\mathscr{G}_{Log} = \langle \Sigma_{derivations}, \Sigma_{Log}, \mathscr{L}_{Log}, \mathsf{S}\rangle$, whose abstract vocabulary will be $\Sigma_{derivations}$ as well, that will encode the semantics we can build from these derivation trees.

To form the logical propositions, the object signature Σ_{Log} contains:

- The atomic types e and t.

- The logical constants[6]:

$$
\begin{array}{llll}
\wedge & : t \multimap t \multimap t & \vee & : t \multimap t \multimap t \\
\Rightarrow & : t \multimap t \multimap t & \neg & : t \multimap t \\
\exists & : (e \to t) \multimap t & \forall & : (e \to t) \multimap t
\end{array}
$$

- The non-logical constants:

$$
\begin{array}{llll}
\mathsf{sleep} & : e \multimap t & \mathsf{love, liked} & : e \multimap e \multimap t \\
\mathsf{claim, think} & : e \multimap t \multimap t & \mathsf{seem} & : e \multimap (e \multimap t) \multimap t \\
\mathsf{WHO} & : (e \multimap t) \multimap t & \mathsf{big, dog} & : e \multimap t
\end{array}
$$

The definition of $\mathscr{L}_{Log}$ mainly raises the question of translating the atomic types of $\Sigma_{derivations}$. We simply follow the very standard interpretation of these (syntactic) types into the (semantic) types given in (Montague, 1974):

$$
\begin{array}{llllll}
\mathsf{S} & : t & \mathsf{NP} & : (e \multimap t) \multimap t & \mathsf{N} & : e \to t \\
\mathsf{VP} & : e \multimap t & \mathsf{WH} & : (e \multimap t) \multimap t
\end{array}
$$

Following these typing constraints, we can complete the definition of the lexicon as in table 2.6.

As before, it is left to the reader (or to the ACG development toolkit) to check that the abstract terms of Table 2.5 are interpreted by $\mathscr{L}_{Log}$ as in Table 2.7.

---

[5]Using for instance the ACG Development Toolkit (available at http://www.loria.fr/equipes/calligramme/acg/).

[6]As usual, we note the λ-term $\exists\,(\lambda^o x.P)$ as $\exists x.P$. The same, *mutatis mutandis*, holds for $\forall$.

Constants of $\Sigma_{derivations}$	Its image by $\mathscr{L}_{typed\ trees}$	The corresponding TAG tree
c_{sleeps} $: (S \multimap S) \multimap (VP \multimap VP) \multimap NP \multimap S$	γ_{sleeps} $: (\tau \multimap \tau) \multimap (\tau \multimap \tau) \multimap \tau \multimap \tau$ $= \lambda^{\mathrm{o}} avs.a\,(S_2\, s\,(v\,(VP_1\, sleeps)))$	S → NP, VP; VP — *sleeps*
$c_{to\ love}$ $: (S \multimap S) \multimap (VP \multimap VP) \multimap NP \multimap NP \multimap S$	$\gamma_{to\ love}$ $: (\tau \multimap \tau) \multimap (\tau \multimap \tau) \multimap \tau \multimap \tau \multimap \tau$ $= \lambda^{\mathrm{o}} avso.S_2\, o$ $\qquad (a\,(S_2\, s\,(v\,(VP_1\, to\ love))))$	S → NP, S; S → NP, VP; VP — *to love*
c_{claims} $: (S \multimap S) \multimap (VP \multimap VP) \multimap NP \multimap S \multimap S$	γ_{claims} $: (\tau \multimap \tau) \multimap (\tau \multimap \tau) \multimap \tau \multimap \tau \multimap \tau$ $= \lambda^{\mathrm{o}} avsc.a\,(S_2\, s\,(a\,(VP_2\, claims\, c)))$	S → NP, VP; VP → *claims*, S*
c_{seems} $: (VP \multimap VP) \multimap VP \multimap VP$	γ_{seems} $: (\tau \multimap \tau) \multimap \tau \multimap \tau$ $= \lambda^{\mathrm{o}} vx.v\,(VP_2\, seems\, x)$	VP → *seems*, VP*
c_{liked} $: (S \multimap S) \multimap (VP \multimap VP) \multimap WH \multimap NP \multimap S$	γ_{liked} $: (\tau \multimap \tau) \multimap (\tau \multimap \tau) \multimap \tau \multimap \tau \multimap \tau$ $= \lambda^{\mathrm{o}} avws.S_2\, w$ $\qquad (a\,(S_2\, s\,(v\,(VP_1\, liked))))$	S → WH, S; S → NP, VP; VP — *liked*
$c_{does\ think}$ $: (S \multimap S) \multimap (VP \multimap VP) \multimap NP \multimap S \multimap S$	$\gamma_{does\ think}$ $: (\tau \multimap \tau) \multimap (\tau \multimap \tau)$ $\qquad \multimap \tau \multimap \tau \multimap \tau$ $= \lambda^{\mathrm{o}} avsc.a\,(S_2\, does$ $\qquad (S_2\, s\,(v\,(VP_2\, think\, c))))$	S → *does*, S; S → NP, VP; VP → *think*, S*
c_{who} $: WH$	γ_{who} $: \tau$ $= WH_1\, who$	WH — *who*
c_{big} $: (N \multimap N) \multimap N \multimap N$	γ_{big} $: (\tau \multimap \tau) \multimap \tau \multimap \tau$ $= \lambda^{\mathrm{o}} an.a\,(N_2\, big\, n)$	N → *big*, N*
c_{dog} $: (N \multimap NP) \multimap (N \multimap N) \multimap NP$	γ_{dog} $: (\tau \multimap \tau) \multimap (\tau \multimap \tau) \multimap \tau$ $= \lambda^{\mathrm{o}} da.d\,(a\,(N_1\, dog))$	N — *dog*
c_{a} $: N \multimap NP$	γ_{a} $: \tau \multimap \tau$ $= \lambda^{\mathrm{o}} n.NP_2\, a\, n$	NP → *a*, N*
c_{every} $: N \multimap NP$	γ_{every} $: (\tau \multimap \tau) \multimap \tau \multimap \tau$ $= \lambda^{\mathrm{o}} n.NP_2\, every\, n$	NP → *every*, N*

Table 2.4: ACG lexicon for a richer TAG (following TAG syntactic analysis of (Gardent and Kallmeyer, 2003))

Terms of $\mathcal{A}(\mathscr{G}_{typed\ trees})$	Representation as TAG derived tree
$c_{sleeps}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, (c_{dog}\, c_a\, (c_{black}\, (c_{big}\, (c_{fierce}\, I_{\mathsf{N}}))))$ $:= \mathsf{S}_2\, (\mathsf{NP}_2\, a\, (\mathsf{N}_2\, fierce\, (\mathsf{N}_2\, big\, (\mathsf{N}_2\, black\, (\mathsf{N}_2\, dog)))))\, (\mathsf{VP}_1\, sleeps)$	 *a* — fierce — big — black — dog — sleeps (tree)
$c_{chases}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}(c_{dog}\, c_{every}\, I_{\mathsf{N}})\, (c_{cat}\, c_a\, I_{\mathsf{N}})$ $:= \mathsf{S}_2\, (\mathsf{NP}_2\, every\, (\mathsf{N}_1\, dog))\, (\mathsf{VP}_2\, chases\, (\mathsf{NP}_2\, a\, (\mathsf{N}_1\, cat)))$	*every dog chases a cat* (tree)
$c_{loves}\, (c_{claims}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, c_{Paul})\, I_{\mathsf{VP}}\, c_{John}\, c_{Mary}$ $:= \mathsf{S}_2\, (\mathsf{NP}_1\, Paul)\, (\mathsf{VP}_2\, claims\, (\mathsf{S}_2\, (\mathsf{NP}_1\, John)\, (\mathsf{VP}_2\, loves\, (\mathsf{NP}_1\, Mary))))$	*Paul claims John loves Mary* (tree)
$c_{to\ love}\, (c_{claims}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, c_{Paul})\, (c_{seems}\, I_{\mathsf{VP}})\, c_{John}\, c_{Mary}$ $:= \mathsf{S}_2\, (\mathsf{NP}_1\, Mary)$ $\quad (\mathsf{S}_2(\mathsf{NP}_1\, Paul)$ $\quad\quad (\mathsf{VP}_2\, claims\, (\mathsf{S}_2\, (\mathsf{NP}\, John)\, (\mathsf{VP}_2\, seems\, (\mathsf{VP}_1\, to\ love)))))$	*Mary Paul claims John seems to love* (tree)
$c_{liked}, (c_{said}\, (c_{does\ think}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, c_{Paul})\, I_{\mathsf{VP}}\, c_{John})\, I_{\mathsf{VP}}\, c_{who}\, c_{Bill}$ $:= \mathsf{S}_2\, (\mathsf{WH}_1\, who)$ $\quad (\mathsf{S}_2\, does\, (\mathsf{S}_2\, (\mathsf{NP}_1\, Paul)$ $\quad\quad (\mathsf{VP}_2\, think\, (\mathsf{S}_2\, (\mathsf{NP}_1\, John)\, (\mathsf{VP}_2\, said$ $\quad\quad\quad (\mathsf{S}_2\, (\mathsf{NP}_1\, Bill)(\mathsf{VP}_1\, liked)))))))$	*who does Paul think John said Bill liked* (tree)

Table 2.5: Terms of $\mathcal{A}(\mathscr{G}_{typed\ trees})$ with their interpretation in $\Lambda(\Sigma_{trees})$

$$
\begin{aligned}
c_{sleeps} &:=_{Log} \lambda^o Sas.S(s(a(\lambda^o x.\mathsf{sleep}\, x))) & c_{to\, love} &:=_{Log} \lambda^o Saso.S(s(a(\lambda^o x.o(\lambda^o y.\mathsf{love}\, x\, y)))) \\
c_{claims} &:=_{Log} \lambda^o Sasc.S(s(a(\lambda^o x.\mathsf{claim}\, x\, c))) & c_{seems} &:=_{Log} \lambda^o ar.a(\lambda^o x.\mathsf{seem}\, x\, r) \\
c_{liked} &:=_{Log} \lambda^o Saws.w(\lambda^o y.S(s(a(\lambda^o x.\mathsf{like}\, x\, y)))) & c_{does\, think} &:=_{Log} \lambda^o Sasc.S(s(a(\lambda^o x.\mathsf{think}\, x\, c))) \\
c_{who} &:=_{Log} \lambda^o P.\mathsf{WHO}\, P & c_{big} &:=_{Log} \lambda^o an.a(\lambda x.(\mathsf{big}\, x) \wedge (n\, x)) \\
c_{dog} &:=_{Log} \lambda^o da.d(a(\lambda x.\mathsf{dog}\, x)) & c_a &:=_{Log} \lambda^o PQ.\exists x.(P\, x) \wedge (Q\, x) \\
c_{every} &:=_{Log} \lambda^o PQ.\forall x.(P\, x) \Rightarrow (Q\, x)
\end{aligned}
$$

Table 2.6: ACG lexicon for semantic interpretation

$$
\begin{aligned}
c_{sleeps}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, (c_{dog}\, c_a\, (c_{black}\, (c_{big}\, (c_{fierce}\, I_{\mathsf{N}})))) \quad &:=_{Log}\ \exists x.((\mathsf{new}\, x) \wedge ((\mathsf{big}\, x) \wedge ((\mathsf{black}\, x) \\
&\qquad\qquad \wedge(\mathsf{dog}\, x)))) \wedge (\mathsf{sleep}\, x) \\
c_{chases}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}(c_{dog}\, c_{every}\, I_{\mathsf{N}})\, (c_{cat}\, c_a\, I_{\mathsf{N}}) \quad &:=_{Log}\ \forall x.(\mathsf{dog}\, x) \Rightarrow (\exists x'.(\mathsf{cat}\, x) \wedge (\mathsf{chase}\, x\, x')) \\
c_{loves}\, (c_{claims}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, c_{Paul})\, I_{\mathsf{VP}}\, c_{John}\, c_{Mary} \quad &:=_{Log}\ \mathsf{claim\, p\, (love\, j\, m)} \\
c_{to\, love}\, (c_{claims}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, c_{Paul})\, (c_{seems}\, I_{\mathsf{VP}})\, c_{John}\, c_{Mary} \quad &:=_{Log}\ \mathsf{claim\, p\, (seem\, j\, (\lambda^o x.love\, x\, m))} \\
c_{liked}(c_{said}\, (c_{does\, think}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, c_{Paul})\, I_{\mathsf{VP}}\, c_{John})\, I_{\mathsf{VP}}\, c_{who}\, c_{Bill} &:=_{Log}\ \mathsf{WHO}(\lambda^o y.\mathsf{think\, p\, (say\, j\, (like\, b\, y)))}
\end{aligned}
$$

Table 2.7: Terms of $\mathcal{A}(\mathscr{G}_{typed\ trees})$ and their semantic interpretation in $\lambda^o(\Sigma_{Log})$

These terms illustrate how using very standard semantic interpretation recipes, the abstract terms, hence the derivation trees of a TAG, are perfectly able to model:

1. Quantifier scoping: Indeed, while the derivation tree indicated no direct dependence between the determiner and the verb predicate (see Figure 2.1(b)), and even worse, the scoping is the other way around, using type raising (as in (Montague, 1974)) we reverse the functor-argument positions *inside the lexical semantic recipes*. While at the abstract level, the term for the noun phrase is a parameter of the term for the verb, the interpretation of the latter make the former scope over the verbal predicate (see for instance $\lambda^o s.s(\ldots(\lambda^o x.\mathsf{sleep}\, x)))$.

   Since the same thing occurs in the semantic recipe for nouns where the determiner is interpreted as taking scope over the nouns predicate (as in $\lambda^o d \ldots .d(\ldots(\lambda^o x.\mathsf{dog}\, x)))$, the result is as expected. This contrast for instance with the proposal in (Kallmeyer, 2002) to add link information to the derivation tree.

2. Wh-questions: here again, the scoping relation between the Wh-word and the (complex) sentence is rendered because while auxiliary trees ($\mathsf{S} \multimap \mathsf{S}$) appear in the abstract term as parameter, the interpretation of abstract terms for verbs make them appear as taking scope over the other argument ($\lambda^o S \ldots .S(\ldots)$).

3. Multiple adjunction: Finally, the interpretation makes precise how to possible auxiliary trees (for *seems* and for *claims*) should interact when occurring on a same abstract term (*to love* in the abstract term $c_{to\, love}\, (c_{claims}\, I_{\mathsf{S}}\, I_{\mathsf{VP}}\, c_{Paul})\, (c_{seems}\, I_{\mathsf{VP}})\, c_{John}\, c_{Mary}$: *Mary Paul claims John seems to love*). Indeed, in the interpretation of $c_{to\, love}$, the respective scope of the two parameters is completely described: $\lambda^o Sa \cdots . \cdots S(a(\cdots))$.

Remark 2.4. *This lexicon cannot model scope ambiguity and only give the subject wide scope reading. Various solutions can be found in (Pogodalla, 2004) (using underspecified representation languages) or in (Pogodalla, 2007a) (using categorial grammar related techniques as exposed in Chapter 3).*

Exercice 2.2. *Define an alternative value of $[\![c_{to\, love}]\!]_{Log}$ so that we get the object wide scope reading.*

The current architecture we have is then the one described in Figure 2.2, that relates derived trees with semantic representations. We can also use the composition (as in "function composition") properties of ACG to add the string level. This is the object of the next section.
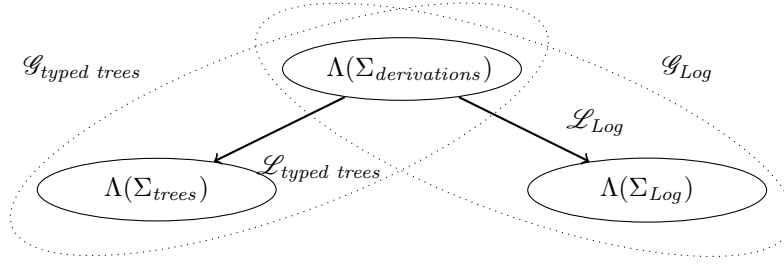
Figure 2.2: First ACG architecture for TAG

2.2.4 Computing the Yield

In order to compute the yield of a tree, wan can make these preliminary remarks:

- Every tree has to be turned into a string. So, in ACG terms, trees (τ) are interpreted into strings (σ).

- A non terminal (tree) symbol of arity n concatenate the n strings associated to its n daughters.

- A terminal (tree) symbol has to be turned into the same symbol as a string.

So we first need a signature Σ_{string}:

- whose unique atomic type is σ, the type for strings;

- whose constants contain:

  - a concatenation operator $+ : \sigma \multimap \sigma \multimap \sigma$;
  - the empty string $\epsilon : \sigma$;

- whose constants also contain a constant $w : \sigma$ for each w terminal symbol of the TAG.

Then we can define the ACG $\mathscr{G}_{yield} = \langle \Sigma_{trees}, \Sigma_{string}, \mathscr{L}_{yield}, \tau \rangle$ with:

- $\tau :=_{yield} \sigma$;

- for any terminal w of the TAG, $w :=_{yield} w$;

- for any non terminal X of arity n of the TAG, $X :=_{yield} \lambda^o x_1 \cdots x_n . x_1 + \cdots + x_n$.

Exercice 2.3. *Check that $\gamma_{likes} :=_{yield} \lambda^o aso.s + a(likes + o)$. Remember that γ_{likes} is not a constant of Σ_{trees}.*

We then get the architecture of Table 2.3. It's worth noting that we could perfectly compose $\mathscr{G}_{typed\ trees}$ and and have only one ACG (the result of the composition, whose lexicon would be $\mathscr{L}_{yield} \circ \mathscr{L}_{typed\ trees}$). But the modular approach can also be useful, as the next section shows.

2.2.5 Adding Some More Control to Get TAG

The construct we proposed until now makes the underlying assumption that whenever there is an adjunction, that is a parameter ($X \multimap X$) itself is the result of some "simple" combination of constants of $\Sigma_{derivations}$. Here, "simple" means "by way of application", what the TAG derivation tree representation illustrate. However, in the general case, it is possible that a term of type ($X \multimap X$) results not only form application of constants, but also as *abstraction* as in $\lambda^o x. c_1 (c_2 x) c_3$ for instance.
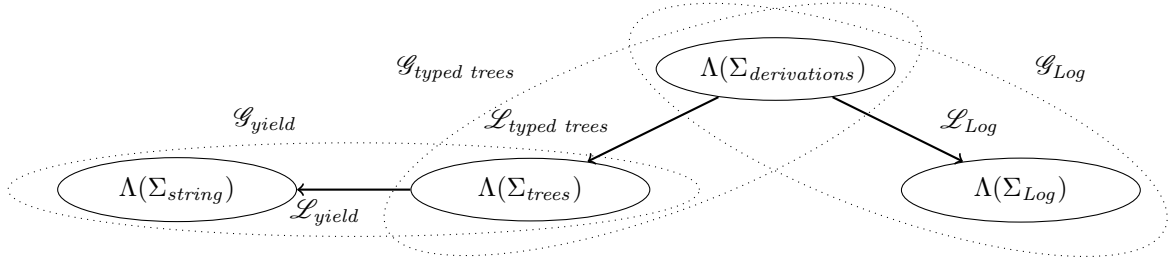
Figure 2.3: ACG architecture for TAG adding the yield

Take for instance the tree $c_{uncertain} : (\mathsf{VP} \multimap \mathsf{VP}) \multimap \mathsf{S} \multimap \mathsf{S}$ which can be interpreted as:

$$\begin{array}{c} \mathsf{S} \\ \diagup \quad \diagdown \\ \mathsf{S} \qquad \mathsf{VP} \\ \mid \\ uncertain \end{array}$$

(as in *whether he will actually survive the experience is uncertain*, see (XTAG Research Group, 2001, Section 6.23). In this case, the S parameter is meant to be substituted (and the copula is adjoined at the VP node). But nothing prevent us to build the term $\lambda^{\circ} x.c_{uncertain}\, c_{is}\, x$ and to use it *as an auxiliary tree* at some node S in some other derivation.

The only way to prevent terms with abstraction is to have a signature with at most 2nd order types for the constants: using a term with an abstraction would mean having a constant whose type is $(\alpha \multimap \beta) \multimap \gamma$, hence at least of 3rd order. So on top of $\Sigma_{derivations}$, we need a new higher-order signature Σ_{TAG} such that:

- it contains an atomic type X_A for any X non-terminal symbol that is the root (and the foot) node of an auxiliary tree (*i.e.*something that was modelled in $\Sigma_{derivations}$ by a term of type $X \multimap X$;

- it contains an atomic type X for any non-terminal symbol where a substitution can occur;

- it contains a constant for each elementary tree (as $\Sigma_{derivations}$ does);

- it contains constants I_X of type X_A for to represent fake adjunctions of identity.

So for instance, while we have $c_{sleeps} : (\mathsf{S} \multimap \mathsf{S}) \multimap (\mathsf{VP} \multimap \mathsf{VP}) \multimap \mathsf{S}$ in $\Sigma_{derivations}$, we have $C_{sleeps} : \mathsf{S}_A \multimap \mathsf{VP}_A \multimap \mathsf{NP} \multimap \mathsf{S}$ in Σ_{TAG}.

And with $C_{uncertain} : \mathsf{VP}_A \multimap \mathsf{S} \multimap \mathsf{S}$, we don't have any "equivalence" between the type $\mathsf{S} \multimap \mathsf{S}$ of $\lambda^{\circ} x.C_{uncertain}\, C_{is}x$ with the type S_A. Hence this term cannot be used as auxiliary tree.

Thanks to the modular architecture of ACG, this can be expressed without changing anything to the previous construction, but only by adding a new ACG $\mathscr{G}_{TAG} = \langle \Sigma_{TAG}, \Sigma_{derivations}, \mathscr{L}_{TAG}, \mathsf{S} \rangle$ where $\mathscr{L}_{TAG}$ is such that:

- for any type X_A of Σ_{TAG}, $X_A :=_{TAG} X \multimap X$;

- for any other type X of Σ_{TAG}, $X :=_{TAG} X$;

- for any constant C_w of Σ_{TAG}, $C_w :=_{TAG} c_w$;

- for any constant $I_X : X_A$ of Σ_{TAG}, $I_X :=_{TAG} I_X$.

Because of the homomorphism between $\mathcal{A}(_{TAG})$ and $\mathcal{O}(_{TAG})$, which is the subset of $\Lambda(\Sigma_{derivations})$ that can be represented as trees (that is the TAG derivation trees, as Section 2.2.2 shows), $\mathcal{A}(_{TAG})$ is also a representation of the usual TAG derivation trees.

Figure 2.4 sums up the overall architecture for modelling TAG with ACG. It goes from the derivation trees to the yield, and, interestingly, branches at an intermediate level to build the semantic representation. This presentation is a modularized version of (de Groote, 2002; Pogodalla, 2004).
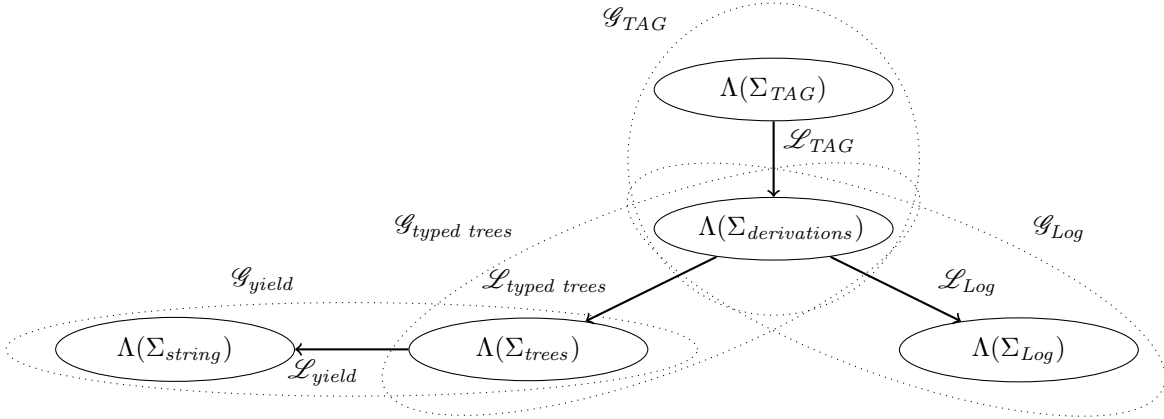
Figure 2.4: Complete ACG architecture for modelling TAG

If we consider $\mathscr{G} = \mathscr{G}_{yield} \circ \mathscr{G}_{typed\ trees} \circ \mathscr{G}_{TAG}$, the ACG resulting from the composition of the three ACG, the order of $\mathscr{G}$ is 2 (its order, the maximum order of the types of its abstract constants, is the same as the one of $\mathscr{G}_{TAG}$ since they have the same abstract constants). The order of the lexicon of $\mathscr{G}$ is 3 (it is the maximum order of the types of the realizations of its abstract types).

Exercice 2.4. *Let $\ell : \mathbb{N} \to \mathbb{N}$ be the following function: if n has the form 2^k then $\ell(n) = k$. Otherwise, $\ell(n) = 1$. Let $L = \{(a^n, \ell(n)) | n \in \mathbb{N}\}$. Can you find $\mathscr{G}_{TAG}$, $\mathscr{G}_{typed\ trees}$, $\mathscr{G}_{yield}$ and $\mathscr{G}_{Log}$ such that:*

- $\mathcal{O}(\mathscr{G}_{yield} \circ \mathscr{G}_{typed\ trees} \circ \mathscr{G}_{TAG}) = \{a^n | n \in \mathbb{N}\}$;

- *for all $n \in \mathbb{N}$ and for all $t \in \mathcal{A}(\mathscr{G}_{TAG})$ such that $\mathscr{L}_{yield} \circ \mathscr{L}_{typed\ trees} \circ \mathscr{L}_{TAG}(t) = a^n$, $\mathscr{L}_{Log} \circ \mathscr{L}_{typed\ trees} \circ \mathscr{L}_{TAG}(t) = \ell(n)$.*

2.2.6 Relaxing the Control to Get MCTAG

In Multicomponent TAGs (Joshi, 1988; Joshi, Levy, and Takahashi, 1975; Weir, 1988), instead of having only one adjunction in a single tree at a time, it's possible to specify a sequence of auxiliary trees that can adjoin simultaneously at different addresses of a sequence of trees. Depending on additional constraints, the complexity of parsing varies:

Tree-local MCTAG: the adjunctions of a set of auxiliary trees have to take place in a single tree. Then the same languages can be generated (strings and derived trees);

Set-local MCTAG a sequence of trees can be adjoined into distinct nodes of any member of a single elementary tree sequence. Then the generative capacity increases and reach m-LCFRS Weir, 1988;

Non-local MCTAG the two previous constraints are dropped. Then language for which the world recognition problem is NP-complete can be generated Champollion, 2007; Rambow, 1994; Rambow and Satta, 1992.

Let's assume the following terms: (Schuler, Chiang, and Dras, 2000) derives *Does John seem likely to sleep* using a pair $m_{does\ seem}$ "made of" the auxiliary trees C_{does} and C_{seem}, and a pair m_{likely} "made of" the auxiliary trees C_S and C_{likely}:

- C_{does} adjoins on C_S while C_{seem} adjoins to C_{likely};

- the resulting pair adjoin on a same tree for the verb *to sleep* at the root node S and at the VP node.

Constants of the signatures (Σ_{TAG}, $\Sigma_{derivations}$ and Σ_{trees} resp.)	The corresponding TAG tree
C_{does} : S_A c_{does} : $\mathsf{S} \multimap \mathsf{S}$ γ_{does} : $\tau \multimap \tau$ γ_{does} $\lambda x.\mathsf{S}_2 \, does \, x$	S $/ \quad \backslash$ $does \qquad \mathsf{S}^*$
C_{S} : $\mathsf{S}_A \multimap \mathsf{S}_A$ c_{S} : $(\mathsf{S} \multimap \mathsf{S}) \multimap \mathsf{S} \multimap \mathsf{S}$ γ_{S} : $(\tau \multimap \tau) \multimap \tau \multimap \tau$ γ_{S} $= \lambda ax.a\,x$	S^*

It is possible to model pairs (n-uples resp.) as a higher-order term whose parameter is a function taking two arguments (n arguments resp.). So for instance, $m_{does\ seem}$ would be modelled by $\lambda f.f\, C_{does}\, C_{seem}$ and m_{likely} would be modelled by $\lambda f.f\, C_{\mathsf{S}}\, C_{likely}$. These terms belong to a new higher-order signature Σ_{tuple}.

While adjunction was modeled until by having the initial tree taking as parameter an auxiliary tree, we now have to model the initial tree as taking pairs (or n-uples) of auxiliary trees. So we need constants for initial trees as $m_{to\ sleep}$ to be of type:

$$((\mathsf{S}_A \multimap \mathsf{VP}_A \multimap \mathsf{S}) \multimap \mathsf{S}) \multimap \mathsf{NP} \multimap \mathsf{S}$$

The first parameter $(\mathsf{S}_A \multimap \mathsf{VP}_A \multimap \mathsf{S}) \multimap \mathsf{S}$ is the pair that can be adjoined at note S and at node VP. $m_{to\ love}$ would be interpreted as:

$$m_{to\ love} \quad := \lambdaº Pso.P(\lambdaº xy.C_{to\ love}\, x\, y\, s\, o)$$

The pair P plays the role of the pair, hence take a function with two argument as parameters. These function precisely describes where the two components x and y of the pair should occur in the whole tree. Note that it is described using the former definition on $C_{to\ love}$: we simply specify the place where each of the components of the pair should contribute. The attentive reader may already guess at this point that a new ACG is showing up...

In the same spirit we would have:

$$m_{likely} \quad : ((\mathsf{S}_A \multimap \mathsf{VP}_A \multimap \mathsf{S}) \multimap \mathsf{S}) \multimap (\mathsf{S}_A \multimap \mathsf{VP}_A \multimap \mathsf{S}) \multimap \mathsf{S}$$
$$m_{likely} \quad := \lambdaº Pf.P(\lambdaº xy.f(\lambdaº s.C_{\mathsf{S}}(x\,s))(\lambdaº v.C_{seem}(y\,v)))$$

Hence

$$m_{likely}\, m_{does\ seem} \qquad := \lambdaº f.(\lambdaº f.f\, C_{does}\, C_{seem})(\lambdaº xy.f(\lambdaº s.C_{\mathsf{S}}(x\,s))(\lambdaº v.C_{likely}(y\,v)))$$
$$\rightarrow_\beta \lambdaº f.f(\lambdaº s.C_{\mathsf{S}}(C_{does}\,s))(\lambdaº v.C_{likely}(C_{seem}\,v))$$
$$m_{to\ love}\,(m_{likely}\, m_{does\ seem}) \quad := \lambdaº s_0.C_{to\ love}(\lambdaº s.C_{\mathsf{S}}(C_{does}\,s))(\lambdaº v.C_{likely}(C_{seem}\,v))\,s_0 : \mathsf{NP} \multimap \mathsf{S}$$

The latter term only lacks its subject. Then, the interpretation into trees and strings is just the same as for any terms of $\Lambda(\Sigma_{TAG})$. So, actually, we can define a new ACG $\mathscr{G}_{tuple} = \langle \Sigma_{tuple}, \Sigma_{TAG}, \mathscr{L}_{tuple}, \mathsf{S} \rangle$ that allows us to define and use tuples of auxiliary trees.

However, just as $\mathscr{G}_{typed\ trees}$ is not enough and would allow us to have "fake" auxiliary trees by using abstraction, $\mathscr{G}_{tuple}$ is not enough to model set-local MCTAG: the same mechanism of abstraction could be used to model "fake" auxiliary tree sequence, hence would allow us to have(at least some of) the same effects as non-local MCTAG. The solution here again is to add an ACG $\mathscr{G}_{MCTAG}$ whose abstract signature provide an atomic type for each kind of tuple, constants with this type that are realized in $\Lambda(\Sigma_{tuple})$ by the kind of terms we just defined and *whose type are 2nd order*.

For instance, we would have:

$$M_{to\ love} \qquad : [\mathsf{S}_A, \mathsf{VP}_A] \multimap \mathsf{NP} \multimap \mathsf{S}$$
$$M_{does\ seem} \quad : [\mathsf{S}_A, \mathsf{VP}_A]$$
$$M_{likely} \qquad : [\mathsf{S}_A, \mathsf{VP}_A] \multimap [\mathsf{S}_A, \mathsf{VP}_A]$$

where $[\mathsf{S}_A, \mathsf{VP}_A]$ is an atomic type whose name should help to memorize the kind of tuple it models. And for the lexicon of $\mathscr{G}_{MCTAG}$ we would have:

$$
\begin{aligned}
[\mathsf{S}_A, \mathsf{VP}_A] &:= (\mathsf{S}_A \multimap \mathsf{VP}_A \multimap \mathsf{S}) \multimap \mathsf{S} \\
\mathsf{NP} &:= \mathsf{NP} \\
\mathsf{S} &:= \mathsf{S} \\
M_{to\ love} &:= m_{to\ love} \\
M_{does\ seem} &:= m_{does\ seem} \\
M_{likely} &:= m_{likely}
\end{aligned}
$$

Figure 2.5 illustrates the final architecture we propose.

Remark 2.5. *Because the abstract constants of $\mathscr{G}_{MCTAG}$ are at most of order 2, and so is the ACG, we know that the object language of the ACG $\mathscr{G}_{yield} \circ \mathscr{G}_{typed\ trees} \circ \mathscr{G}_{TAG} \circ \mathscr{G}_{tuple} \circ \mathscr{G}_{MCTAG}$ is in m-LCFRS. Note however that the order of the lexicon is 5, hence is not "minimal" (for any ACG of order 2 and whose lexicon is of order $n \geq 4$, the language it generates can be generated by an ACG of order 2 whose lexicon is of order 4).*

Remark 2.6. *The multiplicity of possible tuples acting on a same other tuple implies multiple abstract constants. For instance, if there exist possible tuples of length 1 and of type $S \multimap S$ and $VP \multimap VP$, we need an additional entry*

$$M'_{to\ love} : [S_A] \multimap [VP_A] \multimap NP \multimap S$$

This corresponds to (Weir, 1988, p.102)'s proof of inclusion of the language generated by MCTAG into the ones generated by LCFRS: "Each composition function mentioned in the CFG encoding the MCTAG derivations corresponds to one of the ways that n derives tree sequences can be multi-adjoined into an elementary tree sequence".
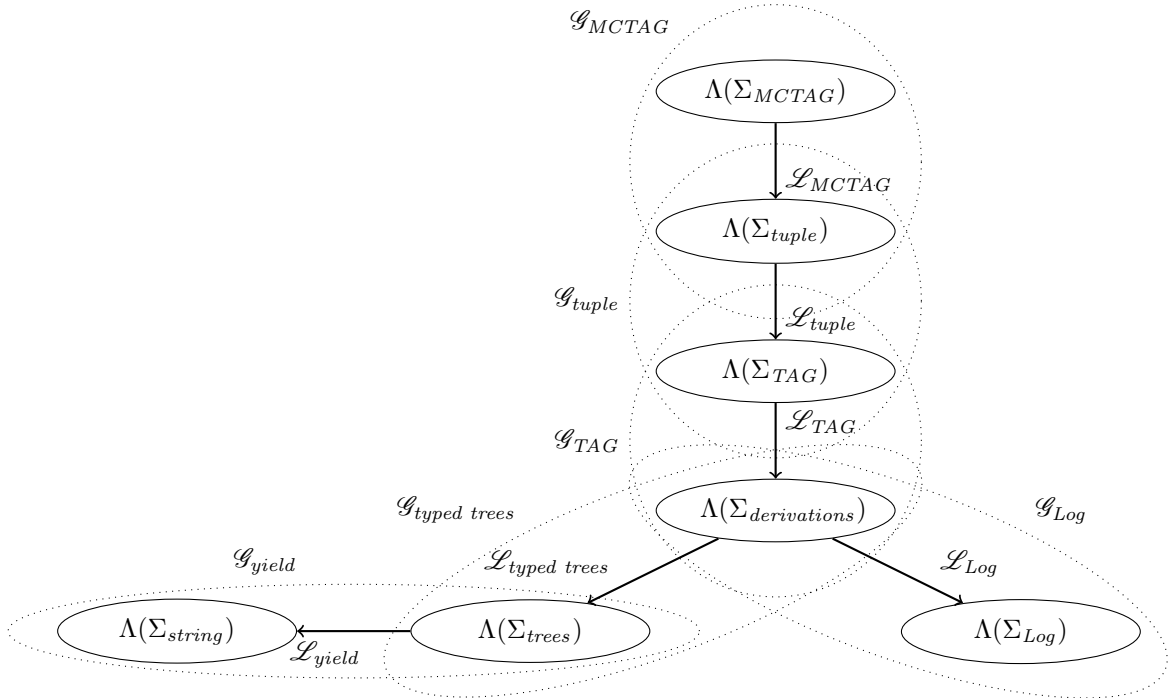

Figure 2.5: Complete ACG architecture for modelling TAG and MCTAG

2.3   Scope Ambiguity

This section is aimed to show how we can model scope ambiguity within the TAG formalism by relating it to the way it's done in Chapters 3 and 4 (a french version is available Pogodalla, 2007a).

Chapter 3

Syntax-Semantics Architecture: Comparing CG and ACG

This chapter is an extended version of (Pogodalla, 2007a). It aims at showing:

1. How ACG relate to more classical Categorial Grammar formalisms.

2. How the compositional property of ACG can highlight the difference of viewpoints on syntax in various formalisms.

3.1 Semantic Ambiguity in Natural Language

When trying to build the semantic representation of a natural language expression, it may happen that a single expression produces many semantic representations. This *ambiguity* may arise from different sources: lexical ambiguity, as in (1a) for the word *bank*; structural ambiguity as in (1b) which does not by itself indicate whether the man used a telescope to see the woman, or if the woman the man saw had a telescope; scope ambiguity as in (1c) that may express that there is a woman that every man loves, but that for every man, there is a woman he loves as well.

(1) a. I visited the bank

   b. The man saw a woman with a telescope

   c. Every man loves a woman

In this chapter, we focus on scope ambiguities where a single syntactic analysis can yield many semantic representations. This kind of scope ambiguity can occur with quantified noun phrases (*every*, *some*, *most*, etc.) but also with adverbs and *how-many* questions, conjunctions, etc.

Nevertheless, and this is a main concern for some of the approaches that deal with scope ambiguities, and for our present proposal as well, this distinction between semantic ambiguity and structural ambiguity may be quite difficult to express in some formalisms. Basically, there are two ways to address scope ambiguities. One way is to build two syntactic structures (parse trees) from a single expression, then, from these syntactic structures, to build two semantic representations in a functional way. The other way is to build a single syntactic structure from the expression e and to associate to the latter, in a non-functional way, two semantic representations. These two ways are represented by two frameworks: the type-logical framework, where ambiguity is modeled by the process (proof search), and the underspecification framework, where ambiguity is modeled in a (formal) language.

Our approach aims at providing a proof-theoretic approach (based on proof search) without creating spurious syntactic ambiguities. The next section gives a description of the type-logical approach and of the underspecification approach. Then section 3.3 reminds what Abstract Categorial Grammars are and motivate our proposal in that framework. Section 3.4 gives and exemplifies our proposal and section 3.5 makes some comparison between our approach and other approaches.

3.2   Type-logical and Underspecification Frameworks

Ambiguity in Categorial and Type-Logical Grammars.   From a computational linguistics point of view, the most influential approach to quantifier scoping phenomena is Montague, 1974. Recent proposals in that framework are Carpenter, 1997; Moortgat, 1991. We don't give here the full details of theses approaches and delay this to the discussion and comparison with our approach at section 3.5.1. But the main ideas are twofold:

- the parse structure is a proof (in sequent calculus, or in natural deduction);

- the semantic representation is functionally build from the proof via the Curry-Howard isomorphism.

This explain why semantic ambiguity also requires some kind of syntactic ambiguity: since there is just one way to get the semantic interpretation from the syntactic structure, to get two (or more) semantic representations require two (or more) of these syntactic structures. This is achieved with types that are higher-order and enable hypothetical reasoning. Section 3.5.1 gives more formal details.

Despite its success in the modeling of a wide range of scope ambiguity phenomena (see for instance Carpenter, 1997 or Morrill, 1994), this way of modeling ambiguity is a major drawback for people considering that these ambiguities should occur only at the semantic level and not at the syntactic one. This partly motivated the storage techniques of Cooper, 1983. More recently, it has been addressed by a wide range of work around underspecification.

Ambiguity with Underspecification Languages.   The underlying idea for underspecification formalisms (see for instance Bos, 1995; Copestake et al., 2005; Egg, Koller, and Niehren, 2001; Muskens, 1995) is to add a level between syntactic structures and semantic representation: the underspecified representation. Basically, it is a tree description Rogers and Vijay-Shanker, 1992(or a set of constraints, or a specification) of the syntactic tree of the logical formulas that are the possible semantic representations. So going to the syntactic structure to the semantic structure becomes a two stage process: a first one that maps the syntactic structure to a (unique) underspecified formula, and a second stage that builds from the latter formula the possible semantic representations.

Drawbacks:

- another intermediate language

- not the one for generation

Alternatives.   Our proposal aims at giving an account of scope ambiguity that does not rely on syntactic ambiguity nor on another intermediate language. Our model is based on the Abstract Categorial Grammars (ACG) framework de Groote, 2001. This framework provides both a way to encode different kind of syntactic formalisms (CFG, TAG, m-LCFRS de Groote, 2002; de Groote and Pogodalla, 2004) and way to specify the syntax-semantics interface Pogodalla, 2004. We are then able to provide a general way of dealing with scope ambiguities in different (syntactical) formalisms, particularly with type-logical formalisms. Moreover, this framework is suitable to model discourse de Groote, 2006.

3.3 Abstract Categorial Grammars

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. This general picture can of course be adapted to the need of the modeling. In order to be able to model non linearity (this is useful for semantics), we use an extension of the ACG with both *linear* and *non-linear* implication but the principles follow de Groote, 2001's definitions.[1]

Definition 3.1. *Let A be a set of atomic types. The set $\mathscr{T}(A)$ of implicative types build upon A is defined with the following grammar:*

$$\mathscr{T}(A) ::= A \mid \mathscr{T}(A) \multimap \mathscr{T}(A) \mid \mathscr{T}(A) \to \mathscr{T}(A)$$

Definition 3.2. *A higher-order signature Σ is a triple $\Sigma = \langle A, C, \tau \rangle$ where:*

- *A is a finite set of atomic types;*
- *C is a finite set of constants;*
- *$\tau : C \to \mathscr{T}(A)$ is a function assigning a types to constants.*

Definition 3.3. *Let X be an infinite countable set of λ-variables. The set $\Lambda(\Sigma)$ of λ-terms built upon a higher-order signature $\Sigma = \langle A, C, \tau \rangle$ is inductively defined as follows:*

- *if $c \in C$ then $c \in \Lambda(\Sigma)$;*
- *if $x \in X$ then $x \in \Lambda(\Sigma)$;*
- *if $x \in X$ and $t \in \Lambda(\Sigma)$ and x occurs free in t exactly once, then $\lambda^o x.t \in \Lambda(\Sigma)$;*
- *if $x \in X$ and $t \in \Lambda(\Sigma)$ and x occurs free in t, then $\lambda x.t \in \Lambda(\Sigma)$;*
- *if $t, u \in \Lambda(\Sigma)$ and the set of free variables of u and t are disjoint then $(t\,u) \in \Lambda(\Sigma)$.*

Note there is a linear λ-abstraction and a (usual) intuitionistic λ-abstraction. There also are the usual notion of α-conversion and β-reduction.

Given a higher-order signature Σ, the typing rules are given with an inference system whose judgments are of the following form: $\Gamma; \Delta \vdash_\Sigma t : \alpha$ where:

- Γ is a finite set of non-linear variable typing declaration;
- Δ is a finite set of linear variable typing declaration.

Both Γ and Δ may be empty. Here are the typing rules:

$$\frac{}{\Gamma; \vdash_\Sigma c : \tau(c)} \text{ (const.)}$$

$$\frac{}{\Gamma; x : \alpha \vdash_\Sigma x : \alpha} \text{ (lin. var.)} \qquad \frac{}{\Gamma, x : \alpha; \vdash_\Sigma x : \alpha} \text{ (var.)}$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda^o x.t : \alpha \multimap \beta} \text{ (lin. abs.)}$$

$$\frac{\Gamma, x : \alpha; \Delta \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda x.t : \alpha \to \beta} \text{ (abs.)}$$

---

[1]Formal properties of this extension, as expressiveness and the computational properties, are beyond the scope of this chapter.

$$\frac{\Gamma; \Delta_1 \vdash_\Sigma t : \alpha \multimap \beta \qquad \Gamma; \Delta_2 \vdash_\Sigma u : \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash_\Sigma (t\, u) : \beta} \text{ (lin. app.)}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma t : \alpha \to \beta \qquad \Gamma; \vdash_\Sigma u : \alpha}{\Gamma; \Delta \vdash_\Sigma (t\, u) : \beta} \text{ (app.)}$$

Definition 3.4. *Let $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ be two higher-order signatures, a lexicon $\mathscr{L} = \langle F, G \rangle$ from Σ_1 to Σ_2 is such that:*

- *$F : A_1 \to \mathscr{T}(A_2)$. We also note $F : \mathscr{T}(A_1) \to \mathscr{T}(A_2)$ its homomorphic extension[2];*

- *$G : C_1 \to \Lambda(\Sigma_2)$. We also note $G : \Lambda(\Sigma_1) \to \Lambda(\Sigma_2)$ its homomorphic extension;*

- *F and G are such that for all $c \in C_1$, $\vdash_{\Sigma_2} G(c) : F(\tau_1(c))$ is provable.*

We also use $\mathscr{L}$ instead of F or G.

Definition 3.5. *An* abstract categorial grammar *is a quadruple $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, S \rangle$ where:*

- *$\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ are two higher-order signatures;*

- *$\mathscr{L} : \Sigma_1 \to \Sigma_2$ is a lexicon;*

- *$S \in \mathscr{T}(A_1)$ is the* distinguished type *of the grammar.*

Definition 3.6. *Given an ACG $\mathscr{G}$, the* abstract language *is defined by*

$$\mathcal{A}(\mathscr{G}) = \{t \in \Lambda(\Sigma_1) \,|\, \vdash_{\Sigma_1} t : S \text{ is derivable}\}$$

The object language *is defined by*

$$\mathcal{O}(\mathscr{G}) = \{u \in \Lambda(\Sigma_2) \,|\, \exists t \in \mathcal{A}(\mathscr{G}) \text{ s.t. } u = \mathscr{L}(t)\}$$

Note that $\mathscr{L}$ binds the parse structures of $\mathcal{A}(\mathscr{G})$ to the concrete expressions of $\mathcal{O}(\mathscr{G})$. Depending on the choice of Σ_1, Σ_2 and $\mathscr{L}$, it can map for instance derivation trees and derived trees for TAG 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. Moreover, this link between an abstract and a concrete structure can apply not only to syntactical formalisms, but also to semantic formalisms Pogodalla, 2004.

A crucial point is that ACG can be mixed in different ways: in a transversal way, were two ACG 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 (figure 3.1 illustrates these composition between three ACG). This chapter exemplifies both of this usage. It is also at the heart of our proposal.
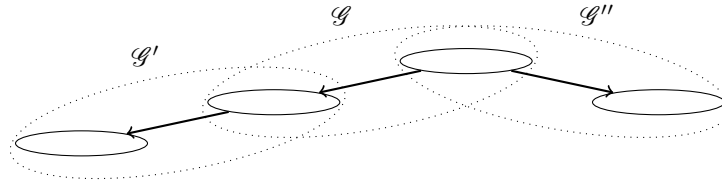

Figure 3.1: Ways of combining ACG

---

[2]such that $F(\alpha \multimap \beta) = F(\alpha) \multimap F(\beta)$ and $F(\alpha \to \beta) = F(\alpha) \to F(\beta)$

Now, let us run into an example. Here is how scope ambiguities would be modeled in the usual way by categorial grammars. We first define the syntactic part.

$$\Sigma_{\text{synt}} = \begin{cases} A_{\text{synt}} & = \{\mathsf{NP}, \mathsf{N}, \mathsf{S}\} \\ C_{\text{every}} & : \mathsf{N} \multimap ((\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}) \qquad\quad C_{\text{man}} \quad : \mathsf{N} \\ C_{\text{some}} & : \mathsf{N} \multimap ((\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}) \qquad\quad C_{\text{woman}} \quad : \mathsf{N} \\ C_{\text{love}} & : \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S} \end{cases}$$

Then the language of strings[3].

$$\Sigma_{\text{string}} = \begin{cases} A_{\text{string}} & = \{\textsc{String}\} \\ every & : \textsc{String} \qquad man \quad : \textsc{String} \quad some \quad : \textsc{String} \quad + \quad : \textsc{String} \multimap \textsc{String} \\ loves & : \textsc{String} \qquad woman : \textsc{String} \quad \epsilon \qquad\;\, : \textsc{String} \end{cases}$$

And finally the lexicon.

$$\mathscr{L}^0_{syntax} = \begin{cases} \mathsf{N} := \textsc{String} \quad \mathsf{NP} := \textsc{String} \quad \mathsf{S} := \textsc{String} \\ C_{\text{every}} \quad := \lambda^{\!o} x R.R(every + x) \qquad\qquad C_{\text{man}} \qquad := man \\ C_{\text{some}} \quad := \lambda^{\!o} x R.R(some + x) \qquad\qquad C_{\text{woman}} \quad := woman \\ C_{\text{love}} \quad := \lambda^{\!o} xy.x + loves + y \end{cases}$$

Let $\mathscr{G}_{\text{syntax}} = \langle \Sigma_{\text{synt}}, \Sigma_{\text{string}}, \mathscr{L}^0_{syntax}, \mathsf{S} \rangle$ be an ACG. Does the sentence(1c) belong to $\mathcal{O}(\mathscr{G}_{\text{syntax}})$? It amounts to find $t \in \mathcal{A}(\mathscr{G}_{\text{syntax}})$ such that $\mathscr{L}^0_{syntax}(t) = every + man + loves + some + woman$[4]. There are two such terms: $t_1 = (C_{\text{every}} C_{\text{man}})(\lambda x.(C_{\text{some}} C_{\text{woman}})(\lambda y.C_{\text{love}}\, x\, y))$ and $t_2 = (C_{\text{some}} C_{\text{woman}})(\lambda y.(C_{\text{every}} C_{\text{man}})(\lambda x.C_{\text{love}}\, x\, y))$. Indeed we have:

$$\begin{aligned} \mathscr{L}^0_{syntax}(t_1) &= (\lambda^{\!o} R.R(every + man)) \\ &\qquad (\lambda^{\!o} x.(\lambda^{\!o} Q.Q(some + woman))(\lambda^{\!o} y.x + loves + y)) \\ &= (\lambda^{\!o} R.R(every + man)) \\ &\qquad (\lambda^{\!o} x.(\lambda^{\!o} y.x + loves + y)(some + woman)) \\ &= (\lambda^{\!o} R.R(every + man))(\lambda^{\!o} x.x + loves + some + woman) \\ &= (\lambda^{\!o} x.x + loves + some + woman)(every + man) \\ &= every + man + loves + some + woman \end{aligned}$$

$$\begin{aligned} \mathscr{L}^0_{syntax}(t_2) &= (\lambda^{\!o} R.R(some + woman)) \\ &\qquad (\lambda^{\!o} y.(\lambda^{\!o} Q.Q(every + man))(\lambda^{\!o} x.x + loves + y)) \\ &= (\lambda^{\!o} R.R(some + woman)) \\ &\qquad (\lambda^{\!o} y.(\lambda^{\!o} x.x + loves + y)(every + man)) \\ &= (\lambda^{\!o} R.R(some + woman))(\lambda^{\!o} y.every + man + loves + y) \\ &= (\lambda^{\!o} y.every + man + loves + y)(some + woman)) \\ &= every + man + loves + some + woman \end{aligned}$$

But it is interesting to look at the corresponding proofs of $\vdash_{\Sigma_{\text{synt}}} t_1 : \mathsf{S}$ and $\vdash_{\Sigma_{\text{synt}}} t_2 : \mathsf{S}$[5]. Let us define Π_1, Π_2 and Π_3 the following proofs:

---

[3]ϵ represents the empty string.

[4]In the very general case, this problem, known as *parsing ACG* is not decidable. However, some restrictions (as linearity, but other ones too that are not linear) make it decidable (and polynomial sometime). For such discussions, see Salvati, 2005, 2006; Yoshinaka, 2006

[5]We omit the non-linear part, hence the ';', of the sequent because it is always empty in that example.

$$\Pi_1 = \begin{cases} \dfrac{\vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{every}} : \mathsf{N} \multimap (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S} \qquad \vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{man}} : \mathsf{N}}{\vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{every}} C_{\mathrm{man}} : (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}} \end{cases}$$

$$\Pi_2 = \begin{cases} \dfrac{\vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{some}} : \mathsf{N} \multimap (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S} \qquad \vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{woman}} : \mathsf{N}}{\vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{some}} C_{\mathrm{woman}} : (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}} \end{cases}$$

$$\Pi_3 = \begin{cases} \dfrac{\dfrac{x : \mathsf{NP} \vdash_{\Sigma_{\mathrm{synt}}} x : \mathsf{NP} \qquad \vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{love}} : \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S}}{x : \mathsf{NP} \vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{love}} x : \mathsf{NP} \multimap \mathsf{S}} \qquad y : \mathsf{NP} \vdash_{\Sigma_{\mathrm{synt}}} y : \mathsf{NP}}{x : \mathsf{NP}, y : \mathsf{NP} \vdash_{\Sigma_{\mathrm{synt}}} C_{\mathrm{love}} x \, y : \mathsf{S}} \end{cases}$$

Then, the proof for t_1 is the following:

$$\dfrac{\Pi_1 \qquad \dfrac{\Pi_2 \qquad \dfrac{\Pi_3}{x : \mathsf{NP} \vdash_{\Sigma_{\mathrm{synt}}} \lambda^o y.C_{\mathrm{love}} x \, y : \mathsf{NP} \multimap \mathsf{S}}}{\dfrac{x : \mathsf{NP} \vdash_{\Sigma_{\mathrm{synt}}} (C_{\mathrm{some}} C_{\mathrm{woman}})(\lambda^o y.C_{\mathrm{love}} x \, y) : \mathsf{S}}{\vdash_{\Sigma_{\mathrm{synt}}} \lambda^o x.(C_{\mathrm{some}} C_{\mathrm{woman}})(\lambda^o y.C_{\mathrm{love}} x \, y) : \mathsf{NP} \multimap \mathsf{S}}}}{\vdash_{\Sigma_{\mathrm{synt}}} (C_{\mathrm{every}} C_{\mathrm{man}})(\lambda^o x.(C_{\mathrm{some}} C_{\mathrm{woman}})(\lambda^o y.C_{\mathrm{love}} x \, y)) : \mathsf{S}}$$

and the proof for t_2 is the following:

$$\dfrac{\Pi_2 \qquad \dfrac{\Pi_1 \qquad \dfrac{\Pi_3}{y : \mathsf{NP} \vdash_{\Sigma_{\mathrm{synt}}} \lambda^o x.C_{\mathrm{love}} x \, y : \mathsf{NP} \multimap \mathsf{S}}}{\dfrac{y : \mathsf{NP} \vdash_{\Sigma_{\mathrm{synt}}} (C_{\mathrm{every}} C_{\mathrm{man}})(\lambda^o x.C_{\mathrm{love}} x \, y) : \mathsf{S}}{\vdash_{\Sigma_{\mathrm{synt}}} \lambda^o y.(C_{\mathrm{every}} C_{\mathrm{man}})(\lambda^o.C_{\mathrm{love}} x \, y) : \mathsf{NP} \multimap \mathsf{S}}}}{\vdash_{\Sigma_{\mathrm{synt}}} (C_{\mathrm{some}} C_{\mathrm{woman}})(\lambda^o y.(C_{\mathrm{every}} C_{\mathrm{man}})(\lambda^o x.C_{\mathrm{love}} x \, y)) : \mathsf{S}}$$

It is clear that we essentially get there the usual syntactic ambiguity of the type-logical approaches (be it in the Lambek calculus style or with the $\Uparrow$ binder of Moortgat, 1991, see Morrill, 1994, p. 153 or Carpenter, 1997, p. 224) that depends on the order in which abstractions occur.

Now, we can look at the semantic part. It will rely on the same abstract language. However, we need a higher-order signature for the semantic representation. So we first define Σ_{sem} as

$$\Sigma_{\mathrm{sem}} = \begin{cases} A_{\mathrm{sem}} & = \{e, t\} \\ \forall & : (e \to t) \multimap t \quad \exists & : (e \to t) \multimap t \\ \Rightarrow & : t \multimap t \multimap t \quad \wedge & : t \multimap t \multimap t \\ \mathbf{man} & : e \multimap t \quad \mathbf{woman} & : e \multimap t \\ \mathbf{love} & : e \multimap e \multimap t \quad \neg & : t \multimap t \end{cases}$$

Then the lexicon[6]:

$$\mathscr{L}_{sem} = \begin{cases} \mathsf{N} := e \multimap t \quad \mathsf{NP} := e \quad \mathsf{S} := t \\ C_{\mathrm{every}} & := \lambda^o PQ.\forall x.(P\,x \Rightarrow Q\,x) \qquad C_{\mathrm{man}} & := \mathbf{man} \\ C_{\mathrm{some}} & := \lambda^o PQ.\exists x(P\,x \wedge Q\,x) \qquad C_{\mathrm{woman}} & := \mathbf{woman} \\ C_{\mathrm{love}} & := \mathbf{love} \end{cases}$$

We can now define the ACG $\mathscr{G}_{\mathrm{sem}} = \langle \Sigma_{\mathrm{synt}}, \Sigma_{\mathrm{sem}}, \mathscr{L}_{sem}, \mathsf{S} \rangle$. And we have the following two readings:

$$\mathscr{L}_{sem}(t_1) = (\lambda^o Q.\forall x.\mathbf{man}\, x \Rightarrow Q x)$$
$$(\lambda x.(\lambda^o Q.\exists y(\mathbf{woman}\, y \wedge Q\, y))(\lambda y.\mathbf{love}\, x \, y))$$
$$= \forall x.\mathbf{man}\, x \Rightarrow \exists y(\mathbf{woman}\, y \wedge \mathbf{love}\, x \, y)$$

---

[6] We use the usual notation $\forall x.P$ instead of $\forall(\lambda x.P)$.

$$\mathscr{L}_{sem}(t_2) = (\lambda^o Q.\exists y(\textbf{woman}\, x \wedge Qx))$$
$$(\lambda y.(\lambda^o Q.\forall x.\textbf{man}\, y \Rightarrow Q\, y)(\lambda x.\textbf{love}\, x\, y))$$
$$= \exists y(\textbf{woman}\, y \wedge \forall x.\textbf{man}\, x \Rightarrow \textbf{love}\, x\, y)$$

This example shows four things:

- how ACG transfer structures by way of sharing the abstract language (the string expression and the semantic representations share the same structure, namely t_1 and t_2);

- how scope ambiguity is modeled in type-logical approaches (with higher-order types and the different possible orders for the derivation rules)

- that as soon as the semantic transfer from the syntactic structure is functional (here by the lexicon), semantic ambiguity can only occur if there is syntactic ambiguity

- that parsing (in its usual sense in computational linguistics) requires both inverting a lexicon (here $\mathscr{L}_{syntax}^0$) and applying another one (here $\mathscr{L}_{sem}$.

This last remark is crucial for our proposal: in order to encode a non-functional relation, as the one exemplified between syntactic structure and semantic representation, we need to compose at least two ACG that share a same abstract language. Hence, we get a composition model like in figure 3.1 where $\mathscr{G}'$ builds the syntactic structures from strings, where $\mathscr{G}''$ builds the semantic representation from a new kind of structures which is related to the syntactic structure by $\mathscr{G}$. The next section expose our proposal based on that idea.

3.4 Encoding a Non-Functional Relation

The first step is to design an ACG that will model the relation between parse structures and string expressions, and more precisely to define its abstract signature. The requirement that quantifiers do not entail ambiguity at that level imposes they have not a higher-order type any more. This is the main difference with the signature Σ_{synt} we previously defined. In the new signature Σ_{syntax}, c_{every} has now the expected type $\mathsf{N} \multimap \mathsf{NP}$, which is not higher-order. Note we don't change the set of atomic types.

$$\Sigma_{\text{syntax}} = \begin{cases} A_{\text{synt}} \\ c_{\text{every}} \quad : \mathsf{N} \multimap \mathsf{NP} & c_{\text{man}} \quad : \mathsf{N} \\ c_{\text{some}} \quad : \mathsf{N} \multimap \mathsf{NP} & c_{\text{woman}} \quad : \mathsf{N} \\ c_{\text{love}} \quad : \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S} \end{cases}$$

$$\mathscr{L}_{syntax} = \begin{cases} \mathsf{N} := \textsc{String} \quad \mathsf{NP} := \textsc{String} \quad \mathsf{S} := \textsc{String} \\ c_{\text{every}} \quad := \lambda^o x.every + x & c_{\text{man}} \quad := man \\ c_{\text{some}} \quad := \lambda^o x.some + x & c_{\text{woman}} \quad := woman \\ c_{\text{love}} \quad := \lambda^o xy.x + loves + y \end{cases}$$

Let $\mathscr{G}_{\text{syntax}} = \langle \Sigma_{\text{syntax}}, \Sigma_{\text{string}}, \mathscr{L}_{syntax}, \mathsf{S} \rangle$ be an ACG. Contrary to the previous example, there now is a unique $t_0 \in \mathcal{A}(\mathscr{G}_{\text{syntax}})$ such that $\mathscr{L}_{syntax}(t_0) = every + man + loves + some + woman$. And $t_0 = c_{\text{love}}(c_{\text{every}}c_{\text{man}})(c_{\text{some}}c_{\text{woman}})$.

In order to make ambiguity appear, we need another ACG $\mathscr{G}_{\text{amb}}$ whose object signature is the abstract signature of $\mathscr{G}_{\text{syntax}}$. As abstract signature for $\mathscr{G}_{\text{amb}}$, we simply use Σ_{synt}. The key part is in the following lexicon:

$$\mathscr{L}_{amb} = \begin{cases} \mathsf{N} := \mathsf{N} \quad \mathsf{NP} := \mathsf{NP} \quad \mathsf{S} := \mathsf{S} \\ C_{\text{every}} \quad := \lambda^o xR.R(c_{\text{every}}x) & C_{\text{man}} \quad := c_{\text{man}} \\ C_{\text{some}} \quad := \lambda^o xR.R(c_{\text{some}}x) & C_{\text{woman}} \quad := c_{\text{woman}} \\ C_{\text{love}} \quad := c_{\text{loves}} \end{cases}$$

Note that only the constants dedicated to model quantifiers are changed (they get a higher-order type). With $\mathscr{G}_{\text{amb}} = \langle \Sigma_{\text{synt}}, \Sigma_{\text{syntax}}, \mathscr{L}_{amb}, \mathsf{S} \rangle$, we have $t_0 \in \mathcal{O}(\mathscr{G}_{\text{amb}})$ because

$$\mathscr{L}_{amb}(t_1) = \mathscr{L}_{amb}(t_2) = t_0$$

With $\mathscr{G}_{sem}$ unchanged, we are now able to associate to the expression *every man loves some woman* a single syntactic structure (namely t_0) and to semantic representations (namely $\mathscr{L}_{amb}(t_1)$ and $\mathscr{L}_{amb}(t_2)$). Note that pushing the higher-order type requirement apart from the syntactic side allows us to avoid the use of a special type constructor, such as $\Uparrow$, hence the need for the corresponding introduction and elimination rules.

(2) John and every kid ran

Sentences like (2) make the conjunction of quantified and non-quantified NPs. Look for instance at the following extensions (the name of the constants should make clear to which signature, lexicon or ACG this extension relates) to show how it works:

$$
\begin{aligned}
C_{\mathrm{run}} \quad &: \mathsf{NP} \multimap \mathsf{S} \\
C_{\mathrm{kid}} \quad &: \mathsf{N} \\
C_{\mathrm{John}} \quad &: \mathsf{NP} \\
C_{\mathrm{and}} \quad &: ((\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}) \multimap ((\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}) \multimap (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S} \\
c_{\mathrm{run}} \quad &: \mathsf{NP} \multimap \mathsf{S} \\
c_{\mathrm{kid}} \quad &: \mathsf{N} \\
c_{\mathrm{John}} \quad &: \mathsf{NP} \\
c_{\mathrm{and}} \quad &: \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{NP}
\end{aligned}
$$

$$
\mathscr{L}_{amb} = \begin{cases}
C_{\mathrm{run}} \quad := c_{\mathrm{run}} \\
C_{\mathrm{kid}} \quad := c_{\mathrm{kid}} \\
C_{\mathrm{John}} \quad := c_{\mathrm{John}} \\
C_{\mathrm{and}} \quad := \lambda^{\circ} PQR.P(\lambda^{\circ} x.Q(\lambda^{\circ} y.R(c_{\mathrm{and}}\, x\, y)))
\end{cases}
$$

$$
\mathscr{L}_{sem} = \begin{cases}
C_{\mathrm{run}} \quad := \mathbf{run} \\
C_{\mathrm{kid}} \quad := \mathbf{kid} \\
C_{\mathrm{John}} \quad := \mathbf{j} \\
C_{\mathrm{and}} \quad := \lambda^{\circ} PQ.\lambda R.(PR) \wedge (QR)
\end{cases}
$$

$$
\mathscr{L}_{syntax} = \begin{cases}
c_{\mathrm{run}} \quad := \lambda^{\circ} s.s + ran \\
c_{\mathrm{kid}} \quad := kid \\
c_{\mathrm{John}} \quad := John \\
c_{\mathrm{and}} \quad := \lambda^{\circ} xy.x + and + y
\end{cases}
$$

Because c_{and} has not a higher-order type, the parse structure of (2) is $t_3 = c_{\mathrm{run}}(c_{\mathrm{and}}c_{\mathrm{John}}(c_{\mathrm{every}}c_{\mathrm{kid}}))$, Then, since $\lambda^{\circ} P.PC_{\mathrm{John}} : (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}$ is derivable in our system, trying to find an antecedent to t_3 by $\mathscr{L}_{amb}$ we find $t_4 = C_{\mathrm{and}}(\lambda^{\circ} P.PC_{\mathrm{John}})(C_{\mathrm{every}}C_{\mathrm{kid}})C_{\mathrm{run}}$. t_4 has the usual structure of terms that represent the conjunction of quantified noun phrases and not quantified noun phrases in type-logical approaches. Then, with type raising of the non-quantified NP, we get the usual semantic representation from through $\mathscr{L}_{sem}$:

$$\mathscr{L}_{sem}(t_4) = (\mathbf{run\,j}) \wedge (\forall x.\mathbf{kid}\, x \Rightarrow \mathbf{run}\, x)$$

In the next sections, we show how to model some other phenomena that occur in sentences:

(3) a. John saw a kid and so did Bill

　　b. John seeks a book

　　c. every kid didn't run

3.4.1　Conjunction and Verbal Ellipsis

Whereas we make some simplification on the syntactic side, the following extensions enable the analysis of (3a):

$$
\begin{aligned}
C_{\mathrm{and\ so}} \quad &: (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S} \\
C_{\mathrm{see}} \quad &: \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S} \\
c_{\mathrm{and\ so}} \quad &: (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S} \\
c_{\mathrm{see}} \quad &: \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S}
\end{aligned}
$$

$$\mathcal{L}_{amb} = \begin{cases} C_{\text{see}} & := c_{\text{see}} \\ C_{\text{and so}} & := c_{\text{and so}} \end{cases}$$

$$\mathcal{L}_{sem} = \begin{cases} C_{\text{and so}} & := \lambda P.\lambda^{\text{o}} xy.(Px) \wedge (Py) \\ C_{\text{see}} & := \textbf{see} \end{cases}$$

$$\mathcal{L}_{syntax} = \begin{cases} c_{\text{see}} & := saw \\ c_{\text{and so}} & := \lambda^{\text{o}} Rxy.(Rx) + and\ so\ did + y \end{cases}$$

With these lexicon there is a unique parse structure $t_5 = c_{\text{and so}}(\lambda^{\text{o}} x.c_{\text{see}} x(c_{\text{a}} c_{\text{kid}})) c_{\text{John}} c_{\text{Bill}}$ for (3a). But with

$$t_6 = C_{\text{and so}}(\lambda^{\text{o}} x.C_{\text{a}} C_{\text{kid}}(\lambda^{\text{o}} y.C_{\text{see}} xy)) C_{\text{John}} C_{\text{Bill}}$$

and

$$t_7 = C_{\text{a}} C_{\text{kid}}(\lambda^{\text{o}} y.C_{\text{and so}}(\lambda x.C_{\text{see}} xy) C_{\text{John}} C_{\text{Bill}})$$

we have that $\mathcal{L}_{amb}(t_6) = \mathcal{L}_{amb}(t_7) = t_5$, hence two semantic readings:

$$\begin{aligned} \mathcal{L}_{sem}(t_6) &= (\exists x.(\textbf{kid}\,x) \wedge (\textbf{see}\,\textbf{j}\,x)) \wedge (\exists x.(\textbf{kid}\,x) \wedge (\textbf{see}\,\textbf{b}\,x)) \\ \mathcal{L}_{sem}(t_7) &= \exists x.(\textbf{kid}\,x) \wedge ((\textbf{see}\,\textbf{j}\,x) \wedge (\textbf{see}\,\textbf{b}\,x)) \end{aligned}$$

3.4.2 *De re* and *De dicto* Readings

This section shows how to model the *de re* and the *de dicto* readings of (3b).

$$\begin{aligned} C_{\text{seek}} &: \mathsf{NP} \multimap ((\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}) \multimap \mathsf{S} \\ C_{\text{book}} &: \mathsf{N} \\ c_{\text{seek}} &: \mathsf{NP} \multimap \mathsf{NP} \multimap \mathsf{S} \\ c_{\text{book}} &: \mathsf{N} \end{aligned}$$

$$\mathcal{L}_{amb} = \begin{cases} C_{\text{seek}} & := \lambda^{\text{o}} xP.P(\lambda^{\text{o}} y.c_{\text{seek}}\,x\,y) \\ C_{\text{book}} & := c_{\text{book}} \end{cases}$$

$$\mathcal{L}_{sem} = \begin{cases} C_{\text{seek}} & := \lambda^{\text{o}} xo.\textbf{try}\,x\,(\lambda^{\text{o}} z.o(\lambda^{\text{o}} y.\textbf{find}\,z\,y)) \\ C_{\text{book}} & := \textbf{book} \end{cases}$$

$$\mathcal{L}_{syntax} = \begin{cases} c_{\text{seek}} & := \lambda^{\text{o}} xy.x + seeks + y \\ c_{\text{book}} & := book \end{cases}$$

With these lexicon, (3b) has a unique parse structure $t_8 = c_{\text{seek}} c_{\text{John}}(c_{\text{a}} c_{\text{book}})$. But with

$$t_9 = C_{\text{seek}} C_{\text{John}}(C_{\text{a}} C_{\text{book}})$$

and

$$t_{10} = (C_{\text{a}} C_{\text{book}})(\lambda^{\text{o}} y.C_{\text{seek}} C_{\text{John}}\,(\lambda^{\text{o}} Q.Q\,y))$$

we have that $\mathcal{L}_{amb}(t_9) = \mathcal{L}_{amb}(t_{10}) = t_8$, hence two semantic readings:

$$\begin{aligned} \mathcal{L}_{sem}(t_9) &= \textbf{try}\,\textbf{j}\,(\lambda^{\text{o}} x.\exists y.(\textbf{book}\,y) \wedge (\textbf{find}\,x\,y)) \\ \mathcal{L}_{sem}(t_{10}) &= \exists y.(\textbf{book}\,y) \wedge (\textbf{try}\,\textbf{j}\,(\lambda^{\text{o}} x.\textbf{find}\,x\,y)) \end{aligned}$$

3.4.3 Quantification and Negation

Our last example shows how to parse (3c).

$$\begin{aligned} C_{\text{didnt}} &: (((\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}) \multimap \mathsf{S}) \multimap ((\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{S}) \multimap \mathsf{S} \\ c_{\text{didnt}} &: (\mathsf{NP} \multimap \mathsf{S}) \multimap \mathsf{NP} \multimap \mathsf{S} \end{aligned}$$

$$\begin{aligned} \mathcal{L}_{amb} &= \left\{ C_{\text{didnt}} \quad := \lambda^{\text{o}} PR.R(\lambda^{\text{o}} x.P(\lambda^{\text{o}} Q.c_{\text{didnt}}\,Q\,x)) \right. \\ \mathcal{L}_{sem} &= \left\{ C_{\text{didnt}} \quad := \lambda^{\text{o}} PQ.\neg(P\,Q) \right. \\ \mathcal{L}_{syntax} &= \begin{cases} c_{\text{didnt}} & := \lambda^{\text{o}} Rx.x + didn't + (R\,\epsilon) \\ c_{\text{book}} & := book \end{cases} \end{aligned}$$

With these lexicon, (3c) has a unique parse structure $t_{11} = c_{\text{didnt}}\, c_{\text{run}}(c_{\text{every}}c_{\text{kid}})$. But with

$$t_{12} = C_{\text{didnt}}(\lambda^{\text{o}}Q.Q\,C_{\text{run}})(C_{\text{every}}C_{\text{kid}})$$

and

$$t_{13} = (C_{\text{every}}C_{\text{kid}})(\lambda^{\text{o}}y.C_{\text{didnt}}(\lambda^{\text{o}}Q.Q\,C_{\text{run}})(\lambda^{\text{o}}P.P\,y))$$

we have that $\mathscr{L}_{amb}(t_9) = \mathscr{L}_{amb}(t_{10}) = t_8$, hence two semantic readings:

$$
\begin{array}{ll}
\mathscr{L}_{sem}(t_12) & = \neg(\forall x. C_{\text{kid}}\,x \Rightarrow C_{\text{run}}\,x)\\
\mathscr{L}_{sem}(t_{13}) & = \forall x. C_{\text{man}}\,x \Rightarrow \neg(C_{\text{run}}\,x)
\end{array}
$$

3.4.4 Current Limitations

There are some cases where it is not possible to extract quantifiers out of the relative clauses. For instance, (4) has no reading where the universal quantifier has scope over the existential one. As for now, we don't know how to express these constraints in the ACG formalism. Type-logical formalisms deal with these phenomena in using substructural logics and structural control Morrill, 1994.

(4) a man that every woman finds walks

The use of implicative linear logic, more precisely the first order fragment, has been proposed to model some of these phenomena Moot and Piazza, 2001. Extensions of the ACG type system in the same spirit have to be explored. In such a framework, we could also distinguish restrictions coming from syntax (with the type of the constants of Σ_{syntax}) and the restrictions coming from semantics (with the type of the constants of Σ_{synt}).

3.5 Comparison with Related Approaches

3.5.1 Scoping Constructor

Because the underlying type systems in the case of ACG and other type-logical formalisms may be quite different (associative and commutative *vs.* not associative, not commutative with structural rules), direct comparison between the expressive power of these approaches is not possible. However, we can compare ACG with a commutative and associative type-logical formalism (then there is only one implication, the linear implication). In that case we can rephrase the inference rules for this system with the scoping constructor. Types are augmented with this constructor:

$$\mathscr{TL}(A) ::= A\,|\,\mathscr{TL}(A) \multimap \mathscr{TL}(A)\,|\,\mathscr{TL}(A) \Uparrow \mathscr{TL}(A)$$

$$\frac{}{x : \alpha \vdash_{\text{TL}} x : \alpha}\ (\text{ax.})$$

$$\frac{\Gamma, x : \alpha \vdash_{\text{TL}} t : \beta}{\Gamma \vdash_{\text{TL}} \lambda^{\text{o}}x.t : \alpha \multimap \beta}\ (\text{abs.}) \qquad \frac{\Gamma_1 \vdash_{\text{TL}} t : \alpha \multimap \beta \qquad \Gamma_2 \vdash_{\text{TL}} u : \alpha}{\Gamma_1, \Gamma_2 \vdash_{\text{TL}} (t\,u) : \beta}\ (\text{app.})$$

$$\frac{\Gamma \vdash_{\text{TL}} t : \beta \Uparrow \alpha \qquad \Delta, x : \beta \vdash_{\text{TL}} u : \alpha}{\Gamma, \Delta \vdash_{\text{TL}} t(\lambda x.u) : \alpha}\ (\text{E}_{\Uparrow}) \qquad \frac{\Gamma \vdash_{\text{TL}} t : \beta}{\Gamma \vdash_{\text{TL}} \lambda x.(x\,t) : \beta \Uparrow \alpha}\ (\text{I}_{\Uparrow})$$

We define a translation from $\mathscr{TL}(A)$ to $\mathscr{T}(A)$ as follows:

- if $a \in A$ then $a^{\text{syn}} = a$ and $a^{\text{sem}} = a$

- if $a = \alpha \multimap \beta$ then $a^{\text{syn}} = \alpha^{\text{syn}} \multimap \beta^{\text{syn}}$ and $a^{\text{sem}} = \alpha^{\text{sem}} \multimap \beta^{\text{sem}}$. In the latter formula, $\multimap$ is called a main connective

- if $a = \alpha \Uparrow \beta$ then $a^{\text{syn}} = \alpha^{\text{syn}}$ and $a^{\text{sem}} = (\alpha^{\text{sem}} \multimap \beta^{\text{sem}}) \multimap \beta^{\text{sem}}$ and there is no main connective

- if $\Gamma = x_1 : a_1, \ldots, x_n : a_n$ then $\Gamma^{\text{syn}} = x_1 : a_1{}^{\text{syn}}, \ldots, x_n : a_n{}^{\text{syn}}$ and $\Gamma^{\text{sem}} = x_1 : a_1{}^{\text{sem}}, \ldots, x_n : a_n{}^{\text{sem}}$

Theorem 3.1. *Let $\Gamma \vdash_{TL} t : A$ be a TL sequent. If $\Gamma \vdash_{TL} t : A$ is provable then:*

- $\Gamma^{syn} \vdash A^{syn}$ *is provable;*

- *and $\Gamma^{sem} \vdash u : A^{sem}$ is provable;*

- *and if the last rule of the proof of $\Gamma^{sem} \vdash u : A^{sem}$ does not introduce a main connective, then $t = \lambda^{\text{o}} x.x\, v$;*

- *and $t =_\alpha u$.*

We only sketch the proof here.

Proof. By induction on the proofs. Let Π the proof of $\Gamma \vdash_{\text{TL}} t : A$. We look at the last rule of Π.

- If Π is reduced to an axiom, then $\Gamma^{\text{syn}} \vdash A^{\text{syn}}$ and $\Gamma^{\text{sem}} \vdash u : A^{\text{sem}}$ are provable and reduced to an axiom.

- If the last rule is (abs.) and Π ends with:

$$\frac{\Gamma, x : \alpha \vdash_{\text{TL}} u : \beta}{\Gamma \vdash_{\text{TL}} \lambda^{\text{o}} x.u : \alpha \multimap \beta} \text{ (abs.)}$$

  Then $\Gamma, x : \alpha \vdash_{\text{TL}} u : \beta$ is provable, and by induction hypothesis, $\Gamma, x : \alpha^{\text{syn}} \vdash w : \beta^{\text{syn}}$ and $\Gamma, x : \alpha^{\text{sem}} \vdash v : \beta^{\text{sem}}$ and $v =_\alpha u$. So

$$\frac{\Gamma, x : \alpha^{\text{syn}} \vdash \beta^{\text{syn}}}{\Gamma^{\text{syn}} \vdash \lambda^{\text{o}} x.w : \alpha \multimap \beta^{\text{syn}}}$$

  proves $\Gamma^{\text{syn}} \vdash \lambda^{\text{o}} x.w : \alpha \multimap \beta^{\text{syn}}$ is provable, and

$$\frac{\Gamma, x : \alpha^{\text{sem}} \vdash v : \beta^{\text{sem}}}{\Gamma^{\text{sem}} \vdash \lambda^{\text{o}} x.v : \alpha \multimap \beta^{\text{sem}}}$$

  with the latter $\multimap$ being a main connective. Moreover, since $v =_\alpha u$, we also have $\lambda^{\text{o}} x.v =_\alpha \lambda^{\text{o}} x.u$.

- If the last rule is (app.) and Π ends with:

$$\frac{\Gamma_1 \vdash_{\text{TL}} t : \alpha \multimap \beta \qquad \Gamma_2 \vdash_{\text{TL}} u : \alpha}{\Gamma_1, \Gamma_2 \vdash_{\text{TL}} (t\, u) : \beta} \text{ (app.)}$$

- If the last rule is $(E_\Uparrow)$ and Π ends with:

$$\frac{\Gamma \vdash_{\text{TL}} t : \beta \Uparrow \alpha \qquad \Delta, x : \beta \vdash_{\text{TL}} u : \alpha}{\Gamma, \Delta \vdash_{\text{TL}} t(\lambda x.u) : \alpha}$$

  by induction hypothesis, we have that $\Gamma^{\text{syn}} \vdash t : \beta \Uparrow \alpha^{\text{syn}}$ is provable, that is $\Gamma^{\text{syn}} \vdash t : \beta^{\text{syn}}$ is provable and $\Delta^{\text{syn}}, x : \beta^{\text{syn}} \vdash u : \alpha^{\text{syn}}$ is provable. Hence, $\Delta^{\text{syn}} \vdash \lambda^{\text{o}} x.u : \beta^{\text{syn}} \multimap \alpha^{\text{syn}}$ is provable and $\Gamma^{\text{syn}}, \Delta^{\text{syn}} \vdash (u\, t) : \alpha^{\text{syn}}$ is provable.

  Moreover, by induction hypothesis we have that $\Gamma^{\text{sem}} \vdash t' : \beta \Uparrow \alpha^{\text{sem}}$ is provable, that is $\Gamma^{\text{sem}} \vdash t' : (\beta^{\text{sem}} \multimap \alpha^{\text{sem}}) \multimap \alpha^{\text{sem}}$ is provable. Moreover, $t' = \lambda^{\text{o}} x.x\, v'$ since the $\multimap$ is not a main connective. We also have $\Delta^{\text{sem}}, x : \beta^{\text{sem}} \vdash u' : \alpha^{\text{sem}}$ is provable, hence $\Delta^{\text{sem}} \vdash \lambda^{\text{o}} x.u' : \beta^{\text{sem}} \multimap \alpha^{\text{sem}}$ provable. Then, $\Gamma^{\text{sem}}, \Delta^{\text{sem}} \vdash (\lambda^{\text{o}} x.x\, v')(\lambda^{\text{o}} x.u') : \alpha^{\text{sem}}$ is provable, that is $\Gamma^{\text{sem}}, \Delta^{\text{sem}} \vdash (\lambda^{\text{o}} x, u')v' : \alpha^{\text{sem}}$, and $\Gamma^{\text{sem}}, \Delta^{\text{sem}} \vdash u'[v'/x] : \alpha^{\text{sem}}$

- If the last rule is $(I_\Uparrow)$ and Π ends with:

$$\frac{\Gamma \vdash_{\mathrm{TL}} t : \beta}{\Gamma \vdash_{\mathrm{TL}} \lambda x.(x\,t) : \beta \Uparrow \alpha}$$

by induction hypothesis, we have $\Gamma^{\mathrm{syn}} \vdash t : \beta^{\mathrm{syn}}$ provable, that is $\Gamma^{\mathrm{syn}} \vdash t : \beta \Uparrow \alpha^{\mathrm{syn}}$ provable.

On the semantic side, we have $\Gamma^{\mathrm{sem}} \vdash t' : \beta^{\mathrm{sem}}$ provable. So we can build the following proof:

$$\frac{\dfrac{\Gamma^{\mathrm{sem}} \vdash t' : \beta^{\mathrm{sem}} \qquad x : \beta^{\mathrm{sem}} \multimap \alpha^{\mathrm{sem}} \vdash x : \beta^{\mathrm{sem}} \multimap \alpha^{\mathrm{sem}}}{\Gamma^{\mathrm{sem}}, x : \beta^{\mathrm{sem}} \multimap \alpha^{\mathrm{sem}} \vdash (x\,t') : \alpha^{\mathrm{sem}}}}{\Gamma^{\mathrm{sem}} \vdash \lambda^{\mathrm{o}} x.(x\,t') : (\beta^{\mathrm{sem}} \multimap \alpha^{\mathrm{sem}}) \multimap \alpha^{\mathrm{sem}}}$$

$\square$

Conjecture 3.2. *The implication in Theorem 3.1 is actually an equivalence.*

This basically means that the two proposals are able to model the same phenomena, albeit in different ways. We don't make explicit here the role of the lexicon (and how to define it) so that it ensures the relation between the syntactic and the semantic proofs (namely that the image of the semantic proof is the syntactic one).

3.5.2 Glue Semantics

Glue Semantics (GS) Dalrymple, 1999 was introduced to compute semantic representations from LFG structures, but it has also been applied to other formalisms HPSG Asudeh and Crouch, 2001 and TAG Frank and van Genabith, 2001). The principle is to associate to a parse structure a logical formula (of intuitionistic linear logic) which has to be proven. As in the our approach and in the type-logical approach, this yields possibly many proofs which are then directly turned, via the Curry-Howard isomorphism, into possibly many semantic interpretations. So GS clearly does not fit in the underspecification way to model ambiguities.

It also differ from the type-logical approach by the fact that the syntactic structure does not directly translate into the semantic structure but rather by the way of stating a sequent to prove. This clearly relates to the way the syntactic structure imposes conditions on the semantic structure in our proposal (let u be the parse structure, prove $t \in \mathcal{A}(\mathscr{G}_{\mathrm{sem}})$ and $\mathscr{L}_{sem}(t) = u$). But a formal comparison on GS and ACG is beyond the scope of this chapter.

3.6 Conclusion

This chapter shows how to encode the non-functional relation between syntactic structures and semantic representations in a proof-theoretic setting. This differs from the standard type-logical approach, where semantic ambiguity requires syntactic ambiguity, and from the underspecification framework, where ambiguity is expressed by a specific language. The ACG framework in which it take place also provide a modularity in the way constrains can be described either on the syntactic level or in the semantic level. Moreover, this proposal gives hints on how to extend the type system to take such constraints into account.

Because ACG can model different grammatical formalisms, we think it can help to share insights from different these formalisms and to mix with pragmatic models. It also gives a first base to a comparison between GS and ACG.

On the Syntax-Semantics Interface: From CVG to ACG

This chapter has been published as (de Groote, Pogodalla, and Pollard, 2009), and, consequently, presents a joint work. Cooper's storage technique for scoping *in situ* operators has been employed in theoretical and computational grammars of natural language (NL) for over thirty years, but has been widely viewed as ad hoc and unprincipled. Recent work by Pollard within the framework of convergent grammar (CVG) took a step in the direction of clarifying the logical status of Cooper storage by encoding its rules within an explicit but nonstandard natural deduction (ND) format. Here we provide further clarification by showing how to encode a CVG with storage within a logical grammar framework—abstract categorial grammar (ACG)—that utilizes no logical resources beyond those of standard linear deduction.

Introduction

A long-standing challenge for designers of NL grammar frameworks is posed by **in situ operators**, expressions such as quantified noun phrases (QNPs, e.g. *every linguist*), wh-expressions (e.g. *which linguist*), and comparative phrases (e.g. *more than five dollars*), whose semantic scope is underdetermined by their syntactic position. One family of approaches, employed by computational semanticists Blackburn and Bos, 2005 and some versions of categorial grammar Bach and Partee, 1980 and phrase structure grammar Cooper, 1983; Pollard and Sag, 1994 employs the **storage** technique first proposed by Cooper Cooper, 1975. In these approaches, syntactic and semantic derivations proceed in parallel, much as in classical Montague grammar (CMG Montague, 1973) except that sentences which differ only with respect to the scope of in-situ operators have identical syntactic derivations.[1] Where they differ is in the semantic derivations: the meaning of an in-situ operator is *stored* together with a copy of the variable that occupies the hole in a delimited semantic continuation over which the stored operator will scope when it is *retrieved*; ambiguity arises from nondeterminism with respect to the retrieval site.

Although storage is easily grasped on an intuitive level, it has resisted a clear and convincing logical characterization, and is routinely scorned by theoreticians as 'ad hoc', 'baroque', or 'unprincipled'. Recent work Pollard, 2011a,b within the CVG framework provided a partial clarification by encoding storage and retrieval rules within a somewhat nonstandard ND semantic calculus (Section 4.1). The aim of this chapter is to provide a logical characterization of storage/retrieval free of nonstandard features. To that end, we provide an explicit transformation of CVG interface derivations (parallel syntax-semantic derivations) into a framework (ACG de Groote, 2001) that employs no logical resources beyond those of standard (linear)

---

[1]In CMG, syntactic derivations for different scopings of a sentence differ with respect to the point from which a QNP is 'lowered' into the position of a syntactic variable.

natural deduction. Section 4.2 provides a preliminary conversion of CVG by showing how to re-express the storage and retrieval rules (respectively) by standard ND hypotheses and another rule already present in CVG (analogous to Gazdar's Gazdar, 1981 rule for unbounded dependencies). Section 4.3 introduces the target framework ACG. And Sect. 4.4 describes the transformation of a (pre-converted) CVG into an ACG.

4.1 Convergent Grammar

A CVG for an NL consists of three term calculi for syntax, semantics, and the interface. The syntactic calculus is a kind of applicative multimodal categorial grammar, the semantic calculus is broadly similar to a standard typed lambda calculus, and the interface calculus recursively specifies which syntax-semantics term pairs belong to the NL.[2] Formal presentation of these calculi are given in Appendix A.1.

In the **syntactic calculus**, types are syntactic categories, constants (nonlogical axioms) are words (broadly construed to subsume phrasal affixes, including intonationally realized ones), and variables (assumptions) are traces (axiom schema T), corresponding to 'overt movement' in generative grammar. Terms are (candidate syntactic analyses of) words and phrases.

For simplicity, we take as our basic syntactic types NP (noun phrase), S (nontopicalized sentence), and t (topicalized sentence). Flavors of implication correspond not to directionality (as in Lambek calculus) but to grammatical functions. Thus syntactic arguments are explicitly identitifed as subjects ($\multimap_{\mathrm{s}}$), complements ($\multimap_{\mathrm{c}}$), or hosts of phrasal affixes ($\multimap_{\mathrm{a}}$). Additionally, there is a ternary ('Gazdar') type constructor A_B^C for the category of 'overtly moved' phrases that bind an A-trace in a B, resulting in a C.

Contexts (left of the $\vdash$) in syntactic rules represent unbound traces. The elimination rules (flavors of modus ponens) for the implications, also called merges (M), combine 'heads' with their syntactic arguments. The elimination rule G for the Gazdar constructor implements Gazdar's (Gazdar, 1981) rule for discharging traces; thus G compiles in the effect of a hypothetical proof step (trace binding) immediately and obligatorily followed by the consumption of the resulting abstract by the 'overtly moved' phrase. G requires no introduction rule because it is only introduced by lexical items ('overt movement triggers' such as wh-expressions, or the prosodically realized topicalizer).

In the CVG **semantic calculus**, as in familiar semantic λ-calculi, terms correspond to meanings, constants to word meanings, and implication elimination to function application. But there is no λ-abstraction! Instead, binding of semantic variables is effected by either (1) a semantic 'twin' of the Gazdar rule, which binds the semantic variable corresponding to a trace by (the meaning of) the 'overtly moved' phrase; or (2) by the Responsibility (retrieval) rule (R), which binds the semantic variable that marks the argument position of a stored ('covertly moved') *in situ* operator. Correspondingly, there are two mechanisms for introducing semantic variables into derivations: (1) ordinary hypotheses, which are the semantic counterparts of ('overt movement') traces; and the Commitment (Cooper storage) rule (C), which replaces a semantic operator a of type A_B^C with a variable $x : A$ while placing a (subscripted by x) in the store (also called the *co-context*), written to the left of the $\dashv$ (called co-turnstile).

The CVG **interface calculus** recursively defines a relation between syntactic and semantic terms. Lexical items pair syntactic words with their meanings. Hypotheses pair a trace with a semantic variable and enter the pair into the context. The C rule leaves the syntax of an *in situ* operator unchanged while storing its meaning in the co-context. The implication elimination rules pair each (subject-, complement-, or affix-)flavored syntactic implication elimination rule with ordinary semantic implication elimination. The G rule simultaneously binds a trace by an 'overtly moved' syntactic operator and a semantic variable by the corresponding semantic operator. And the R rule leaves the syntax of the retrieval site unchanged while binding a 'committed' semantic variable by the retrieved semantic operator.

---

[2]To handle phonology, ignored here, a fourth calculus is needed; and then the interface specifies phonology/syntax/semantics triples.

4.2   About the Commitment and Retrieve Rules

In the CVG semantic calculus, C and R are the only rules that make use of the store (co-context), and their logical status is not obvious. This section shows that they can actually be derived from the other rules, in particular from the G rule. Indeed, the derivation on the left can be replaced by the one on the right[3]: one:

$$
\dfrac{\dfrac{\vdots \pi_1}{\Gamma \vdash a : A_B^C \dashv \Delta}}{\Gamma \vdash x : A \dashv a_x : A_B^C, \Delta}\ \mathrm{C}
$$

$$
\dfrac{\vdots \pi_2}{\dfrac{\Gamma, \Gamma' \vdash b : B \dashv a_x : A_B^C, \Delta', \Delta}{\Gamma, \Gamma' \vdash a_x\, b : C \dashv \Delta', \Delta}}\ \mathrm{R}
$$

$\rightsquigarrow$

$$
\dfrac{\overline{x : A \vdash x : A \dashv}}{}
$$

$$
\dfrac{\dfrac{\vdots \pi_1}{\Gamma \vdash a : A_B^C \dashv \Delta} \quad \dfrac{\vdots \pi_2}{x : A, \Gamma' \vdash b : B \dashv \Delta'}}{\Gamma, \Gamma' \vdash a_x\, b : C \dashv \Delta, \Delta'}\ \mathrm{G}
$$

This shows we can eliminate the store, resulting in a more traditional presentation of the underlying logical calculus. On the other hand, in the CVG interface calculus, this technique for elimiating C and R rules does not quite go through because the G rule requires both the syntactic type and the semantic type to be of the form α_β^γ. This difficulty is overcome by adding the following Shift rule to the interface calculus:

$$
\dfrac{\Gamma \vdash a, b : A, B_C^D \dashv \Delta}{\Gamma \vdash \mathrm{S}_E\, a, b : A_E^E, B_C^D \dashv \Delta}\ \mathrm{Shift}_E
$$

where S_E is a functional term whose application to an A produces a A_E^E. Then we can transform

$$
\dfrac{\dfrac{\vdots \pi_1}{\Gamma \vdash a, b : A, B_C^D \dashv \Delta}}{\Gamma \vdash a, x : A, B \dashv b_x : B_C^D, \Delta}\ \mathrm{C}
$$

$$
\dfrac{\dfrac{\vdots \pi_2}{\Gamma, \Gamma' \vdash e, c : E, C \dashv b_x : B_C^D, \Delta, \Delta'}}{\Gamma, \Gamma' \vdash e, b_x\, c : E, D \dashv \Delta', \Delta}\ \mathrm{R}
$$

to:

$$
\dfrac{\overline{t, x : A, B \vdash t, x : A, B \dashv}}{}
$$

$$
\dfrac{\dfrac{\dfrac{\vdots \pi_1}{\Gamma \vdash a, b : A, B_C^D \dashv \Delta}}{\Gamma \vdash \mathrm{S}_E\, a, b : A_E^E, B_C^D \dashv \Delta}\ \mathrm{Shift}_E \quad \dfrac{\vdots \pi_2}{t, x : A, B; \Gamma' \vdash e, c : E, C \dashv \Delta'}}{\Gamma, \Gamma' \vdash (\mathrm{S}_E a)_t\, e, b_x\, c : E, D \dashv \Delta, \Delta'}\ \mathrm{G}
$$

provided $(\mathrm{S}_E a)_t\, e = (\mathrm{S}_E\, a)\,(\lambda t.e) = e[t := a]$. This follows from β-reduction as long as we take S_E to be $\lambda y\, P.P\, y$. Indeed:

$$
(\mathrm{S}_E\, a)\,(\lambda t.e) = (\lambda y\, P.P\, y)\, a\,(\lambda t.e) =_\beta (\lambda P.P\, a)\,(\lambda t.e) =_\beta (\lambda t.e)\, a =_\beta e[t := a]
$$

With this additional construct, we can get rid of the C and R rules in the CVG interface calculus. This construct is used in Section 4.4 to encode CVG into ACG. It can be seen as a rational reconstruction of Montague's quantifier lowering technique as nothing more than β-reduction in the syntax (unavailable to Montague since his syntactic calculus was purely applicative).

---

[3]The fact that we can divide the context into Γ and Γ' and the store into Δ and Δ', and that Γ and Δ are preserved, is shown in Proposition A.1 of Appendix A.2.

4.3   Abstract Categorial Grammar

Motivations.   Abstract Categorial Grammars (ACGs) de Groote, 2001, which derive from type-theoretic grammars in the tradition of Lambek Lambek, 1958, Curry Curry, 1961, and Montague Montague, 1973, provide a framework in which several grammatical formalisms may be encoded de Groote and Pogodalla, 2004. The definition of an ACG is based on a small set of mathematical primitives from type-theory, λ-calculus, and linear logic. These primitives combine via simple composition rules, which offers ACGs a good flexibility. In particular, ACGs generate languages of linear λ-terms, which generalizes both string and tree languages. They also provide the user direct control over the parse structures of the grammar, which allows several grammatical architectures to be defined in terms of ACG.

Mathematical preliminaries.   Let A be a finite set of atomic types, and let $\mathscr{T}_A$ be the set of linear functional types types (in notation, $\alpha \multimap \beta$) built upon A. A *higher-order linear signature* is then defined to be a triple $\Sigma = \langle A, C, \tau \rangle$, where: A is a finite set of atomic types; C is a finite set of constants; and τ is a mapping from C to $\mathscr{T}_A$. A higher-order linear signature will also be called a *vocabulary*. In the sequel, we will write A_Σ, C_Σ, and τ_Σ to designate the three components of a signature Σ, and we will write $\mathscr{T}_\Sigma$ for $\mathscr{T}_{A_\Sigma}$.

We take for granted the definition of a λ-term, and we let the relation of $\beta\eta$-conversion to be the notion of equality between λ-terms. Given a higher-order signature Σ, we write Λ_Σ for the set of *linear simply-typed λ-terms*.

Let Σ and Ξ be two higher-order linear signatures. A *lexicon* $\mathscr{L}$ from Σ to Ξ (in notation, $\mathscr{L} : \Sigma \longrightarrow \Xi$) is defined to be a pair $\mathscr{L} = \langle \eta, \theta \rangle$ such that: η is a mapping from A_Σ into $\mathscr{T}_\Xi$; θ is a mapping from C_Σ into Λ_Ξ; and for every $c \in C_\Sigma$, the following typing judgement is derivable: $\vdash_\Xi \theta(c) : \hat{\eta}(\tau_\Sigma(c))$, where $\hat{\eta} : \mathscr{T}_\Sigma \to \mathscr{T}_\Xi$ is the unique homomorphic extension of η.[4]

Let $\hat{\theta} : \Lambda_\Sigma \to \Lambda_\Xi$ be the unique λ-term homomorphism that extends θ.[5] We will use $\mathscr{L}$ to denote both $\hat{\eta}$ and $\hat{\theta}$, the intended meaning being clear from the context. When Γ denotes a typing environment '$x_1 : \alpha_1, \ldots, x_n : \alpha_n$', we will write $\mathscr{L}(\Gamma)$ for '$x_1 : \mathscr{L}(\alpha_1), \ldots, x_n : \mathscr{L}(\alpha_n)$'. Using these notations, we have that the last condition for $\mathscr{L}$ induces the following property: if $\Gamma \vdash_\Sigma t : \alpha$ then $\mathscr{L}(\Gamma) \vdash_\Xi \mathscr{L}(t) : \mathscr{L}(\alpha)$.

Definition 4.1. *An* abstract categorial grammar *is a quadruple* $\mathscr{G} = \langle \Sigma, \Xi, \mathscr{L}, s \rangle$ *where:*

1. Σ *and* Ξ *are two higher-order linear signatures, which are called the* abstract vocabulary *and the* object vocabulary*, respectively;*

2. $\mathscr{L} : \Sigma \longrightarrow \Xi$ *is a lexicon from the abstract vocabulary to the object vocabulary;*

3. $s \in \mathscr{T}_\Sigma$ *is a type of the abstract vocabulary, which is called the* distinguished type *of the grammar.*

A possible intuition behind this definition is that the object vocabulary specifies the surface structures of the grammars, the abstract vocabulary specifies its abstract parse structures, and the lexicon specifies how to map abstract parse structures to surface structures. As for the distinguished type, it plays the same part as the start symbol of the phrase structures grammars. This motivates the following definitions.

The *abstract language* of an ACG is the set of closed linear λ-terms that are built on the abstract vocabulary, and whose type is the distinguished type:

$$\mathcal{A}(\mathscr{G}) = \{ t \in \Lambda_\Sigma \,|\, \vdash_\Sigma t : s \text{ is derivable} \}$$

On the other hand, the object language of the grammar is defined to be the image of its abstract language by the lexicon:

$$\mathcal{O}(\mathscr{G}) = \{ t \in \Lambda_\Xi \,|\, \exists u \in \mathcal{A}(\mathscr{G}). \, t = \mathscr{L}(u) \}$$

---

[4]That is $\hat{\eta}(a) = \eta(a)$ and $\hat{\eta}(\alpha \multimap \beta) = \hat{\eta}(\alpha) \multimap \hat{\eta}(\beta)$.

[5]That is $\hat{\theta}(c) = \theta(c)$, $\hat{\theta}(x) = x$, $\hat{\theta}(\lambda x. t) = \lambda x. \hat{\theta}(t)$, and $\hat{\theta}(t\, u) = \hat{\theta}(t)\, \hat{\theta}(u)$.

It is important to note that, from a purely mathematical point of view, there is no structural difference between the abstract and the object vocabulary: both are higher-order signatures. Consequently, the intuition we have given above is only a possible interpretation of the definition, and one may conceive other possible grammatical architectures. Such an architecture consists of two ACGs sharing the same abstract vocabulary, the object vocabulary of the first ACG corresponding to the syntactic structures of the grammar, and the one of the second ACG corresponding to the semantic structures of the grammar. Then, the common abstract vocabulary corresponds to the transfer structures of the syntax/semantics interface. This is precisely the architecture that the next section will exemplify.

4.4 ACG encoding of CVG

The Overall Architecture. As Section 4.1 shows, whether a pair of a syntactic term and a semantic term belongs to the language depends on whether it is derivable from the lexicon in the CVG interface calculus. Such a pair is indeed an *(interface) proof term* corresponding to the derivation. So the first step towards the encoding of CVG into ACG is to provide an abstract language that generates the same proof terms as those of the CVG interface. For a given CVG G, we shall call $\Sigma_{I(G)}$ the higher-order signature that will generate the same proof terms as G. Then, any ACG whose abstract vocabulary is $\Sigma_{I(G)}$ will generate these proof terms. And indeed we will use two ACG sharing this abstract vocabulary to map the (interface) proof terms into syntactic terms and into semantic terms respectively. So we need two other signatures: one allowing us to express the syntactic terms, which we call $\Sigma_{\text{SimpleSyn}(G)}$, and another allowing us to express the semantic terms, which we call $\Sigma_{\text{Log}(G)}$.

Finally, we need to be able to recover the two components of the pair out of the proof term of the interface calculus. This means having two ACG sharing the same abstract language (the closed terms of $\Lambda(\Sigma_{I(G)})$ of some distinguished type) and whose object vocabularies are respectively $\Sigma_{\text{SimpleSyn}(G)}$ and $\Sigma_{\text{Log}(G)}$. Fig. 4.1 illustrates the architecture with $\mathscr{G}_{\text{Syn}} = \langle \Sigma_{I(G)}, \Sigma_{\text{SimpleSyn}(G)}, \mathscr{L}_{\text{Syn}}, s \rangle$ the first ACG that encodes the mapping from interface proof terms to syntactic terms, and $\mathscr{G}_{\text{Sem}} = \langle \Sigma_{I(G)}, \Sigma_{\text{Log}(G)}, \mathscr{L}_{\text{Log}}, s \rangle$ the second ACG that encodes the mapping from interface proof terms to semantic formulas. It should be clear that this architecture can be extended so as to get phonological forms and conventional logical forms (say, in TY_2) using similar techniques. The latter requires non-linear λ-terms, an extension already available to ACG de Groote and Maarek, 2007 . So we focus here on the (simple) syntax-semantics interface only, which requires only linear terms.
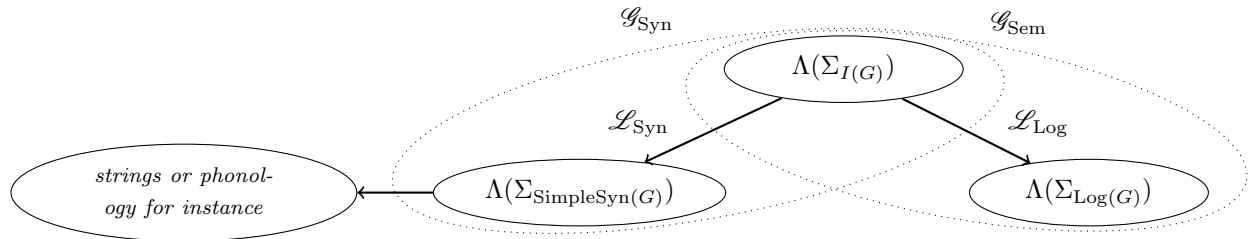
Figure 4.1: Overall architecture of the ACG encoding of a CVG

We begin by providing an example of a CVG lexicon (Table 4.1). Recall that the syntactic type t is for overtly topicalized sentences, and $\multimap a$ is the flavor of implication for affixation. We recursively define the translation $\overline{\cdot}^\tau$ of CVG pairs of syntactic and semantics types to $\Sigma_{I(G)}$ as:

- $\overline{\alpha, \beta}^\tau = \langle \alpha, \beta \rangle$ if either α or β is atomic or of the form γ_δ^ϵ. Note that this new type $\langle a, \beta \rangle$ is an *atomic* type of $\Sigma_{I(G)}$;

- $\overline{\alpha \multimap \beta, \alpha' \multimap \beta'}^\tau = \overline{\alpha, \alpha'}^\tau \multimap \overline{\beta, \beta'}^{\tau}$[6].

---

[6]This translation preserves the order of the types. Hence, in the ACG settings, it allows abstraction everywhere. This does not fulfill one of the CVG requirements. However, since it is always possible from an ACG $\mathscr{G}$ to build a new ACG $\mathscr{G}'$ such that

When ranging over the set of types provided by the CVG lexicon[7], we get all the atomic types of $\Sigma_{I(G)}$. Then, for any $w, f : \alpha, \beta$ of the CVG lexicon of G, we add the constant $\overline{w, f}^{\,\text{c}} = \text{w}$ of type $\overline{\alpha, \beta}^{\,\tau}$ to the signature $\Sigma_{I(G)}$.

The application of $\overline{\cdot}^{\,\text{c}}$ and $\overline{\cdot}^{\,\tau}$ to the lexicon of Table 4.1 yields the signature $\Sigma_{I(G)}$ of Table 4.2. Being able to use the constants associated to the topicalization operators in building new terms requires additional constants having e.g. $\langle \text{NP}, e_\pi^\pi \rangle$ as parameters. We delay this construct to Sect. 4.4.

Table 4.1: CVG lexicon for topicalization

Chris, Chris' :	NP, e	top, top' :	$\text{NP} \multimap_{\text{a}} \text{NP}_{\text{S}}^{t}, e \multimap e_\pi^\pi$
liked, like' :	$\text{NP} \multimap_{\text{c}} \text{NP} \multimap_{\text{s}} \text{S}, e \multimap e \multimap \pi$	$\text{top}_{\text{in-situ}}$, top' :	$\text{NP} \multimap_{\text{a}} \text{NP}, e \multimap e_\pi^\pi$

Table 4.2: ACG translation of the CVG lexicon for topicalization

Chris :	$\langle \text{NP}, e \rangle$	top :	$\langle \text{NP}, e \rangle \multimap \langle \text{NP}_{\text{S}}^{t}, e_\pi^\pi \rangle$
liked :	$\langle \text{NP}, e \rangle \multimap \langle \text{NP}, e \rangle \multimap \langle \text{S}, \pi \rangle$	top$_{\text{in-situ}}$:	$\langle \text{NP}, e \rangle \multimap \langle \text{NP}, e_\pi^\pi \rangle$

Constants and types in $\Sigma_{\text{SimpleSyn}(G)}$ and $\Sigma_{\text{Log}(G)}$ simply reflect that we want them to build terms in the syntax and in the semantics respectively. First, note that a term of type α_β^γ, according to the CVG rules, can be applied to a term of type $\alpha \multimap \beta$ to return a term of type γ. Moreover, the type α_β^γ does not exist in any of the ACG object vocabularies. Hence we recursively define the $[\![\cdot]\!]$ function that turns CVG syntactic and semantic types into linear types (as used in higher-order signatures) as:

- $[\![a]\!] = a$ if a is atomic

- $[\![\alpha_\beta^\gamma]\!] = ([\![\alpha]\!] \multimap [\![\beta]\!]) \multimap [\![\gamma]\!]$

- $[\![\alpha \multimap_{\text{x}} \beta]\!] = [\![\alpha]\!] \multimap [\![\beta]\!]$

Then, for any CVG constant $w, f : \alpha, \beta$ we have $\overline{w, f}^{\,\text{c}} = \text{w} : \overline{\alpha, \beta}^{\,\tau}$ in $\Sigma_{I(G)}$:

$$\begin{aligned}
\mathscr{L}_{\text{Syn}}(\text{w}) &= w & \mathscr{L}_{\text{Log}}(\text{w}) &= f \\
\mathscr{L}_{\text{Syn}}(\overline{\alpha, \beta}^{\,\tau}) &= [\![\alpha]\!] & \mathscr{L}_{\text{Log}}(\overline{\alpha, \beta}^{\,\tau}) &= [\![\beta]\!]
\end{aligned}$$

So the lexicon of Table 4.1 gives[8]:

$$\begin{aligned}
\mathscr{L}_{\text{Syn}}(\text{Chris}) &= \text{Chris} & \mathscr{L}_{\text{Syn}}(\text{liked}) &= \lambda xy. \big[{}^{\text{s}}y \, \big[\text{liked } x^{\,\text{c}} \big] \big] \\
\mathscr{L}_{\text{Log}}(\text{Chris}) &= \text{Chris'} & \mathscr{L}_{\text{Log}}(\text{liked}) &= \lambda xy.\text{like' } y \, x
\end{aligned}$$

And we get the trivial translations:

$$\begin{aligned}
\mathscr{L}_{\text{Syn}}(\text{liked sandy chris}) &= \big[{}^{\text{s}}\text{Chris } \big[\text{liked Sandy}^{\,\text{c}} \big] \big] : \text{S} \\
\mathscr{L}_{\text{Log}}(\text{liked sandy chris}) &= \text{like' Chris' Sandy'} : \pi
\end{aligned}$$

On the Encoding of CVG Rules. There is a trivial one-to-one mapping between the CVG rules Lexicon, Trace, and Subject and Complement Modus Ponens, and the standard typing rules of linear λ-calculus of ACG: constant typing rule (non logical axiom), identity rule and application. So the ACG derivation that proves $\vdash_{\Sigma_{I(G)}} \text{liked sandy chris} : \langle \text{S}, \pi \rangle$ in $\Lambda(\Sigma_{I(G)})$ is isomorphic to the CVG derivation $\vdash \big[{}^{\text{s}}\text{Chris } \big[\text{liked Sandy}^{\,\text{c}} \big] \big], \text{like' Sandy' Chris'} : \text{S}, \pi \dashv$. But the CVG G rule has no counterpart in the ACG type system. So it needs to be introduced using constants in $\Sigma_{I(G)}$.

Let's assume a CVG derivation using the following rule:

$\mathcal{O}(\mathscr{G}') = \{ t \in \mathcal{A}(\mathscr{G}) | t$ consists only in applications$\}$ (see the construct in Appendix B), we can assume without loss of generality that we here deal only with second order terms.

[7]Actually, we should also consider additional types issuing from types of the form α_β^γ when one of the α, β or γ is itself a type of this form.

[8]In order to help recognizing the CVG syntactic forms, we use additional operators of arity 2 in $\Sigma_{\text{SimpleSyn}(G)}$: $\big[{}^{\text{s}} p \big]$ instead of writing $(p \, s)$ when p is of type $\alpha \multimap_{\text{s}} \beta$ and $\big[p \, c^{\,\text{x}} \big]$ instead of just $(p \, c)$ when p is of type $\alpha \multimap_{\text{x}} \beta$ with $x \neq s$. This syntactic sugar is not sufficient to model the different flavors of the implication in CVG, the latter topic being beyond the scope of this paper.

$$\frac{\Gamma \vdash a, d : A_B^C, D_E^F \dashv \Delta \qquad t, x : A, D; \Gamma' \vdash b, e : B, E \dashv \Delta'}{\Gamma; \Gamma' \vdash a_t\, b, d_x\, e : C, F \dashv \Delta; \Delta'}\ \text{G}$$

with $\overset{\vdots}{\pi_1}$ over the left premise and $\overset{\vdots}{\pi_2}$ over the right premise.

and that we are able to build two terms (or two ACG derivations) $\mathrm{T}_1 : \langle A_B^C, D_E^F \rangle$ and $\mathrm{T}_2 : \overline{B, E}^{\,\tau}$ of $\Lambda(\Sigma_{I(G)})$ corresponding to the two CVG derivations π_1 and π_2. Then, adding a constant $G_{\langle A_B^C, D_E^F \rangle}$ of type $\langle A_B^C, D_E^F \rangle \multimap (\overline{A, D}^{\,\tau} \multimap \overline{B, E}^{\,\tau}) \multimap \overline{C, F}^{\,\tau}$ in $\Sigma_{I(G)}$, we can build a new term $G_{\langle A_B^C, D_E^F \rangle}\, \mathrm{T}_1\, (\lambda y.\mathrm{T}_2) :$ $\overline{C, F}^{\,\tau} \in \Lambda(\Sigma_{I(G)})$. It is then up to the lexicons to provide the good realizations of $G_{\langle A_B^C, D_E^F \rangle}$ so that if $\mathscr{L}_{\mathrm{Syn}}(\mathrm{T}_1) = a$, $\mathscr{L}_{\mathrm{Log}}(\mathrm{T}_1) = d$, $\mathscr{L}_{\mathrm{Syn}}(\mathrm{T}_2) = b$ and $\mathscr{L}_{\mathrm{Log}}(\mathrm{T}_2) = e$ then $\mathscr{L}_{\mathrm{Syn}}(G_{\langle A_B^C, D_E^F \rangle}\, \mathrm{T}_1\, (\lambda y.\mathrm{T}_2)) = a\,(\lambda y.b)$ and $\mathscr{L}_{\mathrm{Log}}(G_{\langle A_B^C, D_E^F \rangle}\, \mathrm{T}_1\, (\lambda y.\mathrm{T}_2)) = d\,(\lambda y.e)$. This is realized when the following equalities hold: $\mathscr{L}_{\mathrm{Syn}}(G_{\langle A_B^C, D_E^F \rangle}) = \mathscr{L}_{\mathrm{Log}}(G_{\langle A_B^C, D_E^F \rangle}) = \lambda Q\, R.Q\, R$. A CVG derivation using the (not in-situ) topicalization lexical item and the G rule from $\vdash \big[\mathsf{Sandy}\, \mathsf{top}^{\,\mathrm{a}}\big], \mathsf{top'}\, \mathsf{Sandy'} : \mathsf{NP}_\mathsf{S}^t, e_\pi^\pi \dashv$ and from $t, x : \mathsf{NP}, e \vdash$ $\big[^{\mathrm{s}}\mathsf{Chris}\, \big[\mathsf{liked}\, t^{\,\mathrm{c}}\big]\big], \mathsf{like'}\, x\, \mathsf{Chris'} : \mathsf{S}, \pi \dashv$ would result (conclusion of a G rule) in a proof of the following CVG sequent: $\vdash \big[\mathsf{Sandy}\, \mathsf{top}^{\,\mathrm{a}}\big]\big[^{\mathrm{s}}\mathsf{Chris}\, \big[\mathsf{liked}\, t^{\,\mathrm{c}}\big]\big], (\mathsf{top'}\, \mathsf{Sandy'})_x\, (\mathsf{like'}\, x\, \mathsf{Chris'}) : t, \pi \dashv$, the latter being isomorphic to the derivation in $\Lambda(\Sigma_{I(G)})$ proving:

$\vdash_{\Sigma_{I(G)}} G_{\langle \mathsf{NP}_\mathsf{S}^t, e_\pi^\pi \rangle}(\textsc{top}\ \textsc{sandy})(\lambda x.\textsc{liked}\, x\, \textsc{chris}) : \langle t, \pi \rangle$. Let's call this term T. Then with $\mathscr{L}_{\mathrm{Syn}}(\textsc{top}) = \lambda x.\big[\mathsf{top}\, x^{\,\mathrm{a}}\big] : [\![\mathsf{NP} \multimap_\mathrm{a} \mathsf{NP}_\mathsf{S}^t]\!] = \mathsf{NP} \multimap (\mathsf{NP} \multimap \mathsf{S}) \multimap t$, $\mathscr{L}_{\mathrm{Log}}(\textsc{top}) = \mathsf{top'} : [\![e \multimap e_\pi^\pi]\!] = e \multimap (e \multimap \pi) \multimap \pi$, and $\mathscr{L}_{\mathrm{Syn}}(G_{\langle \mathsf{NP}_\mathsf{S}^t, e_\pi^\pi \rangle}) = \mathscr{L}_{\mathrm{Log}}(G_{\langle \mathsf{NP}_\mathsf{S}^t, e_\pi^\pi \rangle}) = \lambda P\, Q.P\, Q$, we have the expected result:

$$\begin{aligned}
\mathscr{L}_{\mathrm{Syn}}(t) &= \big[\mathsf{Sandy}\, \mathsf{top}^{\,\mathrm{a}}\big](\lambda x.\big[^{\mathrm{s}}\mathsf{Chris}\, \big[\mathsf{liked}\, x^{\,\mathrm{c}}\big]\big]) \\
\mathscr{L}_{\mathrm{Log}}(t) &= (\mathsf{top'}\, \mathsf{Sandy'})(\lambda x.\mathsf{like'}\, x\, \mathsf{Chris'})
\end{aligned}$$

The C and R Rules. Section 4.2 shows how we can get rid of the C and R rules in CVG derivations. It brings into play an additional Shift rule and an additional operator S. It should be clear from the previous section that we could add an abstract constant corresponding to this Shift rule. The main point is that its realization in the syntactic calculus by $\mathscr{L}_{\mathrm{Syn}}$ should be $\mathsf{S} = \lambda e\, P.P\, e$ and its realization in the semantics by $\mathscr{L}_{\mathrm{Log}}$ should be the identity.

Technically, it would amount to have a new constant $\mathsf{S}_{\langle A, B_C^D \rangle} : \langle a, B_C^D \rangle \multimap \langle A_E^E, B_C^D \rangle$ such that $\mathscr{L}_{\mathrm{Log}}(\mathsf{S}_{\langle A, B_C^D \rangle}) = \lambda x.x : [\![B_C^D]\!] \multimap [\![B_C^D]\!]$ (this rule does not change the semantics) and $\mathscr{L}_{\mathrm{Syn}}(\mathsf{S}_{\langle A, B_C^D \rangle}) = \lambda x\, P.P\, x : [\![A]\!] \multimap ([\![A]\!] \multimap [\![E]\!]) \multimap [\![E]\!]$ (this rule shift the syntactic type). But since this Shift rule is meant to occur together with a G rule to model C and R, the kind of term we will actually consider is: $t = G_{\langle A_E^E, B_C^D \rangle}(\mathsf{S}_{\langle A, B_C^D \rangle}\, x)\, Q$ for some $x : \langle A, B_C^D \rangle$ and $Q : \langle A_E^E E, B_C^D \rangle$. And the interpretations of t in the syntactic and in the semantic calculus are:

$$\begin{aligned}
\mathscr{L}_{\mathrm{Log}}(t) &= (\lambda P\, Q.P\, Q) & \mathscr{L}_{\mathrm{Syn}}(t) &= (\lambda P\, Q.P\, Q) \\
&\quad ((\lambda y.y)\mathscr{L}_{\mathrm{Log}}(x))\mathscr{L}_{\mathrm{Log}}(Q) & &\quad ((\lambda e P.P\, e)\mathscr{L}_{\mathrm{Syn}}(x))\mathscr{L}_{\mathrm{Syn}}(Q) \\
&= \mathscr{L}_{\mathrm{Log}}(x)\, \mathscr{L}_{\mathrm{Log}}(Q) & &= \mathscr{L}_{\mathrm{Syn}}(Q)\, \mathscr{L}_{\mathrm{Syn}}(x)
\end{aligned}$$

So basically, $\mathscr{L}_{\mathrm{Log}}(\lambda x\, Q.t) = \mathscr{L}_{\mathrm{Log}}(G_{\langle A_E^E E, B_C^D \rangle})$, and this expresses that nothing new happens on the semantic side, while $\mathscr{L}_{\mathrm{Syn}}(\lambda x\, Q.t) = \lambda x\, Q.Q\, x$ expresses that, somehow, the application is reversed on the syntactic side.

Rather than adding these new constants S (for each type), we integrate their interpretation into the associated G constant[9]. This amounts to compiling the composition of the two terms. So if we have a pair of type A, B_C^D occurring in a CVG G, we add to $\Sigma_{I(G)}$ a new constant $G_{\langle A, B_C^D \rangle}^\mathsf{S} : \langle A, B_C^D \rangle \multimap (\overline{\langle A, B \rangle}^{\,\tau} \multimap \overline{\langle E, C \rangle}^{\,\tau}) \multimap \overline{\langle E, D \rangle}^{\,\tau}$ (basically the above term t) whose interpretations are: $\mathscr{L}_{\mathrm{Syn}}(G_{\langle A, B_C^D \rangle}^\mathsf{S}) = \lambda P\, Q.Q\, P$ and $\mathscr{L}_{\mathrm{Syn}}(G_{\langle A, B_C^D \rangle}^\mathsf{S}) = \lambda P\, Q.P\, Q$.

---

[9]It correspond to the requirement that the Shift rule occurs just before the G rule in the modeling the interface C and R rule with the the the G rule.

For instance, if we now use the in-situ topicalizer of Table [4.1] (triggered by stress for instance), from $\vdash S_{\mathsf{S}}\left[\mathsf{Sandy\ top_{\text{in-situ}}}^{\ \mathsf{a}}\right], \mathsf{top'\ Sandy'} : \mathsf{NP}_{\mathsf{S}}^{\mathsf{S}}, e_{\pi}^{\pi} \dashv$ and $t, x : \mathsf{NP}, e \vdash \left[{}^{\mathsf{s}}\mathsf{Chris}\left[\mathsf{liked}\ t^{\ \mathsf{c}}\right]\right], \mathsf{like'}\ x\ \mathsf{Chris'} : \mathsf{S}, \pi \dashv$ we can derive, using the G rule:

$$\vdash (S_{\mathsf{S}}\left[\mathsf{Sandy\ top_{\text{in-situ}}}^{\ \mathsf{a}}\right])_{t}\left[{}^{\mathsf{s}}\mathsf{Chris}\left[\mathsf{liked}\ t^{\ \mathsf{c}}\right]\right], (\mathsf{top'\ Sandy'})_{x}(\mathsf{like'}\ x\ \mathsf{Chris'}) : \mathsf{S}, \pi \dashv$$

Note that:

$$(S_{\mathsf{S}}\left[\mathsf{Sandy\ top_{\text{in-situ}}}^{\ \mathsf{a}}\right])_{t}(\left[{}^{\mathsf{s}}\mathsf{Chris}\left[\mathsf{liked}\ t^{\ \mathsf{c}}\right]\right]) = ((\lambda e\,P.P\,e)\left[\mathsf{Sandy\ top_{\text{in-situ}}}^{\ \mathsf{a}}\right])$$
$$(\lambda t.\left[{}^{\mathsf{s}}\mathsf{Chris}\left[\mathsf{liked}\ t^{\ \mathsf{c}}\right]\right])$$
$$=_{\beta} \left[{}^{\mathsf{s}}\mathsf{Chris}\left[\mathsf{liked}\left[\mathsf{Sandy\ top_{\text{in-situ}}}^{\ \mathsf{a}}\right]^{\mathsf{c}}\right]\right]$$

In order to map this derivation to an ACG term, we use the constant $\text{TOP}_{\text{IN-SITU}} : \langle\mathsf{NP}, e\rangle \multimap \langle\mathsf{NP}, e_{\pi}^{\pi}\rangle$ and the constant that will simulate the G rule and the Shift rule together $G_{\langle\mathsf{NP}, e_{\pi}^{\pi}\rangle}^{\mathsf{S}} : \langle\mathsf{NP}, e_{\pi}^{\pi}\rangle \multimap (\langle\mathsf{NP}, e\rangle \multimap \langle\mathsf{S}, \pi\rangle) \multimap \langle\mathsf{S}, \pi\rangle$ such that, according to what precedes: $\mathscr{L}_{\mathrm{Syn}}(G_{\langle\mathsf{NP}, e_{\pi}^{\pi}\rangle}^{\mathsf{S}}) = \lambda P\,Q.Q\,P$ and $\mathscr{L}_{\mathrm{Log}}(G_{\langle\mathsf{NP}, e_{\pi}^{\pi}\rangle}^{\mathsf{S}}) = \lambda P\,Q.P\,Q$. Then the previous CVG derivation corresponds to the following term of $\Lambda(\Sigma_{I(G)})$: $t = G_{\langle\mathsf{NP}, e_{\pi}^{\pi}\rangle}^{\mathsf{S}}(\text{TOP}_{\text{IN-SITU}}\ \text{SANDY})(\lambda x.\ldots$ and its expected realizations as syntactic and semantic terms are:

$$
\begin{aligned}
\mathscr{L}_{\mathrm{Syn}}(t) &= (\lambda P\,Q.Q\,P)(\left[\mathsf{Sandy\ top_{\text{in-situ}}}^{\ \mathsf{a}}\right]) & \mathscr{L}_{\mathrm{Log}}(t) &= (\lambda P\,Q.P\,Q)(\mathsf{top'\ Sandy'}) \\
&\quad(\lambda x.\left[{}^{\mathsf{s}}\mathsf{Chris}\left[\mathsf{liked}\ x^{\ \mathsf{c}}\right]\right]) & &\quad(\lambda x.\mathsf{like'}\ x\ \mathsf{Chris'}) \\
&= \left[{}^{\mathsf{s}}\mathsf{Chris}\left[\mathsf{liked}\left[\mathsf{Sandy\ top_{\text{in-situ}}}^{\ \mathsf{a}}\right]^{\mathsf{c}}\right]\right] & &= (\mathsf{top'\ Sandy'})(\lambda x.\mathsf{like'}\ x\ \mathsf{Chris'})
\end{aligned}
$$

Finally the $G_{\langle\alpha,\beta\rangle}$ and $G_{\langle\alpha,\beta\rangle}^{\mathsf{S}}$ are the only constants of the abstract signature having higher-order types. Hence, they are the only ones that will possibly trigger abstractions, fulfilling the CVG requirement.

When used in quantifier modeling, ambiguities are dealt with in CVG by the non determinism of the order in which semantic operators are retrieved from the store. It corresponds to the (reverse) order in which their ACG encoding are applied in the final term. However, by themselves, both accounts don't provide control on this order. Hence, when several quantifiers occur in the same sentence, all the relative orders of the quantifiers are possible.

Conclusion

We have shown how to encode a linguistically motivated *parallel* formalism, CVG, into a framework, ACG, that has mainly been used to encode syntactocentric formalisms until now. In addition to providing a logical basis for the CVG store mechanism, this encoding also sheds light on the various components (such as higher-order signatures) that are used in the interface calculus. It is noteworthy that the signature used to generate the interface proof terms relate to what is usually called *syntax* in mainstream categorial grammar, whereas the CVG *simple syntax* calculus is not expressed in such frameworks (while it can be using ACG, see Pogodalla, 2007b).

CVG Related Definitions and Properties

A.1 The CVG calculi

A.1.1 The CVG syntactic calculus

$$\frac{}{\vdash a : A} \text{ Lex} \qquad\qquad \frac{}{t : A \vdash t : A} \text{ T } (t \text{ fresh})$$

$$\frac{\Gamma \vdash b : A \multimap_{\mathrm{s}} B \qquad \Delta \vdash a : A}{\Gamma, \Delta \vdash \left\lceil {}^{\mathrm{s}}a\; b \right\rceil : B} \text{ M}_s \qquad \frac{\Gamma \vdash b : A \multimap_{\mathrm{c}} B \qquad \Delta \vdash a : A}{\Gamma, \Delta \vdash \left\lceil b\; a^{\,\mathrm{c}} \right\rceil : B} \text{ M}_c$$

$$\frac{\Gamma \vdash b : A \multimap_{\mathrm{a}} B \qquad \Delta \vdash a : A}{\Gamma, \Delta \vdash \left\lceil b\; a^{\,\mathrm{a}} \right\rceil : B} \text{ M}_a$$

$$\frac{\Gamma \vdash a : A_B^C \qquad t : A; \Gamma' \vdash b : B}{\Gamma; \Gamma' \vdash a_t\, b : C} \text{ G}$$

A.1.2 The CVG semantic calculus

$$\frac{}{\vdash a : A \dashv} \text{ Lex} \qquad\qquad \frac{}{x : B \vdash x : B \dashv} \text{ T } (x \text{ fresh})$$

$$\frac{\vdash f : A \multimap B \dashv \Delta \qquad \vdash a : A \dashv \Delta'}{\vdash (f\, a) : B \dashv \Delta; \Delta'} \text{ M}$$

$$\frac{\Gamma \vdash a : A_B^C \dashv \Delta \qquad x : A; \Gamma' \vdash b : B \dashv \Delta'}{\Gamma; \Gamma' \vdash a_x\, b : C \dashv \Delta; \Delta'} \text{ G}$$

$$\frac{\vdash a : A_B^C \dashv \Delta}{\vdash x : A \dashv a_x : A_B^C; \Delta} \text{ C } (x \text{ fresh}) \qquad \frac{\vdash b : B \dashv a_x : A_B^C; \Delta}{\Gamma \vdash (a_x\, b) : C \dashv \Delta} \text{ R}$$

A.1.3   The CVG interface calculus

$$\frac{}{\vdash w, c : A, B \dashv} \text{ Lex} \qquad\qquad \frac{}{x, t : A, B \vdash x, t : A, B \dashv} \text{ T}$$

$$\frac{\Gamma \vdash f, v : A \multimap_{\mathrm{s}} B, C \multimap D \dashv \Delta \qquad \Gamma' \vdash a, c : A, C \dashv \Delta'}{\Gamma; \Gamma' \vdash \left[{}^{\mathrm{s}} a\; f\right], (v\, c) : B, D \dashv \Delta; \Delta'} \text{ M}_s$$

$$\frac{\Gamma \vdash f, v : A \multimap_{\mathrm{c}} B, C \multimap D \dashv \Delta \qquad \Gamma' \vdash a, c : A, C \dashv \Delta'}{\Gamma; \Gamma' \vdash \left[f\; a\; {}^{\mathrm{c}}\right], (v\, c) : B, C \dashv \Delta; \Delta'} \text{ M}_c$$

$$\frac{\Gamma \vdash f, v : A \multimap_{\mathrm{a}} B, C \multimap D \dashv \Delta \qquad \Gamma' \vdash a, c : A, C \dashv \Delta'}{\Gamma; \Gamma' \vdash \left[f\; a\; {}^{\mathrm{a}}\right], (v\, c) : B, C \dashv \Delta; \Delta'} \text{ M}_c$$

$$\frac{\Gamma \vdash a, d : A_B^C, D_E^F \dashv \Delta \qquad t, x : A, D; \Gamma' \vdash b, e : B, E \dashv \Delta'}{\Gamma; \Gamma' \vdash a_t\, b, d_x\, e : C, F \dashv \Delta; \Delta'} \text{ G}$$

$$\frac{\Gamma \vdash a, b : A, B_C^D \dashv \Delta}{\Gamma \vdash a, x : A, B \dashv b_x : B_C^D; \Delta} \text{ C } (x \text{ fresh}) \qquad \frac{\vdash e, c : E, C \dashv b_x : B_C^D; \Delta}{\Gamma \vdash e, (b_x\, c) : E, D \dashv \Delta} \text{ R}$$

Example of a simple interface derivation:

$$\frac{\begin{array}{c}\vdots\, \pi \\ \vdash \left[\text{liked Sandy}\,{}^{\mathrm{c}}\right], \text{like' Sandy'} : \mathsf{NP} \multimap_{\mathrm{s}} \mathsf{S}, e \multimap \pi \dashv \end{array} \qquad \dfrac{}{\vdash \text{Chris}, \text{Chris} : \mathsf{NP}, e \dashv} \text{ Lex}}{\vdash \left[{}^{\mathrm{s}}\text{Chris}\left[\text{liked Sandy}\,{}^{\mathrm{c}}\right]\right], \text{like' Sandy' Chris'} : \mathsf{S}, \pi \dashv} \text{ M}_s$$

$$\pi = \frac{\dfrac{}{\vdash \text{liked}, \text{like'} : \mathsf{NP} \multimap_{\mathrm{c}} \mathsf{NP} \multimap_{\mathrm{s}} \mathsf{S}, e \multimap e \multimap \pi \dashv} \text{ Lex} \qquad \dfrac{}{\vdash \text{Sandy}, \text{Sandy'} : \mathsf{NP}, e \dashv} \text{ Lex}}{\vdash \left[\text{liked Sandy}\,{}^{\mathrm{c}}\right], \text{like' Sandy'} : \mathsf{NP} \multimap_{\mathrm{s}} \mathsf{S}, e \multimap \pi \dashv} \text{ M}_c$$

Example using the G rule

$$\frac{\begin{array}{c}\vdots\, \pi_1 \\ \vdash \left[\text{Sandy top}\,{}^{\mathrm{a}}\right], \text{top' Sandy'} : \mathsf{NP}_{\mathsf{S}}^{t}, e_{\pi}^{\pi} \dashv \end{array} \qquad \begin{array}{c}\vdots\, \pi_2 \\ t, x : \mathsf{NP}, e \vdash \left[{}^{\mathrm{s}}\text{Chris}\left[\text{liked } t\,{}^{\mathrm{c}}\right]\right], \text{like' } x \text{ Chris'} : \mathsf{S}, \pi \dashv \end{array}}{\vdash \left[\text{Sandy top}\,{}^{\mathrm{a}}\right](\lambda t.\left[{}^{\mathrm{s}}\text{Chris}\left[\text{liked } t\,{}^{\mathrm{c}}\right]\right]), (\text{top' Sandy'})(\lambda x.\text{like' } x \text{ Chris'}) : t, \pi \dashv} \text{ G}$$

with trivial derivations for π_1 and π_2.

A.2   On CVG derivations

Proposition A.1. *Let π be a CVG semantic derivation. It can be turned into a CVG semantic derivation where all C and R pairs of rule have been replaced by the above schema, and which derives the same term.*

Proof. This is proved by induction on the derivations. If the derivation stops on a Lexicon, Trace, Modus Ponens, G or C rule, this is trivial by application of the induction hypothesis.

If the derivation stops on a R rule, the C and R pair has the above schema. Note that nothing can be erased from Γ in π_2 because every variable in Γ occur (freely) only in a and Δ. So using a G rule (the only one that can delete material from the left hand side of the sequent) would leave variables in the store that could not be bound later. The same kind of argument shows that nothing can be retrieved from Δ before a_x had been retrieved. This means that no R rule can occur in π_2 whose corresponding C rule is in π_1 (while there can be a R rule with a corresponding C rule introduced in π_2). Hence we can make the transform and apply the induction hypothesis to the two premises of the new G rule. $\qquad\square$

B | Appendix

How to build an applicative ACG

Let $\Sigma_{\mathrm{HO}} = \langle A_{\mathrm{HO}}, C_{\mathrm{HO}}, \tau_{\mathrm{HO}} \rangle$. This section section shows how to build an ACG $\mathscr{G} = \langle \Sigma_{\mathrm{2nd}}, \Sigma_{\mathrm{HO}}, \mathscr{L}, \mathsf{S}' \rangle$ such that $\mathcal{O}(\mathscr{G})$ is the set of $t : \mathsf{S} \in \Lambda_{\Sigma_{\mathrm{HO}}}$ such that there exists π a proof of $\vdash_{\Sigma_{\mathrm{HO}}} t : \mathsf{S}$ and π does not use the abstraction rule. This construction is very similar to the one given in Hinderer, 2008, Chap. 7.

Definition B.1. *Let α be a type. We inductively define the set $\mathsf{Decompose}(\alpha)$ as:*

- *if α is atomic, $\mathsf{Decompose}(\alpha) = \{\alpha\}$;*

- *if $\alpha = \alpha_1 \multimap \alpha_2$, $\mathsf{Decompose}(\alpha) = \{\alpha\} \cup \{\alpha_1\} \cup \mathsf{Decompose}(\alpha_2)$.*

Let T be a set of types. We then define:

- *$\mathsf{Base}(T) = \cup_{\alpha \in T} \mathsf{Decompose}(T)$;*

- *$\mathsf{At}(T)$ a set of fresh atomic types that is in a one to one correspondence with $\mathsf{Base}(T)$. We note $:=$ one of the correspondence from $\mathsf{At}(T)$ to $\mathsf{Base}(t)$ (we also note $:=$ its unique homomorphic extension that is compatible with $\multimap$. The later is not necessarily a bijection);*

- *let $\alpha \in \mathsf{Base}(T)$. The set $\mathsf{AtP}_T(\alpha)$ of its* atomic profiles *is inductively defined as:*

  - *if α is atomic, $\mathsf{AtP}_T(\alpha) = \{\alpha'\}$ such that α' is the unique element of $\mathsf{At}(T)$ and $\alpha' := \alpha$;*
  - *if $\alpha = \alpha_1 \multimap \alpha_2$, $\mathsf{AtP}_T(\alpha) = \{\alpha'\} \cup \{\alpha'_1 \multimap \alpha'_2 \,|\, \alpha'_2 \in \mathsf{AtP}_T(\alpha_2)\}$ where:*
    - *α' is uniquely defined in $\mathsf{At}(T)$ and $\alpha' := \alpha$;*
    - *α'_1 is uniquely defined in $\mathsf{At}(T)$ and $\alpha'_1 := \alpha_1$. There exists such an α'_1 because $\alpha_1 \in \mathsf{Decompose}(\alpha)$ and $\mathsf{Decompose}(\alpha) \subset \mathsf{Base}(T)$ when $\alpha \in \mathsf{Base}(T)$.*

    Note that for the same reason, α'_2 is well defined.

  Note that for any $\alpha \in \mathsf{Base}(T)$, The types in $\mathsf{AtP}_T(\alpha)$ are of order at most 2.

Proposition B.1. *Let T be a set of types and $\alpha \in \mathsf{Base}(T)$ with $\alpha = \alpha_1 \multimap \ldots \multimap \alpha_k \multimap \alpha_0$ such that α_0 is atomic. Then $|\mathsf{AtP}_T(\alpha)| = k + 1$.*

Proof. By induction. $\qquad\square$

Proposition B.2. *Let T be a set of types and $\alpha \in \mathsf{Base}(T)$. Then for all $\alpha' \in \mathsf{AtP}_T(\alpha)$ we have $\alpha' := \alpha$.*

Proof. By induction. $\qquad\square$

In the following, we always consider $T = \cup_{c \in C_{\mathrm{HO}}} \tau_{\mathrm{HO}}(c)$. We then can define $\Sigma_{\mathrm{2nd}} = \langle A_{\mathrm{2nd}}, C_{\mathrm{2nd}}, \tau_{\mathrm{2nd}} \rangle$ with:

- $A_{2nd} = \mathsf{At}(T)$

- $\mathsf{S}' \in A_{2nd}$ the unique term such that $\mathsf{S}' := \mathsf{S}$

- $C_{2nd} = \cup_{c \in C_{HO}}\{\langle c, \alpha' \rangle | \alpha' \in \mathsf{AtP}_T(\tau_{HO}(c))\}$ ($\mathsf{AtP}_T(\tau_{HO}(c))$ is well defined because $\tau_{HO}(c) \in \mathsf{Base}(T)$)

- for every $c' = \langle c, \alpha' \rangle \in C_{2nd}$, $\tau_{2nd}(c') = \alpha'$

Note that according to Proposition B.1, for every constant c of C_{HO} of arity k (i.e. $\tau_{HO}(c) = \alpha_1 \multimap \ldots \multimap \alpha_k \multimap \alpha_0$), there are $k+1$ constants in C_{2nd}.

Finally, in order to completely define $\mathscr{G}$, we need to define $\mathscr{L}$:

- for $\alpha' \in A_{2nd}$, there exists a unique $\alpha \in \mathsf{Base}(T)$ such that $\alpha' := \alpha$ by construction of $\mathsf{At}(T)$. We set $\mathscr{L}(\alpha') = \alpha$.

- for $c' = \langle c, \alpha' \rangle \in C_{2nd}$, we set $\mathscr{L}(c') = c$

According to Proposition B.2, we have $\mathscr{L}(\tau_{2nd}(c')) = \alpha$ where α is the type of $\mathscr{L}(c')$ so $\mathscr{L}$ is well defined.

Proposition B.3. *There exists* $t : \alpha \in \Lambda_{\Sigma_{HO}}$ *build using only applications if and only if there exists* $t' : \alpha'$ *a closed term of* $\Lambda_{\Sigma_{2nd}}$ *with* α' *the unique element of* $\mathsf{At}(T)$ *such that* $\alpha' := \alpha$ *and* $\mathscr{L}(t') = t$.

Proof. $\Rightarrow$ We prove it by induction on t. If t is a constant, we take $t' = \langle t, \alpha'$ with α' the unique element of $\mathsf{At}(T)$ such that $\alpha' := \alpha$. By definition, $\mathscr{L}(t') = t$.

If $t = c\, u_1 \ldots u_k$, then $c \in C_{HO}$ is of type $\alpha_1 \multimap \ldots \multimap \alpha_k \multimap \alpha$ and for all $i \in [1, k]$ u_k is of type α_i. We know there exist $c' = \langle c, \beta' \rangle \in \Sigma_{2nd}$ such that $\beta' = \alpha'_1 \multimap \ldots \alpha'_k \multimap \alpha'$ with for all $i \in [1, k]$, α'_i is the unique element of $\mathsf{At}(T)$ such that $\alpha'_i := \alpha_i$ and α' the unique element of $\mathsf{At}(T)$ such that $\alpha' := \alpha$. By induction hypothesis, we also have for all $i \in [1, k]$ a term $u'_i : \alpha'_i$ with α'_i the unique element of $\mathsf{At}(T)$ such that $\alpha'_i := \alpha_i$ and $\mathscr{L}(u'_i) = u_i$.

If we take $t' = \langle c, \beta' \rangle u'_1 \ldots u'_k$, we have $\mathscr{L}(t') = \mathscr{L}(\langle c, \beta' \rangle u'_1 \ldots u'_k) = \mathscr{L}(\langle c, \beta' \rangle)\, \mathscr{L}(u'_1) \ldots \mathscr{L}(u'_k) = c\, u_1 \ldots u_k = t$ which completes the proof.

$\Leftarrow$ If $\alpha' \in \mathsf{At}(T)$ and t' is a closed term then because Σ_{2nd} is of order 2, then t' is build only using applications. Hence its image by $\mathscr{L}$ is also only build using applications.

$\square$

Bibliography

Abeillé, A. (2002). *Une grammaire électronique du fran cais.* CNRS Éditions.

Asudeh, A. and Crouch, R. (2001). Glue for HPSG. *Proceedings 8th Int. Conference on Head-Driven Phrase Structure Grammar.*

Bach, E. and Partee, B. H. (1980). Anaphora and Semantic Structure. *Proceedings the Chicago Linguistic Society, l6: Parasession on Anaphora.* Reprinted in Barbara H. Partee, Compositionality in Formal Semantics (Blackwell), pp. 122-152.

Blackburn, P. and Bos, J. (2005). *Representation and Inference for Natural Language. A First Course in Computational Semantics.* CSLI.

Bos, J. (1995). Predicate Logic Unplugged. *Proceedings of the Tenth Amsterdam Colloquium.*

Carpenter, B. (1997). *Type-Logical Semantics.* The MIT Press.

Champollion, L. (2007). Lexicalized non-local MCTAG with dominance links is NP-complete. *In* Penn, G. and Stabler, E., (eds.), *Proceedings of Mathematics of Language (MOL) 10.* http://www.ling.upenn.edu/~champoll/npcomplete-mctag-proceedings.pdf.

Cooper, R. (1975). Montague's Semantic Theory and Transformational Syntax. PhD thesis. University of Massachusetts at Amherst.

Cooper, R. (1983). *Quantification and Syntactic Theory.* Reidel, Dordrecht.

Copestake, A. et al. (2005). Minimal Recursion Semantics: An introduction. *Journal of Research on Language and Computation* **3**, pp. 281–332.

Culicover, P. W. and Jackendoff, R. (2005). *Simpler Syntax.* Oxford University Press.

Curry, H. B. (1961). Some Logical Aspects of Grammatical Structure. *In* Jakobson, R., (ed.), *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics.* American Mathematical Society, pp. 56–68.

Dalrymple, M. (ed.). *Semantics and Syntax in Lexical Functional Grammar: The Resource Logic Approach.* MIT Press.

de Groote, P. (2001). Towards Abstract Categorial Grammars. *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference,* pp. 148–155.

de Groote, P. (2002). Tree-Adjoining Grammars as Abstract Categorial Grammars. *TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks.* http://www.loria.fr/equipes/calligramme/acg/publications/2002-tag+6.pdf. Università di Venezia, pp. 145–150.

de Groote, P. (2006). Towards a Montagovian account of dynamics. *In* Gibson, M. and Howell, J., (eds.), *Proceedings of Semantics and Linguistic Theory (SALT) 16.* http://elanguage.net/journals/index.php/salt/article/view/16.1/1791.

de Groote, P. and Maarek, S. (2007). Type-theoretic extensions of Abstract Categorial Grammars. *New Directions in Type-Theoretic Grammars, proceedings of the workshop.* http://let.uvt.nl/general/people/rmuskens/ndttg/ndttg2007.pdf, pp. 18–30.

de Groote, P. and Pogodalla, S. (2003). *m*-Linear Context-Free Rewriting Systems as Abstract Categorial Grammars. *In* Oehrle, R. and Rogers, J., (eds.), *MOL 8, proceedings of the eighth Mathematics of Language Conference.*

de Groote, P. and Pogodalla, S. (2004). On the expressive power of Abstract Categorial Grammars: Representing context-free formalisms. *Journal of Logic, Language and Information* **13**(4), pp. 421–438. DOI: 10.1007/s10849-004-2114-x. URL: http://hal.inria.fr/inria-00112956.

de Groote, P., Pogodalla, S., and Pollard, C. (2009). On the Syntax-Semantics Interface: From Convergent Grammar to Abstract Categorial Grammar. *In* Kanazawa, M., Ono, H., and de Queiroz, R., (eds.), *16th Workshop on Logic, Language, Information and Computation.* Vol. 5514. LNAI/FoLLI. Springer, pp. 182–196. DOI: 10.1007/978-3-642-02261-6_15. URL: http://hal.inria.fr/inria-00390490.

Egg, M., Koller, A., and Niehren, J. (2001). The Constraint Language for Lambda Structures. *Journal of Logic, Language, and Information* **10**, pp. 457–485.

Frank, A. and van Genabith, J. (2001). Glue Tag: Linear Logic based Semantics Construction for LTAG - and What it Teaches Us About the Relation Between LFG and LTAG. *In* Butt, M. and King, T. H., (eds.), *Proceedings of the LFG '01 Conference.* Online Proceedings. http://cslipublications.stanford.edu/LFG/6/lfg01.html. CSLI Publications.

Gardent, C. and Kallmeyer, L. (2003). Semantic construction in Feature-Based TAG. *Proceedings of the 10th Meeting of the European Chapter of the Association for Computational Linguistics (EACL).*

Gazdar, G. (1981). Unbounded dependencies and coordinate structure. *Linguistic Inquiry* **12**, pp. 155–184.

Hinderer, S. (2008). Automatisation de la construction sémantique dans TYn. PhD thesis. Université Henri Poincaré – Nancy 1.

Jackendoff, R. (2002). *Foundations of Language: Brain, Meaning, Grammar, Evolution.* Oxford University Press.

Joshi, A. K. and Schabes, Y. (1997). *Tree-adjoining grammars. In Handbook of formal languages.* Ed. by G. Rozenberg and A. Salomaa. Vol. 3. Springer. Chap. 2.

Joshi, A. K. (1988). An Introduction to Tree Adjoining Grammars. *In* Manaster-Ramer, A., (ed.), *Mathematics of Language, Proceedings of a conference held at the University of Michigan, Ann Arbor, October 1984.* Amsterdam: John Benjamins.

Joshi, A. K., Levy, L. S., and Takahashi, M. (1975). Tree Adjunct Grammars. *Journal of Computer and System Sciences* **10**(1), pp. 136–163.

Kallmeyer, L. (2002). Using an Enriched TAG Derivation Structure as Basis for Semantics. *Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+6).*

Kanazawa, M. (2007). Parsing and Generation as Datalog Queries. *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL).* http://www.aclweb.org/anthology/P/P07/P07-1023. Prague, Czech Republic: Association for Computational Linguistics, pp. 176–183.

Lambek, J. (1958). The Mathematics of Sentence Structure. *American Mathematical Monthly* **65**(3), pp. 154–170.

Montague, R. (1973). The Proper Treatment of Quantification in Ordinary English. *In* Hintikka, J., Moravcsik, J., and Suppes, P., (eds.), *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics.* Reprinted in *Formal Philosophy*, by Richard Montague, Yale University Press, New Haven, CT, 1974, pp. 247–270. Dordrecht, Holland: D. Reidel Publishing Co., pp. 221–242.

Montague, R. (1974). *The Proper Treatment of Quantification in Ordinary English. In Formal Philosophy: Selected Papers of Richard Montague.* Re-edited in "Formal Semantics: The Essential Readings", Paul Portner and Barbara H. Partee, editors. Blackwell Publishers, 2002. Yale University Press.

Moortgat, M. (1991). *Generalized quantifiers and discontinuous type constructors. In Discontinuous constituency.* Ed. by W. Sijtsma and A. van Horck. De Gruyter.

Moot, R. and Piazza, M. (2001). Linguistic Applications of First Order Intuitionistic Linear Logic. *Journal of Logic, Language and Information* **10**, pp. 211–232.

Morrill, G. V. (1994). *Type Logical Grammar Categorial Logic of Signs.* Kluwer Academin Publishers.

Muskens, R. (1995). *Order-Independence and Underspecification. In Ellipsis, Underspecification, Events and More in Dynamic Semantics.* Ed. by J. Groenendijk. DYANA Deliverable R.2.2.C, pp. 15–34.

Pogodalla, S. (2004). Computing Semantic Representation: Towards ACG Abstract Terms as Derivation Trees. *Seventh International Workshop on Tree Adjoining Grammar and Related Formalisms - TAG+7*, pp. 64–71. URL: http://hal.inria.fr/inria-00107768.

Pogodalla, S. (2007a). Ambiguïté de portée et approche fonctionnelle des TAG. *Traitement Automatique des Langues Naturelles - TALN 2007*, 10 p. URL: http://hal.inria.fr/inria-00141913.

Pogodalla, S. (2007b). Generalizing a Proof-Theoretic Account of Scope Ambiguity. *7th International Workshop on Computational Semantics - IWCS-7.* URL: http://hal.inria.fr/inria-00112898.

Pogodalla, S., Quatrini, M., and Retoré, C. (2011). *Logic and Grammar. Essays Dedicated to Alain Lecomte on the Occasion of His 60th Birthday.* Ed. by S. Pogodalla, M. Quatrini, and C. Retoré. Vol. 6700. Lecture Notes in Computer Science. Springer, p. 159. DOI: 10.1007/978-3-642-21490-5. URL: http://hal.inria.fr/inria-00607880.

Pollard, C. (2011a). *Covert Movement in Logical Grammar. In Logic and Grammars: Essays Dedicated to Alain Lecomte on the Occasion of His 60th Birthday.* Ed. by S. Pogodalla, M. Quatrini, and C. Retoré. Vol. 6700. LNCS/LNAI. Springer, pp. 17–40. DOI: 10.1007/978-3-642-21490-5_2. URL: http://hal.inria.fr/inria-00607880.

Pollard, C. (2011b). *The Calculus of Responsibility and Commitment. In Ludics, Dialogue and Interaction. PRELUDE Project — 2006–2009. Revised Selected Papers.* Ed. by A. Lecomte and S. Troçon. Vol. 6505. LNCS. Springer, pp. 174–187. DOI: 10.1007/978-3-642-19211-1_11.

Pollard, C. and Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar.* Distributed by University of Chicago Press. Stanford, CA: CSLI Publications.

Rambow, O. (1994). Formal and Computational Aspects of Natural Language Syntax. PhD thesis. Philadelphia, PA,USA: University of Pennsylvania.

Rambow, O. and Satta, G. (1992). Formal Properties of Non-Locality. *Proceedings of the 1st International Workshop on Tree Adjoining Grammrs.*

Rogers, J. and Vijay-Shanker, K. (1992). Reasoning with Descriptions of Trees. *Meeting of the Association for Computational Linguistics*, pp. 72–80.

Salvati, S. (2005). Problèmes de filtrage et problèmes d'analyse pour les grammaires catégorielles abstraites. PhD thesis. Institut National Polytechnique de Lorraine.

Salvati, S. (2006). Syntactic Descriptions: A Type System for Solving Matching Equations in the Linear λ-Calculus. *In* Pfenning, F., (ed.), *Proceedings of RTA 2006.* Vol. 4098. Lecture Notes in Computer Science. DOI: 10.1007/11805618_12.

Schuler, W., Chiang, D., and Dras, M. (2000). Multi-component TAG and notions of formal power. *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics.* Hong Kong: Association for Computational Linguistics, pp. 448–455. DOI: http://dx.doi.org/10.3115/1075218.1075275.

Vijay-Shanker, K. (1992). Using Descriptions of Trees in a Tree Adjoining Grammar. *Computational Linguistics* **18**(4), pp. 481–518.

Weir, D. J. (1988). Characterizing Mildly Context-Sensitive Grammar Formalisms. PhD thesis. University of Pennsylvania.

XTAG Research Group (2001). *A Lexicalized Tree Adjoining Grammar for English.* Tech. rep. IRCS-01-03. IRCS, University of Pennsylvania.

Yoshinaka, R. (2006). Linearization of Affine Abstract Categorial Grammars. *proceedings of Fromal Grammar 2006.*