A syntax-semantics interface for Tree-Adjoining Grammars through Abstract Categorial Grammars
Sylvain Pogodalla

To cite this version:

HAL Id: hal-01242154
https://inria.hal.science/hal-01242154v5
Submitted on 5 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A syntax-semantics interface for Tree-Adjoining Grammars through Abstract Categorial Grammars

Sylvain Pogodalla
INRIA, Villers-lès-Nancy, France
Université de Lorraine, LORIA, Vandœuvre-lès-Nancy, France
CNRS, LORIA, Vandœuvre-lès-Nancy, France

ABSTRACT
We present a model of the syntax-semantics interface for Tree-Adjoining Grammars (TAGs). It is based on the encoding of TAGs within the framework of Abstract Categorial Grammars (ACGs). This encoding relies on a functional interpretation of the substitution and adjunction operations of TAGs. In ACGs, the abstract terms representing derivation trees are full-fledged objects of the grammar. These terms are mapped onto logical formulas representing the semantic interpretation of natural language expressions that TAGs can analyze. Because of the reversibility properties of ACGs, this provides a way to parse and generate with the same TAG-encoded grammar. We propose several analyses, including for long-distance dependencies, quantification, control and raising verbs, and subordinate clauses. We also show how this encoding easily extends to other phenomena such as idioms or scope ambiguities. All the lexical data for these modellings are provided and can be run with the ACG toolkit, a software package dedicated to the development of ACGs that can use these grammars both for parsing and generation.

1 MOTIVATIONS
1.1 Tree-Adjoining Grammar and semantic representation
The Tree-Adjoining Grammar (TAG) formalism (Joshi et al. 1975; Joshi and Schabes 1997) is a formalism dedicated to the modelling of
natural languages. As the name indicates, the primary objects it considers are trees rather than strings, contrary to, for instance, context-free grammars. As such, the object language a TAG generates is a tree language, the language of the derived trees. These trees result from the application of two operations, substitution and adjunction, to a set of generators: the elementary trees. The substitution operation consists in replacing one leaf of a tree by another tree, while the adjunction operation consists in inserting a tree into another one by replacing an internal node with a whole tree. A sequence of such operations and the elementary trees they operate on can be recorded as a derivation tree. Reading the leaves of the derived tree, or computing the yield, produces the associated generated string language.

The class of the generated string languages strictly includes the one generated by context-free grammars. This property, together with other ones such as the crucial polynomial parsing property, plays an important role in the characterization of the expressive power that natural language modelling requires. Joshi (1985) proposed to call the class of languages (resp. grammars) necessary for describing natural languages the class of mildly context-sensitive languages or mCSL (resp. mildly context-sensitive grammars or mCSG). These formal and computational properties have been extensively studied and provide TAG with appealing features for natural language processing. In addition to its formal properties, TAG has also been studied both from the perspective of fine-grained modellings of a wide range of linguistic phenomena, and from the perspective of large coverage. Large scale grammars have been developed for several languages, including English (XTAG Research Group 2001) and French (Abeillé 2002; Crabbé 2005; de La Clergerie 2005). In addition to these hand-crafted grammars, automatic extraction of TAGs has also been proposed (Xia et al. 2000; Xia 2001; Chen et al. 2006).

Another key feature that makes TAG relevant to natural language modelling lies in the capability of its elementary trees to locally specify (syntactic and semantic) dependencies between parts that can occur arbitrarily far from each other at the surface level at the end of a

---

derivation. This property to locally state, within the elementary trees, dependency constraints is also known as the extended domain of locality (Joshi 1994). Thanks to the adjunction operation, a dependency described locally in an elementary tree can end as a long-distance dependency in the resulting derived tree. The relevant structure to store the relations between the elementary trees that are used in a derivation is then the derivation tree. This makes the latter structure appropriate to derive semantic representations for TAGs.

It was however noticed that derivation trees do not directly express the semantic dependencies, and that they seem to lack some structural information (Vijay-Shanker 1992; Candito and Kahane 1998). To overcome this problem, several approaches have been proposed. Some rely on extensions of the TAG formalism (Rambow et al. 1995, 2001); some others revisit the derivation tree definition in order to allow for recovering all the semantic dependency relations (Schabes and Shieber 1994; Shieber 1994; Kallmeyer 2002; Joshi et al. 2003; Kallmeyer and Joshi 2003). However, solutions to the problem strictly relying on derivation trees have also been proposed. They make use of unification (Kallmeyer and Romero 2004, 2008), functional tree interpretation (Pogodalla 2004a, 2009), synchronous grammars (Nesson and Shieber 2006; Nesson 2009), or tree transduction (Shieber 2006; Kallmeyer and Kuhlmann 2012; Shieber 2014).

1.2 TAG and Abstract Categorial Grammars: our approach

In this article, we elaborate on Pogodalla (2004a, 2009) in order to propose a syntax-semantics interface and a semantic construction process for TAGs. We base our analysis on the framework of Abstract Categorial Grammars (ACGs: de Groote 2001). ACGs derive from type-theoretic grammars in the tradition of Lambek (1958), Curry (1961), and Montague (1973). They can be considered as a framework in which several grammatical formalisms may be encoded (de Groote and Pogodalla 2004), in particular TAGs (de Groote 2002). The definition of an ACG is based on a small set of mathematical primitives from type-theory, $\lambda$-calculus, and linear logic. These primitives combine via simple composition rules, offering ACGs a good flexibility. In particular, ACGs generate languages of linear $\lambda$-terms, which generalize both string and tree languages.
But ACGs are not restricted to languages of \( \lambda \)-terms encoding strings or trees. They can express logic-based semantic representation languages. And moving from one kind to another kind of language is realized by composing ACGs. We take advantage of the different composition modes to control the admissible derivation structures on the one hand, and to model the syntax-semantics interface on the other hand.

The core contribution of this article is to show that ACGs offer a suitable model of the syntax-semantics interface for TAG. By construction, this model is fully compositional and satisfies the homomorphic requirement between parse structures (terms representing derivation trees) and semantic terms. It relies on an encoding of TAGs into ACGs. For a given TAG \( G \), with this encoding, we can construct and relate several ACGs that generate the same string language, derived tree language, and derivation tree language as \( G \). By ACG composition, this encoding is the same as the one proposed by de Groote (2002) (that only addresses the syntactic encoding of TAG into ACG, not the syntax-semantics interface), which ensures the correctness of the (syntactic) encoding. This encoding corresponds to the path with solid lines from TAG derivation trees to Strings in Figure 1. But we introduce an intermediate level, of generalized derivations, on which we base our syntax-semantics interface (the dashed lines in Figure 1). Doing so, we separate the level required for transferring the syntactic structures into semantics, and vice-versa, from the level that controls those structures so that only the ones that TAG considers to be admissible (i.e., TAG derivations) are kept. We show that this level allows us to account for the semantics of long-distance dependencies, quantification, separate modification without multiple adjunction, control verbs, raising verbs, etc. Moreover, this is done in a principled way, following the standard homomorphism between the syntactic and semantic categories of Montague (1973).

Contrary to Pogodalla (2004a) and Kallmeyer and Romero (2004, 2008), and similarly to synchronous TAG analyses (Nesson and Shieber 2006; Nesson 2009), the semantic modelling we propose does not rely on an intermediate underspecification language. We show instead that this is not required in order to model long-distance dependencies, raising verbs, or quantification. We also introduce and precisely describe the syntax-semantics modelling for adjectives (with-
out multiple adjunctions), control verbs, and subordinate clauses. We also discuss the encoding of features. While these modellings can essentially be rephrased in synchronous TAG (and vice-versa: the solid lines of Figure 1 also correspond to the synchronous TAG architecture for the syntax-semantics interface), it is not the case for some other ones, and we show how the approach easily extends, without requiring the design of new parsing algorithms, to other phenomena such as idioms or subordinate clauses, for which we propose a novel modelling. Other TAG extensions such as the cosubstitution operation proposed by Barker (2010) to model scope ambiguities also easily fall within the scope of our approach and can be given a type-raising account. In particular, this account exemplifies how to model the non-functional nature of the form-meaning relation.

Finally, except for the type-raising account of quantification, the ACG model for the syntax-semantics interface of TAG that we propose belongs to the class of second-order ACGs. This class has the property that whatever the language we parse (strings, trees, or any kind of terms, such as first-order or higher-order logical formulas), parsing is polynomial. This parsing is implemented in the ACG toolkit. Consequently, there is a parser that can actually recover the TAG derivation structure (if any) of some string, or of some derived tree, and interpret it as a logical formula, or that can actually recover the TAG derivation structure (if any) of some logical formula and interpret it as a derived

---

2 Synchronous TAG analyses often hinge on Multi-Component TAG (MCTAG: Weir 1988), which is beyond the scope of this article. But we do not consider this to be an essential difference, since this can be integrated into the ACG approach as well (Pogodalla 2009). We discuss the differences between the approaches in Section 9.

3 This encoding is due to Kobele (2012).

4 The toolkit is available at http://acg.loria.fr.
The ACG framework is inherently reversible (Dymetman 1994), and parsing and generation of second-order ACGs are performed in polynomial time, including for the modellings that go beyond TAG (except the type-raising account of quantification), without having to design new parsers. Note, however, that we do not yet address the problem of logical-form equivalence (Shieber 1993) which states that, even if two formulas are logically equivalent, it might be possible to recover a derivation structure for one but not for the other.

We also validated the modellings and the lexicons we provide in this article, both for parsing and generation, by implementing in the ACG toolkit all the examples of this article. This results in a toy grammar (corresponding to about forty elementary trees) exemplifying the analyses of various linguistic phenomena presented in the article. An extension to a real-size TAG grammar for French is ongoing.

1.3 Organisation of the article

In Section 2, we show a functional interpretation of the substitution and adjunction operations. We review the definitions that are necessary to model strings and trees as λ-terms (Section 2.2) and we use them to model the elementary trees of TAG. Section 3 reviews the definitions that are specific to ACGs. We show their main composition models and present their formal properties.

In Section 4, we introduce the ACGs that are necessary for the encoding of the syntax-semantics interface: the ACG relating strings and derived trees in Section 4.1; the ACG relating derived trees and generalized derivation trees in Section 4.2 and Section 4.3. We show that these generalized derivations over-generate with respect to TAG derivation trees: the full TAG encoding is not yet completed, but we already have all the necessary parts to implement the syntax-semantics interface.

Section 5 is devoted to the model of the syntax-semantics interface we propose. We first define the semantic representation language in Section 5.1. Then, in Section 5.2, we show how to interpret the

---

5 The example files are available at https://hal.inria.fr/hal-01242154/file/acg-examples.zip. The script file illustrates the terms we use in this article and refers in comments to the relevant sections, equations, and term names.
generalized derivations as semantic terms and we provide several classical examples.

In Section 6 we complete the faithful TAG encoding by controlling the generalized derivations so that only TAG derivations are accepted. The correctness of the encoding is ensured by recovering, by ACG composition, de Groote’s (2002) encoding. Then, again by ACG composition, we directly obtain a syntax-semantics interface for TAG. Because ACGs do not make use of features, we explain how we model the adjunction constraints induced by feature structures in TAG (Section 7).

In Section 8, we take advantage of the architecture we propose and give examples of modellings that this framework offers and that are beyond TAG. We finally discuss the relation of our approach to the syntax-semantics interface for TAG with other ones, in particular the ones using synchronous grammars or feature unification (Section 9).

2 BACKGROUND

2.1 Adjunction and substitution

A TAG consists of a finite set of elementary trees whose nodes are labelled by terminal and non-terminal symbols. Nodes labelled with terminals can only be leaves. Elementary trees are divided into initial and auxiliary trees. Figure 2 exemplifies such trees. Substituting $a_{John}$ in $\alpha_{sleeps}$ consists in replacing a leaf of $\alpha_{sleeps}$ labelled with a non-terminal symbol $NP$ with the tree $a_{John}$ whose root node is labelled by $NP$ as well. Figure 3(a) shows an example of such a substitution (at Gorn address 1) and of its result. The corresponding derivation tree recording the substitution is represented in Figure 3(b), where

\[ S \]
\[ NP \quad VP \]
\[ V \]
\[ sleeps \]
\[ a_{sleeps} = \]
\[ NP \quad VP \]
\[ V \]
\[ a_{John} = \]
\[ NP \quad John \]
\[ seemingly \]
\[ VP \quad Adv \quad VP^* \]
\[ \beta_{seemingly} = \]

Figure 2: TAG elementary trees

(a) Initial tree with one substitution node (b) Initial tree with no substitution node (c) Auxiliary tree

---

*Substitution sites are often marked by decorating the label with a ↓ symbol.*
the Gorn address labels the edge between the two nodes, each of them being labelled by the name of the trees. Only initial trees, that possibly underwent some substitution or adjunction operations, can substitute into a leaf.

The adjunction of $\beta_{\text{seemingly}}$ into $\alpha_{\text{slopes}}$ consists in inserting the tree $\beta_{\text{seemingly}}$ at the VP node of $\alpha_{\text{slopes}}$: the subtree of $\alpha_{\text{slopes}}$ rooted at its VP node is first removed then substituted to the VP foot node of $\beta_{\text{seemingly}}$ (the leaf with the same label as the root and marked with *). The whole resulting tree is then plugged again at the VP node of $\alpha_{\text{slopes}}$, as Figure 4(a) shows. The associated derivation tree of Figure 4(b) records the adjunction with a dotted edge. Only auxiliary trees, that possibly underwent some substitution or adjunction operations, can adjoin into another tree.

Figure 5 shows a TAG analysis of John seemingly sleeps, which involves both operations, and the associated derivation tree.

2.2 TAG elementary trees as functions

We now present the two operations of adjunction and substitution using a functional interpretation of the elementary trees. We use the
A syntax-semantics interface for TAG through ACG

Figure 5: TAG analysis of John seemingly sleeps

(a) Adjunction and substitution
(b) Derivation tree

standard notations of the typed λ-calculus and we formally present the syntax of λ-terms and their types.

Definition 1 (Types). Let \( A \) be a set of atomic types. The set \( \mathcal{T}(A) \) of implicative types built upon \( A \) is defined with the following grammar:

\[
\mathcal{T}(A) ::= A \mid \mathcal{T}(A) \rightarrow \mathcal{T}(A) \mid \mathcal{T}(A) \rightarrow \mathcal{T}(A)
\]

The set of linear implicative types built upon \( A \) is defined with the following grammar:

\[
\mathcal{T}^\circ(A) ::= A \mid \mathcal{T}^\circ(A) \rightarrow \mathcal{T}^\circ(A)
\]

Definition 2 (Higher-order signatures). A higher-order signature \( \Sigma \) 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;
- \( \tau : C \rightarrow \mathcal{T}(A) \) is a function assigning types to constants.

A higher-order signature \( \Sigma = \langle A, C, \tau \rangle \) is linear if the codomain of \( \tau \) is \( \mathcal{T}^\circ(A) \).

Definition 3 (λ-Terms). Let \( X \) be a countably infinite 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 \( \lambda^\circ x.t \in \Lambda(\Sigma) \);
• if \( x \in X \) and \( t \in \Lambda(\Sigma) \), then \( \lambda x.t \in \Lambda(\Sigma) \);

• if \( t, u \in \Lambda(\Sigma) \) then \( (t \ u) \in \Lambda(\Sigma) \).

Note there is a linear \( \lambda \)-abstraction (denoted by \( \lambda^o \)) and a (usual) intuitionistic \( \lambda \)-abstraction (denoted by \( \lambda \)). A variable that is bound by \( \lambda^o \) occurs exactly once in the body of the abstraction, whereas it can occur zero, one, or any number of times when it is bound by \( \lambda \). This distinction is important when discussing the complexity of parsing with ACGs.

We also use the usual notions of \( \alpha, \beta, \) and \( \eta \) conversions (Barendregt 1984), as well as the left associativity of (linear and non-linear) application (so \( (t \ u) \, v \) is written \( t \, (u \, v) \)), the right associativity of (linear and non-linear) abstraction over several variables (so \( \lambda x. (\lambda y. (\lambda z. t)) = \lambda x. y. z. t \), etc.; the same for \( \lambda^o \)), and the right associativity of implication (so \( \alpha \to (\beta \to \gamma) = \alpha \to \beta \to \gamma \); the same for \( \to \)).

**Definition 4** (Typing judgment). Given a higher-order signature \( \Sigma \), the typing rules are given with an inference system whose judgments are of the form: \( \Gamma; \Delta \vdash_{\Sigma} t : \alpha \) where:

- \( \Gamma \) is a finite set of non-linear variable typing declarations of the form \( x : \beta \) where \( x \) is a variable and \( \beta \) is a type;
- \( \Delta \) is a finite multi-set of linear variable typing declarations of the form \( x : \beta \) where \( x \) is a variable and \( \beta \) is a type. In order to distinguish the elements of the typing declaration, we always use variables with different names.

Both \( \Gamma \) and \( \Delta \) may be empty. If both of them are empty, we usually write \( t : \alpha \) (\( t \) is of type \( \alpha \)) instead of \( \vdash_{\Sigma} t : \alpha \). Moreover, we drop the \( \Sigma \) subscript when the context permits. Table 1 gives the typing rules: constant introduction, variable introduction (linear and non-linear),
linear abstraction and linear application, (non-linear) abstraction and (non-linear) application, in this order.

The fact that \( \Gamma \) is a set and \( \Delta \) is a multi-set corresponds to implicitly allowing for the structural rules of contraction and weakening for the non-linear context, and disallowing them on the linear context.\(^7\)

**Remark 1.** From a logical point of view, the theorems that can be proved using only the non-linear or the linear context are different. For instance, if \( c : \alpha \rightarrow \alpha \rightarrow \beta \) and \( d : \alpha \rightarrow \alpha \rightarrow \beta \) are constants of \( \Sigma \), \( x : \alpha ; \vdash c \, x \, x : \beta \) is derivable as the following derivation shows:

\[
\begin{array}{c}
\vdash x : \alpha ; \Sigma \vdash c : \alpha \rightarrow \alpha \rightarrow \beta \\
\hline
\vdash x : \alpha ; \Sigma \vdash (c \, x) : \alpha \rightarrow \beta \\
\hline
\vdash x : \alpha ; \Sigma \vdash (c \, x) \, x : \beta
\end{array}
\]

whereas \( ; x : \alpha \vdash d \, x \, x : \beta \) is not (and \( ; x : \alpha , y : \alpha ; \vdash d \, x \, y : \beta \) is).

**Remark 2.** The linear context of the second premise in the non-linear application rule (\( \Gamma ; \vdash \Sigma \ u : \alpha \)) is empty. This is required in order to avoid duplicating or erasing linear variables by non-linear application to a linear variable. Otherwise we could have derivations such as:

\[
\begin{array}{c}
\vdash \vdash \\
\hline
\vdash \vdash \vdash x : \alpha ; \Sigma \vdash c \, x \, x : \beta \\
\vdash \vdash \vdash \vdash y : \alpha ; \Sigma \vdash y : \alpha \\
\hline
\vdash \vdash \vdash y : \alpha ; \Sigma \vdash (\lambda x . c \, x \, x) \, y : \beta
\end{array}
\]

Then we have that \( y : \alpha \) belongs to the linear context, but \( (\lambda x . c \, x \, x) \, y \) reduces to \( c \, y \, y \) where \( y \) is duplicated.

**Definition 5** (Linear and almost linear \( \lambda \)-terms). A term without any \( \lambda \)s, where each \( \lambda^o \) binds exactly one variable, and where no subterm contains more than one free occurrence of the same variable is a **linear** \( \lambda \)-term, otherwise it is **non-linear**.

A term where each \( \lambda^o \) binds exactly one variable, where each \( \lambda \) binds at least one variable, and no subterm contains more than one free occurrence of the same variable, except if the variable has an atomic type, is an **almost linear** \( \lambda \)-term.

The notion of linearity and almost linearity are important with respect to the tractability and the computational complexity of the

\(^7\)Contraction corresponds to allowing the duplication of hypotheses, and weakening corresponds to allowing the deletion of useless hypothesis.
parsing algorithms, because they allow for characterising the set of almost linear \( \lambda \)-terms that are \( \beta \)-equivalent to some given almost linear term (Kanazawa 2017, p. 1120).

**Definition 6 (Order).** The order \( \text{ord}(\tau) \) of a type \( \tau \in \mathcal{T}(A) \) is inductively defined as:

- \( \text{ord}(a) = 1 \) if \( a \in A \)
- \( \text{ord}(\alpha \rightarrow \beta) = \max(1 + \text{ord}(\alpha), \text{ord}(\beta)) \) otherwise

By extension, the order of a term is the order of its type.

**Remark 3.** Second-order terms (i.e., terms whose order is 2) play an important role in our TAG encoding, and more generally for the expressive power of ACGs. A second-order term has type \( a_1 \rightarrow a_2 \ldots \rightarrow a_n \rightarrow a \) where \( a, a_1, \ldots, a_n \) are atomic types. If a signature \( \Sigma \) contains only first-order and second-order constants, an easy induction shows that ground terms (i.e., terms with no free variable) of atomic types in \( \Lambda(\Sigma) \) do not contain any variable (bound or free) at all. In particular, they cannot have terms of the form \( \lambda^0 \cdot x \cdot u \) or \( \lambda \cdot x \cdot u \) as sub-terms.

We now assume the single atomic type \( T \) of trees and constants of this type (in Section 2.3 we make explicit how to systematically encode trees into \( \lambda \)-terms).

### 2.2.1 Substitution as function application

The ability for the tree of Figure 6(a) to accept a substitution at its \( \text{NP} \) node allows it to be considered as a function that takes a tree as argument and replaces the \( \text{NP} \) node by this argument. Hence we can represent it as the function \( \gamma'_{\text{sleeps}} \) shown in Figure 6(b) with \( \gamma'_{\text{sleeps}} : T \rightarrow T \). A tree where no substitution can occur can be represented as \( \gamma_{\text{John}} : T \) (see Figure 6(c)).

![Figure 6: Functional interpretation of the substitution operation](image)
Applying the function $\gamma'_\text{sleeps}$ to the simple tree $\gamma_{\text{John}}$ of Figure 6(c) and performing $\beta$-reduction gives the expected result as (1) shows.

\[ (1) \quad \gamma'_\text{sleeps} \gamma_{\text{John}} = \left( \begin{array}{c}
S \\
\downarrow \quad \downarrow
\end{array} \quad \begin{array}{c}
\text{NP} \\
\text{VP} \\
\text{V} \\
\text{s}
\end{array} \quad \left( \begin{array}{c}
S \\
\downarrow \quad \downarrow
\end{array} \quad \begin{array}{c}
\text{NP} \\
\text{VP} \\
\text{V} \\
\text{s}
\end{array} \quad \begin{array}{c}
\text{John} \\
\text{John} \\
\text{V}
\end{array} \quad \begin{array}{c}
\text{s}
\end{array} \right) \rightarrow_{\beta} \begin{array}{c}
\text{VP} \\
\text{Adv} \\
\text{seemingly}
\end{array}
\]

Note that despite the derived tree for $\text{John sleeps}$ having a root labelled by $S$ and the elementary tree for $\text{John}$ having a root labelled by $\text{NP}$, they are represented by the terms $\gamma'_\text{sleeps} \gamma_{\text{John}}$ and $\gamma_{\text{John}}$ which both have type $T$. The issue of recording the distinction is addressed in Section 4.3.

2.2.2 Adjunction as function application

In order to deal with the adjunction operation, we first observe what happens to the auxiliary tree $\beta_{\text{seemingly}}$ in Figure 4 (p. 534): a subtree of the tree it is adjoined to (the $\text{VP}$ rooted subtree of $\alpha_{\text{sleeps}}$) is substituted at the $\text{VP}^*$ foot node of $\beta_{\text{seemingly}}$. This means that the auxiliary tree also behaves as a function from trees to trees and can be represented as in Figure 7(a) with $\gamma'_{\text{seemingly}} : T \rightarrow T$. Then, a tree with an adjunction site can be represented by a term such as $\gamma''_{\text{sleeps}} : (T \rightarrow T) \rightarrow T$ in Figure 7(b). Note the higher-order type of $\gamma''_{\text{sleeps}}$.

In order to model the adjunction, we then apply $\gamma''_{\text{sleeps}}$ to $\gamma'_{\text{seemingly}}$ and perform $\beta$-reductions as (2) shows.

\[ \gamma'_{\text{seemingly}} = \lambda^o v. \left( \begin{array}{c}
\text{VP} \\
\text{Adv} \\
\text{seemingly}
\end{array} \right) \quad \gamma''_{\text{sleeps}} = \lambda^o a. \left( \begin{array}{c}
\text{S} \\
\text{NP} \\
\text{VP} \\
\text{V} \\
\text{s}
\end{array} \right) \]

(a) Function from trees to trees (b) Elementary tree ready to accept an adjunction

Figure 7: Functional interpretation of the adjunction operation
We are now (almost) in a position to define the function standing for the elementary tree representing the intransitive verb \( \text{sleeps} \) in its canonical form as in Figure 8 with \( \gamma'_{\text{sleeps}} : (T \rightarrow T) \rightarrow T \rightarrow T \). Such a term can be used to represent the TAG analysis of \( \text{John seemingly sleeps} \) shown in Figure 5 with the \( \beta \)-reduction of \( \gamma'_{\text{sleeps}} \gamma'_{\text{seemingly}} \gamma_{\text{John}} \) shown in (3).

**Remark 4** (No adjunction). Typing \( \gamma'_{\text{sleeps}} \) with \( (T \rightarrow T) \rightarrow T \rightarrow T \) makes it require an adjunction (the first \( (T \rightarrow T) \) argument) to return a
plain tree term of type $T$. But of course, we also want to use this term in case no adjunction in a TAG analysis would occur, as in $\text{John sleeps}$. We make use of a fake adjunction, applying $\gamma'_{\text{sleeps}}$ to the identity function $I = \lambda^o x. x : T \rightarrow T$. Then (4) holds.

$$
\begin{array}{c}
\gamma'_{\text{sleeps}} \mapsto_\beta \gamma_{\text{John}} \\
\end{array}
$$

Finally, we also have to model the possible adjunction on the $s$ node of $a_{\text{sleeps}}$. So the corresponding term $\gamma_{\text{sleeps}}$ has type $(T \rightarrow T) \rightarrow (T \rightarrow T) \rightarrow T \rightarrow T$ where the first argument stands for the auxiliary tree to be adjoined at the $s$ node, the second argument stands for the auxiliary tree to be adjoined at the $\text{NP}$ node, and the third argument stands for the tree to be substituted at the $\text{NP}$ node as Figure 9 shows.

Remark 5 (Multiple adjunction). Following Vijay-Shanker (1987), the typing we provide prevents two adjunctions from occurring at the same node in the same elementary tree. We discuss this difference with the multiple-adjunction approach of Schabes and Shieber (1994) in Section 5. Accordingly, an auxiliary tree should typically also allow for adjunction at its root. So instead of using $\gamma'_{\text{seemingly}} : T \rightarrow T$, we use the terms defined in Figure 10 in order to analyze sentences

---

\footnote{This idea is present in the seminal work using ACGs (de Groote 2002; Pogodalla 2004a), but also in the synchronous approach (Shieber 2004, 2006) and in Shieber (2014), in the notion of \textit{vestigial auxiliary tree}.}

\footnote{We could also allow adjunctions to the $v$ node in a similar way. But we do not use examples of such adjunctions, and, for the sake of conciseness, we keep the type as small as required by the examples.}
Figure 9: Encoding of \( \alpha_{\text{sleeps}} \) available to substitution and adjunctions both at the VP and at the S nodes

\[
\gamma_{\text{sleeps}} = \lambda^o S \ a \ s.S
\]

Figure 10: Auxiliary tree representation available to adjunction operations

\[
\gamma_{\text{seemingly}} = \lambda^o a \ v.a \left( \begin{array}{c}
\text{Adv} \\
\text{seemingly}
\end{array} \right) : (T \to T) \to T \to T
\]

\[
\gamma_{\text{usually}} = \lambda^o a \ v.a \left( \begin{array}{c}
\text{Adv} \\
\text{usually}
\end{array} \right) : (T \to T) \to T \to T
\]

Figure 11: A TAG analysis of John usually seemingly sleeps as in Figure 11 with the term \( \gamma_{\text{sleeps}} (\gamma_{\text{seemingly}} (\gamma_{\text{usually}} I)) \gamma_{\text{John}} \).

2.3 Trees and strings as \( \lambda \)-terms

So far, we did not make explicit how to represent strings and trees as \( \lambda \)-terms. In particular, we did not explain how strings can combine and how the parent-child relation can be represented. While this is

\[10\] Although \( \lambda^n v, \gamma'_{\text{usually}} (\gamma'_{\text{seemingly}} v) = \gamma_{\text{seemingly}} (\gamma_{\text{usually}} I) \), introducing \( \gamma_{\text{seemingly}} \) and \( \gamma_{\text{usually}} \) with this more complex types is important because, as we will see in Section 6, at the most abstract level, we want terms without any free or bound variable to represent derivations (see Remark 3).
A syntax-semantics interface for TAG through ACG

quite standard, and because we use this encoding to implement the
text example grammars using the ACG toolkit, this section describes how
it can be done.

2.3.1 Encoding strings

We encode strings over an alphabet \( C \) using the following higher-order
signature \( \Sigma_{strings}^C = \langle A_\sigma, C, \tau_\sigma \rangle \) where:

- \( A_\sigma = \{ o \} \) contains a unique atomic type \( o \);
- \( \tau_\sigma \) is the constant function that maps any constant to the type \( \sigma \),
  the string type, defined as \((o \rightarrow o)\). Note it is not an atomic type.

We use the notation \( \Delta \) to introduce terms or types that are defined
using the atomic types and the constants of a signature, but are not
atomic types nor constants of this signature. So \( \sigma \overset{\Delta}{=} (o \rightarrow o) \). We also
define two other terms:

- \( + \overset{\Delta}{=} \lambda^o f \ g . \lambda^o z . f (g \ z) \) (function composition, used with an infix
  notation) to represent concatenation;
- \( \epsilon \overset{\Delta}{=} \lambda^o x . x \) (the identity function) to represent the empty string.

It is easy to check that + is associative and that \( \epsilon \) is a neutral element
for +.

Remark 6. If \( a \) and \( b \) are two strings, \( a + b = \lambda^o z . a \ (b \ z) \). In this article,
we usually do not unfold the definition of + and we use the notation
\( x_1 + \ldots + x_n \) to represent the string \( \lambda^o z . x_1 (\ldots (x_n \ z) \ldots) \).

2.3.2 Encoding trees

Trees were originally defined in TAG using a mapping from positions
(or Gorn addresses) to labels, elements of a vocabulary (Joshi et al.
1975). Hence, the same node label could be used to represent nodes
of different arity. For instance, in Figure 2(a) (p. 533), the \( \forall p \) node has
arity 1 whereas the \( \forall p \) node of Figure 2(c) has arity 2.

We prefer to represent trees as terms over a ranked alphabet as
in Shieber (2004) and Comon et al. (2007) in order to make the en-
coding of trees as \( \lambda \)-terms easier. So we use the notation \( X_n \) with \( n \)
the arity of the symbol \( X_n \). It allows us to distinguish two nodes with
\( n \) and \( m \) \( (n \neq m) \) children that would be mapped to the same label
\( X \) by using the different symbols \( X_n \) and \( X_m \). As terminal symbols can
only occur as leaves, they always have arity 0, so we do not use any subscript for them.

In order to encode the trees defined over a ranked alphabet \( \mathcal{F}_a = (\mathcal{F}, \text{arity}) \), where \( \text{arity} \) is a mapping from \( \mathcal{F} \) to \( \mathbb{N} \), we use the following higher-order signature \( \Sigma_{\text{trees}}^{\mathcal{F}_a} = (A_T, \mathcal{F}, \tau_{\mathcal{F}_a}) \) where:

- \( A_T = \{T\} \) contains a unique atomic type \( T \), the type of trees;
- \( \tau_{\mathcal{F}_a} \) is a function that maps any constant \( X \) such that \( \text{arity}(X) = n \) to the type \( T \overset{n}{\rightarrow} T \). If \( \text{arity}(X) = 0 \), then \( \tau_{\mathcal{F}_a}(X) = T \).

For instance, the TAG elementary trees \( \delta_{\text{anchor}} \) of our running examples can be modelled as the functions (or terms) \( \gamma_{\text{anchor}} \) built on the signature \( \Sigma_{\text{trees}} \) as Table 2 shows.\(^{12}\) Then (5) shows that

<table>
<thead>
<tr>
<th>Terms of ( \Lambda(\Sigma_{\text{trees}}) )</th>
<th>Corresponding TAG tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gamma_{\text{John}} = \text{NP}_1 \text{ John} )</td>
<td>\text{NP} \text{John}</td>
</tr>
<tr>
<td>: ( T )</td>
<td></td>
</tr>
<tr>
<td>( \gamma_{\text{sleeps}} = \lambda^o s. a \ S \ (s_2 \ s \ (a \ (v_1 \ (v_1 \ \text{sleeps})))) )</td>
<td>\text{S} \text{sleeps}</td>
</tr>
<tr>
<td>: ( (T \rightarrow T) \rightarrow (T \rightarrow T) \rightarrow T \rightarrow T )</td>
<td></td>
</tr>
<tr>
<td>( \gamma_{\text{seemingly}} = \lambda^o a \ v. a \ (v_2 \ (\text{Adv}_1 \ \text{seemingly}) \ v) )</td>
<td>\text{Adv} \text{seemingly}</td>
</tr>
<tr>
<td>: ( (T \rightarrow T) \rightarrow T \rightarrow T )</td>
<td></td>
</tr>
<tr>
<td>( \gamma_{\text{usually}} = \lambda^o a \ v. a \ (v_2 \ (\text{Adv}_1 \ \text{usually}) \ v) )</td>
<td>\text{Adv} \text{usually}</td>
</tr>
<tr>
<td>: ( (T \rightarrow T) \rightarrow T \rightarrow T )</td>
<td></td>
</tr>
<tr>
<td>( \gamma_{\text{hence}} = \lambda^o s. a \ (s_2 \ (\text{Adv}_1 \ \text{hence}) \ s) )</td>
<td>\text{Adv} \text{hence}</td>
</tr>
<tr>
<td>: ( (T \rightarrow T) \rightarrow T \rightarrow T )</td>
<td></td>
</tr>
<tr>
<td>( I = \lambda^o x. x : T \rightarrow T )</td>
<td></td>
</tr>
</tbody>
</table>

\(^{11}\) We use the notation \( \delta_{\text{anchor}} \) to refer either to the initial tree \( \alpha_{\text{anchor}} \) or to the auxiliary tree \( \beta_{\text{anchor}} \).

\(^{12}\) Note that \( \text{sleeps} \) and \( \text{seemingly} \) are used as constants of arity 0 and have type \( T \). We also introduce an auxiliary tree that can adjoin to the \( S \) node.
A syntax-semantics interface for TAG through ACG

\[ \gamma_5 = \gamma_{sleeves} I (\gamma_{seemingly} I) \gamma_{John} \text{ corresponds (modulo } \beta \text{) to the tree of Figure 5 (p. 535).} \]

(5) \[ \gamma_5 = \gamma_{sleeves} I (\gamma_{seemingly} I) \gamma_{John} \rightarrow_\beta S_2 (NP_1 \text{John}) (VP_2 (Adv_1 seemingly)) (VP_1 (V_1 sleeps))) \]

We now want to relate the tree that is represented by the term \( \gamma_{sleeves} I (\gamma_{seemingly} I) \gamma_{John} \) to the string \( \text{John seemingly sleeps} \) that is represented by the term \( \text{John} + \text{seemingly} + \text{sleeps} \). We do this in Section 4, using an interpretation of the former as defined by an ACG.

3 ABSTRACT CATEGORIAL GRAMMARS

Grammars can be considered as a device to relate concrete objects to hidden underlying structures. For instance, context-free grammars relate strings to syntactic trees, and TAGs relate derived trees to derivation trees. However, in both cases, the underlying structure is not a first-class citizen of the formalism.

ACGs take another perspective and provide the user a direct way to define the parse structures of the grammar, the abstract language. Such structures are later on interpreted by a morphism, the lexicon, to get the concrete object language. The process of recovering an abstract structure from an object term is called ACG parsing and consists in inverting the lexicon. In this perspective, derivation trees of TAGs are represented as terms of an abstract language, while derived trees and yields are represented by terms of some other object languages: an object language of trees in the first case and an object language of strings in the second. We also use a logical language as the object language to express the semantic representations.

For the sake of self-containedness, we first review the definitions of de Groote (2001).

Definition 7 (Lexicon). 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, \( \Sigma_1 \) being linear. A lexicon \( \mathcal{L} = \langle F, G \rangle \) from \( \Sigma_1 \) to \( \Sigma_2 \) is such that:

- \( F : A_1 \rightarrow \mathcal{T}(A_2) \). We also note \( F : \mathcal{T}(A_1) \rightarrow \mathcal{T}(A_2) \), its homomorphic extension; that is, the function \( \hat{F} \) that extends \( F \) such that \( \hat{F}(\alpha \rightarrow \beta) = \hat{F}(\alpha) \rightarrow \hat{F}(\beta) \) and \( \hat{F}(\alpha \rightarrow \beta) = \hat{F}(\alpha) \rightarrow \hat{F}(\beta) \);
• \( G : C_1 \to \Lambda(\Sigma_2) \). We also note \( G : \Lambda(\Sigma_1) \to \Lambda(\Sigma_2) \), its homomorphic extension; that is, the function \( \hat{G} \) that extends \( G \) such that \( \hat{G}(t \, u) = \hat{G}(t) \, \hat{G}(u), \hat{G}(x) = x, \hat{G}(\lambda x. t) = \lambda x. \hat{G}(t), \) and \( \hat{G}(\lambda^o x. t) = \lambda^o x. \hat{G}(t); \)

• \( F \) 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 \( L \) instead of \( F \) or \( G \).

The lexicon is the interpreting device of ACGs.

**Definition 8** (Abstract Categorial Grammar and vocabulary). An **abstract categorial grammar** is a quadruple \( G = (\Sigma_1, \Sigma_2, L, S) \) 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. \( \Sigma_1 \) (resp. \( \Sigma_2 \)) is called the **abstract vocabulary** (resp. the **object vocabulary**) and \( \Lambda(\Sigma_1) \) (resp. \( \Lambda(\Sigma_2) \)) is the set of **abstract terms** (resp. the set of **object terms**).

- \( L : \Sigma_1 \to \Sigma_2 \) is a lexicon and \( \Sigma_1 \) is a linear signature.

- \( S \) is the **distinguished type** of the grammar.

Given an ACG \( G_{\text{name}} = (\Sigma_1, \Sigma_2, L_{\text{name}}, S) \), instead of using \( L_{\text{name}}(\alpha) = \beta \) (resp. \( L_{\text{name}}(t) = u \)) in order to express that the interpretation of the type \( \alpha \) is the type \( \beta \) (resp. the interpretation of the term \( t \) is the term \( u \)), we use the following notational variants: \( G_{\text{name}}(\alpha) = \beta \) and \( \alpha :=_{\text{name}} \beta \) (resp. \( G_{\text{name}}(t) = u \) and \( t :=_{\text{name}} u \)). The subscript may be omitted if clear from the context.

**Definition 9** (Abstract and object languages). Given an ACG \( G \), the **abstract language** is defined by

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

The **object language** is defined by

\[ \mathcal{O}(G) = \{ u \in \Lambda(\Sigma_2) \mid \exists t \in \mathcal{A}(G) \text{ such that } u = L(t) \} \]

In this article, we consider object languages such as strings or logical formulas, and abstract languages such as derivation trees. Some languages, such as the language of derived trees, will be considered sometimes as object languages, sometimes as abstract languages.

Parsing with an ACG \( G \) any term \( u \) that is built over the object vocabulary of \( G \) amounts to finding the abstract terms \( t \in \mathcal{A}(G) \) such that \( u = G(t) \). In other words, ACG parsing is morphism inversion.
3.1 ACG composition

The lexicon defines the way structures are interpreted. It plays a crucial role in our proposal in two different ways. First, two interpretations may share the same abstract vocabulary, hence mapping a single structure into two different ones. For instance, the structure representing the derivations may be mapped both into a surface form and a semantic form. This composition is illustrated by $G_{\text{derived trees}}$ and $G_{\text{sem.}}$ sharing the $\Sigma_{\text{derivations}}$ vocabulary in Figure 12. It corresponds to the core model of the syntax-semantics interface as proposed in ACGs (de Groote 2001, Section 2.3), but also to the one proposed in synchronous TAG. It allows us to relate the derived trees and the semantic expressions that have the same derivation structures. We use this in Section 5 as our model of the syntax-semantics interface for TAG.

Second, the result of a first interpretation can itself be interpreted by a second lexicon when the object vocabulary of the first interpretation is the abstract vocabulary of the second one. This composition, illustrated by the $G_{\text{yield}} \circ G_{\text{derived trees}}$ composition in Figure 12, provides modularity. It also allows one to control admissible intermediate structures. For instance, the abstract language of $G_{\text{yield}}$ may contain too many structures. If the object language of $G_{\text{derived trees}}$ is a strict subset of this abstract language, then the object language of $G_{\text{yield}} \circ G_{\text{derived trees}}$ is a subset of the object language of $G_{\text{yield}}$. We take advantage of this property in Section 4.3 to enforce the matching between node labels in substitution and adjunction operations, and to further restrict the set of derivations to only TAG derivations in Section 6. This ACG composition corresponds to the typical ACG way to control admissible parse structures.
3.2 Formal properties of ACGs

In this section, we review the main properties of ACGs and mention the relevant references. Two parameters are useful to define a hierarchy of ACGs: the order and the complexity of an ACG.

**Definition 10** (Order and complexity of an ACG; ACG hierarchy). The *order* of an ACG is the maximum of the orders of its abstract constants. The *complexity* of an ACG is the maximum of the orders of the realizations of its atomic types.

We call *second-order ACGs* the set of ACGs whose order is at most 2.

$\text{ACG}_{(n,m)}$ denotes the set of ACGs whose order is at most $n$ and whose complexity is at most $m$.

For instance, in Figure 12, $\Sigma_{\text{yield}}$ is a second-order ACG (because all the constants of $\Sigma_{\text{trees}}$ are of type $T \rightarrow \cdots \rightarrow T$ with $T$ atomic, hence are at most second-order). On the other hand, $\Sigma_{\text{derivations}}$ is third-order (it contains terms such as $c_{\text{seemingly}} : (\text{VP} \rightarrow \text{VP}) \rightarrow \text{VP} \rightarrow \text{VP}$, where $\text{NP}$ and $\text{VP}$ are atomic, that are third-order; see Section 4). Hence $\Sigma_{\text{derived trees}}$ is third-order as well.

The class of second-order ACGs is of particular interest because of its polynomial parsing property (Salvati 2005). When considering strings as the object language, the generated languages coincide with multiple context-free languages (Salvati 2006). When considering trees, the generated languages coincide with the tree languages generated by hyperedge replacement grammars (Kanazawa 2009).

A further refinement on the ACG hierarchy provides a fine-grained correspondence with regular (string or tree) languages, context-free string and linear context-free tree languages, or well-nested multiple context-free languages (string), in particular tree-adjoining languages. Table 3 (p. 549) sums up some of the formal properties of second-order ACGs (de Groote and Pogodalla 2004; Salvati 2006; Kanazawa and Salvati 2007; Kanazawa 2009).

For second-order ACGs, parsing algorithms and optimization techniques are grounded on well established fields such as type-theory and Datalog. Kanazawa (2007) showed how parsing of second-order ACGs reduces to Datalog querying, offering a general method for getting efficient tabular parsing algorithms (Kanazawa 2017). This parsing method applies whatever the object language: representing
A syntax-semantics interface for TAG through ACG

<table>
<thead>
<tr>
<th></th>
<th>String language</th>
<th>Tree language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACG((1,n))</td>
<td>finite</td>
<td>finite</td>
</tr>
<tr>
<td>ACG((2,1))</td>
<td>regular</td>
<td>regular</td>
</tr>
<tr>
<td>ACG((2,2))</td>
<td>context-free</td>
<td>linear context-free</td>
</tr>
<tr>
<td>ACG((2,3))</td>
<td>non-duplicating macro well-nested multiple context-free</td>
<td>(\subset 1)-visit attribute grammar</td>
</tr>
<tr>
<td>ACG((2,4))</td>
<td>mildly context-sensitive (multiple context-free)</td>
<td>tree-generating hyperedge replacement gram.</td>
</tr>
<tr>
<td>ACG((2,4+n))</td>
<td>ACG((2,4))</td>
<td>ACG((2,4))</td>
</tr>
</tbody>
</table>

strings, trees, and also any kind of (almost linear) \(\lambda\)-terms. When the object language consists of logical formulas, the latter can then be parsed as well, and the resulting parse structures can further be interpreted (e.g., as strings) to implement surface realization (Kanazawa 2007). This also allows for deriving algorithms with specific properties such as prefix-correctness in a general way.\(^{13}\)

The computational properties of lexicon inversion for ACGs have been studied for different classes of \(\lambda\)-terms.\(^{14}\) It is worth noting that, as far as second-order ACGs are concerned, whatever the form of the semantic \(\lambda\)-term, lexicon inversion is decidable (Salvati 2010), even with replication and vacuous abstraction of variables, though with a high computational complexity. From a theoretical point of view, this corresponds to removing any kind of semantic monotonicity requirement for generation (Shieber 1988; Shieber et al. 1989) in a very general setting.\(^{15}\)

The examples we present in this article use only almost linear semantic terms. This allows us to run them in the ACG toolkit. The latter implements the method of parsing by reduction to Datalog, and allows

\(^{13}\) For a \(n^6\) prefix-correct Earley recognizer for TAGs, see Kanazawa (2008b).

\(^{14}\) See for instance Kanazawa (2017), Bourreau and Salvati (2011), and Bourreau (2012) for the linear, almost linear, and almost affine cases.

\(^{15}\) In its strongest form, this requirement corresponds to having lexicalized semantic recipes (i.e., where at least one constant appears and no deletion is allowed). Linear and almost linear pure terms (i.e., where no constant occurs) are already dealt with in the Datalog reduction. Allowing deletion leads to more challenging issues. It is used, for instance, for the modelling of ellipsis (Kobele 2007; Bourreau 2012, 2013) or for providing intensional semantics to intension-insensitive words (de Groote and Kanazawa 2013).
us to parse strings, trees, and logical formulas using the grammars we propose. Large scale tests of the software are however ongoing work, and a quantitative evaluation is beyond the scope of this article.

Parsing with ACGs whose order is strictly greater than 2 is equivalent (Salvati 2005) to the decidability of the Multiplicative Exponential fragment of Linear Logic (MELL: Girard 1987). De Groote (2015) shows a reduction of ACG parsing of higher-order ACGs to linear logic programming. It is of course trivially (by linearity) decidable for ACGs where the interpretation of abstract constants always introduces a constant of the object language. But even in this case, third-order ACGs can generate languages that are NP-complete (Yoshinaka and Kanazawa 2005; Salvati 2006). For higher-order ACGs, the ACG toolkit implements abstract term interpretation, but no parsing.

4 RELATING GENERALIZED DERIVATIONS, TAG DERIVED TREES, AND STRINGS WITH ABSTRACT CATEGORIAL GRAMMARS

From now on, we assume a TAG \( \mathcal{G} = (I, A) \) where \( I \) is the set of initial trees and \( A \) the set of auxiliary trees. The labels of the trees in \( I \cup A \) range over the alphabet \( V^0 \), and \( C \subset V^0 \) is the terminal alphabet. \( V \) is the set of symbols of \( V^0 \) disambiguated by subscripting them with their arity (except for terminal symbols of arity 0), and \( V \) is the associated ranked alphabet.

4.1 Derived trees and strings

In the constructions of Section 2.3, we introduced two higher-order signatures: \( \Sigma_{\text{strings}} = \Sigma_{\text{strings}}^C \) and \( \Sigma_{\text{trees}} = \Sigma_{\text{trees}}^V \). We can now relate terms built on them using an ACG \( \mathcal{G}_{\text{yield}} = (\Sigma_{\text{trees}}, \Sigma_{\text{strings}}, \mathcal{L}_{\text{yield}}, T) \) by specifying \( \mathcal{L}_{\text{yield}} \) as follows:

- \( \mathcal{L}_{\text{yield}}(T) = \sigma \) (a tree is interpreted as a string);
- for \( X_n \in V \setminus C \), \( \mathcal{L}_{\text{yield}}(X_n) = \lambda x_1 \cdots x_n . x_1 + \cdots + x_n \) (a tree labelled by a non-terminal symbol is interpreted by the function that concatenates the interpretation of its children);
- for \( a \in C \), \( \mathcal{L}_{\text{yield}}(a) = a \) (a terminal symbol is interpreted by the same symbol as a string).

\[ \text{[ 550 ]} \]
A syntax-semantics interface for TAG through ACG

For instance, (8) (p. 552) shows that the yield of the tree represented by $\gamma_5 = \gamma_{\text{sleeps } I} \gamma_{\text{seemingly } I} \gamma_{\text{John}}$ (p. 545) actually is $\text{John} + \text{seemingly} + \text{sleeps}$ (which can be rephrased as $\gamma_{\text{sleeps } I} (\gamma_{\text{seemingly } I} \gamma_{\text{John}}) := \text{yield } \text{John} + \text{seemingly} + \text{sleeps}$). Indeed, we have (6).

\[
\begin{align*}
\mathcal{G}_{\text{yield}}(\gamma_{\text{John}}) &= \mathcal{G}_{\text{yield}}(\text{NP}_1 \text{ John}) \\
&= \mathcal{G}_{\text{yield}}(\text{NP}_1) \mathcal{G}_{\text{yield}}(\text{John}) \\
&= (\lambda^o x. x) \text{ John} \\
&\text{because } \mathcal{G}_{\text{yield}} \text{ is a morphism} \\
&\to_{\beta} \text{ John}
\end{align*}
\]

And with $\gamma_7 = \text{VP}_2 (\text{Adv}_1 \text{ seemingly}) (\text{VP}_1 (\text{v}_1 \text{ sleeps}))$, we have (7),

\[
\begin{align*}
\mathcal{G}_{\text{yield}}(\gamma_7) &= \mathcal{G}_{\text{yield}}(\text{VP}_2 (\text{Adv}_1 \text{ seemingly}) (\text{VP}_1 (\text{v}_1 \text{ sleeps}))) \\
&= \mathcal{G}_{\text{yield}}(\text{VP}_2) (\mathcal{G}_{\text{yield}}(\text{Adv}_1 \text{ seemingly})) (\mathcal{G}_{\text{yield}}(\text{VP}_1 (\text{v}_1 \text{ sleeps}))) \\
&= (\lambda^o x_1 x_2. x_1 + x_2) (\mathcal{G}_{\text{yield}}(\text{Adv}_1 \text{ seemingly})) (\mathcal{G}_{\text{yield}}(\text{VP}_1 (\text{v}_1 \text{ sleeps}))) \\
&\text{because } \mathcal{G}_{\text{yield}} \text{ is a morphism} \\
&\to_{\beta} \mathcal{G}_{\text{yield}}(\text{Adv}_1 \text{ seemingly}) + \mathcal{G}_{\text{yield}}(\text{VP}_1 (\text{v}_1 \text{ sleeps})) \\
&= (\mathcal{G}_{\text{yield}}(\text{Adv}_1) \mathcal{G}_{\text{yield}}(\text{seemingly})) + (\mathcal{G}_{\text{yield}}(\text{VP}_1) (\mathcal{G}_{\text{yield}}(\text{v}_1 \text{ sleeps}))) \\
&\text{because } \mathcal{G}_{\text{yield}} \text{ is a morphism} \\
&= ((\lambda^o x. x) \mathcal{G}_{\text{yield}}(\text{seemingly})) + ((\lambda^o x. x) (\mathcal{G}_{\text{yield}}(\text{v}_1 \text{ sleeps}))) \\
&\text{by definition of } \mathcal{G}_{\text{yield}} \text{ on } \text{Adv}_1 \text{ and } \text{VP}_1 \\
&= \mathcal{G}_{\text{yield}}(\text{seemingly}) + \mathcal{G}_{\text{yield}}(\text{v}_1 \text{ sleeps}) \\
&= \text{seemingly} + (\mathcal{G}_{\text{yield}}(\text{v}_1) (\mathcal{G}_{\text{yield}}(\text{sleeps}))) \\
&= \text{seemingly} + ((\lambda^o x. x) \text{ sleeps}) \\
&= \text{seemingly} + \text{sleeps}
\end{align*}
\]

hence (8):
In this section, we illustrate how to introduce more control on the accepted structures. Note indeed that according to the definition of $G_{yield}$, whatever is a closed term of type $T$ belongs to its abstract language. For instance, $\gamma_{13} = \gamma_{seemingly} I \gamma_{John}$ is a well-typed term of type $T$ corresponding to the tree of Figure 13 as (9) shows. Consequently, its interpretation $seemingly + John$ belongs to the object language.

In order to avoid such terms belonging to the language we are interested in, we provide another ACG, $G_{derived\ trees}$, such that its object language is a strict subset of $\mathcal{A}(G_{yield})$ (see Figure 12 p. 547). Consequently, the object language of $G_{yield} \circ G_{derived\ trees}$ is a subset (strict in this case, as expected) of $\mathcal{O}(G_{yield})$. $G_{derived\ trees}$ is defined as $G_{derived\ trees} = (\Sigma_{derivations}, \Sigma_{trees}, \mathcal{L}_{derived\ trees}, S)$.

4.3 Generalized derivation trees

4.3.1 A vocabulary for derivations: the $\Sigma_{derivations}$ signature

Adjoining $\gamma_{seemingly}$ on $\gamma_{John}$ is possible in $\Lambda(\Sigma_{trees})$ because the type $T$ does not take the node labels into account. Hence, there is, for in-
A syntax-semantics interface for TAG through ACG

stance, no distinction between trees rooted by VP and trees rooted by NP. We introduce this distinction in a new signature \( \Sigma_{\text{derivations}} = \langle A_{\text{derivations}}, C_{\text{derivations}}, \tau_{\text{derivations}} \rangle \). \( A_{\text{derivations}} = V^0 \) is the set of non-terminal symbols of the TAG grammar \( \mathcal{G} \). Then, for any \( \delta_{\text{anchor}} \in \mathcal{J} \cup \mathcal{A} \), an elementary tree of \( \mathcal{G} \), we define \( c_{\text{anchor}} \) a constant of type \((X^1 \rightarrow X^1) \rightarrow \cdots \rightarrow (X^n \rightarrow X^n) \rightarrow Y^1 \rightarrow \cdots \rightarrow Y^m \rightarrow \alpha\) where:

- the \( X^i \) are the labels of the \( n \) internal nodes of \( \delta_{\text{anchor}} \) labelled with a non-terminal where an adjunction is possible (by convention we use the breadth-first traversal); \(^{17}\)
- the \( Y^i \) are the labels of the \( m \) leaves labelled with non-terminals, not counting the foot node if \( \delta_{\text{anchor}} \) is an auxiliary tree, of \( \delta_{\text{anchor}} \) (by convention, we use the left-right order);
- let \( Z \) be the label of the root node of \( \delta_{\text{anchor}} \). \( \alpha = Z \) if \( \delta_{\text{anchor}} \in \mathcal{J} \) is an initial tree, and \( \alpha = Z'' \rightarrow Z' \) with \( Z'' \) corresponding to the label of the foot node and \( Z' \) corresponding to the label of the root node if \( \delta_{\text{anchor}} \in \mathcal{A} \) is an auxiliary tree. \(^{18}\) In the latter case, we call \( Z'' \rightarrow Z' \) the modifier type of the constant modelling the auxiliary tree.

We get for instance the constants typed as in (10) \(^{19}\) from the elementary trees of Figure 2 (p. 533).

\[
\begin{align*}
c_{\text{sleeps}} : (S \rightarrow S) & \rightarrow (VP \rightarrow VP) \rightarrow NP \rightarrow S \\
c_{\text{John}} : NP \\
c_{\text{seemingly}} : (VP \rightarrow VP) & \rightarrow VP \rightarrow VP
\end{align*}
\]

(10)

For each non-terminal \( X \) of the TAG grammar where an adjunction can occur, we also define \( I_X : X \rightarrow X \) as in (11). These constants play a similar role as \( I \) at the \( \Sigma_{\text{trees}} \) level: they are used when a TAG derivation does not involve any adjunction on sites where it would be possible to have some.

\(^{17}\) Instead of the types \((X^i \rightarrow X^i)\), we may have types \(X^i_{\text{top}} \rightarrow X^i_{\text{bottom}}\) to denote a difference between the top and the bottom feature of the node of label \( X^i \). This is in particular used to account for selecting adjoining constraints as described in Feature-based TAG (FTAG: Vijay-Shanker and Joshi 1988, 1991). See note 18.

\(^{18}\) In standard TAG, we typically have \( Z = Z' = Z'' \). However, we shall see examples in Sections 5.3.2 and 7 where the distinction between \( Z, Z' \), and \( Z'' \) is relevant.

\(^{19}\) We assume that no adjunction is allowed on the \( v \) node nor on the \( \text{Adv} \) node.
Then the set of typed constants of $\Sigma_{\text{derivations}}$ is $C_{\text{derivations}} = \{c_{\text{anchor}} \mid \delta_{\text{anchor}} \in J \cup A\} \cup \{I_x \mid X \in V^0\}$ and $\tau_{\text{derivations}}$ is the associated typing function defined as above. The typing provided by $\Sigma_{\text{derivations}}$ now disallows the application of $c_{\text{seemingly}}$ $I_{VP} : VP \rightarrow VP$ to $c_{John} : NP$.

We now need to relate the terms of $\Lambda(\Sigma_{\text{derivations}})$ to the terms of $\Lambda(\Sigma_{\text{trees}})$ by a suitable interpretation.

### 4.3.2 Interpretation of derivations as derived trees: the $\otimes_{\text{derived trees}}$ ACG

In order to define $\otimes_{\text{derived trees}} = (\Sigma_{\text{derivations}}, \Sigma_{\text{trees}}, \mathcal{L}_{\text{derived trees}}, S)$ we are left with defining $\mathcal{L}_{\text{derived trees}}$. All the atomic types ($S$, $VP$, etc.) are interpreted as trees (i.e., with the $T$ type). And for a TAG elementary tree $\delta_{\text{anchor}}$, the constant $c_{\text{anchor}}$ is interpreted by $\gamma_{\text{anchor}}$ (defined in Section 2.3.2). This leads us to the interpretations of Table 4.

<table>
<thead>
<tr>
<th>$c_{\text{John}}$</th>
<th>$\quad = NP_1$ $\quad$ John : $T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{\text{sleeps}}$ : $(S \rightarrow S) \rightarrow (VP \rightarrow VP) \rightarrow NP \rightarrow S$</td>
<td></td>
</tr>
<tr>
<td>$\equiv$ $\text{derived trees } \gamma_{\text{sleeps}}$ $\quad = \lambda^o a \rightarrow a (S \rightarrow (VP \rightarrow VP) \rightarrow S)$</td>
<td></td>
</tr>
<tr>
<td>$c_{\text{seemingly}}$ : $(VP \rightarrow VP) \rightarrow VP \rightarrow VP$</td>
<td></td>
</tr>
<tr>
<td>$\equiv$ $\text{derived trees } \gamma_{\text{seemingly}}$ $\quad = \lambda^o a \rightarrow a (VP_2 (\text{Adv}_1 \text{ seemingly}) \rightarrow VP)$</td>
<td></td>
</tr>
<tr>
<td>$\gamma_{\text{usually}}$ : $(VP \rightarrow VP) \rightarrow VP \rightarrow VP$</td>
<td></td>
</tr>
<tr>
<td>$\equiv$ $\text{derived trees } \gamma_{\text{usually}}$ $\quad = \lambda^o a \rightarrow a (VP_2 (\text{Adv}_1 \text{ usually}) \rightarrow VP)$</td>
<td></td>
</tr>
<tr>
<td>$\gamma_{\text{hence}}$ : $(S \rightarrow S) \rightarrow S \rightarrow S$</td>
<td></td>
</tr>
<tr>
<td>$\equiv$ $\text{derived trees } \gamma_{\text{hence}}$ $\quad = \lambda^o a \rightarrow a (S \rightarrow (\text{Adv}_1 \text{ hence}) \rightarrow S)$</td>
<td></td>
</tr>
<tr>
<td>$I_{S}$ : $S \rightarrow S$</td>
<td></td>
</tr>
<tr>
<td>$\equiv$ $\text{derived trees } I$ $\quad = \lambda^o x . x : T \rightarrow T$</td>
<td></td>
</tr>
<tr>
<td>$I_{VP}$ : $VP \rightarrow VP$</td>
<td></td>
</tr>
<tr>
<td>$\equiv$ $\text{derived trees } I$ $\quad = \lambda^o x . x : T \rightarrow T$</td>
<td></td>
</tr>
</tbody>
</table>

In Section 4.2, we noticed that $\gamma_{13} = \gamma_{\text{seemingly}} I$ $\gamma_{\text{John}} : T \in \otimes(\mathcal{O}(\mathcal{G}_{\text{yield}}))$ (see Equation (9) on page 552). By definition of the object language of an ACG, its interpretation $\mathcal{O}(\mathcal{G}_{\text{yield}}(\gamma_{13}) = \text{seemingly + John}$ is such that $\mathcal{O}(\mathcal{G}_{\text{yield}}(\gamma_{13}) \in \otimes(\mathcal{O}(\mathcal{G}_{\text{yield}}))$. 
However, $\gamma_{13} \notin \Theta(\mathcal{G}_{\text{derived trees}})$. Indeed, there is no $c_{13}$ such that $\mathcal{G}_{\text{derived trees}}(c_{13}) = \gamma_{13}$. A simple argument using the linearity of the interpretation shows that only $c_{\text{seemingly}}$ (once and only once), $c_{\text{John}}$ (once and only once), and $I_X$ can be used. But $c_{\text{John}}$ can not combine with any of the other terms (none of them use the type $\text{NP}$). Consequently, $\text{seemingly} + \text{John} \notin \Theta(\mathcal{G}_{\text{yield}} \circ \mathcal{G}_{\text{derived trees}})$, as is expected from the TAG grammar.

4.3.3 $\mathcal{G}_{\text{derived trees}}$ abstract terms and generalized derivation trees

It is interesting to note that abstract terms of $\mathcal{G}_{\text{derived trees}}$ describe the way the encoding of trees in $\Sigma_{\text{trees}}$ can combine. We can see this combination in terms such as $\gamma_5 = \gamma_{\text{sleeps} \ I \ (\gamma_{\text{seemingly}} \ I) \ \gamma_{\text{John}}}$, but it is in some sense an artifact of the definition we gave: $\gamma_5$ $\beta$-reduces to a tree that does not show this structure anymore. However, a term such as $c_5 = c_{\text{sleeps} \ I_S \ (c_{\text{seemingly}} \ I_{VP}) \ c_{\text{John}}}$ does not further $\beta$-reduce. Because we considered substitution as function application on arguments of atomic types and adjunction as function application on arguments of second-order types, $c_5$ keeps track of the adjunction of $I_S$ on $c_{\text{sleeps}}$, of the adjunction of $I_{VP}$ on $c_{\text{seemingly}}$, of the adjunction of the latter result on $c_{\text{sleeps}}$, and of the substitution of $c_{\text{John}}$. And the relation $\mathcal{G}_{\text{derived trees}}(c_5) = \gamma_5$ expresses the relation between the record of these operations and the resulting derived tree.

We can represent $c_5$ as a tree (see Figure 14(a)): each node corresponds to a constant, applied to the terms represented by the children of the node. It makes explicit how similar to TAG derivation trees they are (Figure 14(b)). There is a one to one correspondence despite the following superficial differences:

- in the abstract term representation, the fake adjunctions (of $I_X$) are represented;

![Figure 14: Derivation representations](image)

(a) Abstract term of $\mathcal{G}_{\text{derived trees}}$  
(b) TAG derivation tree
• instead of specifying the role of the arguments with the Gorn address, we set a specific order for the arguments.

All the objects of a TAG grammar now have an ACG counterpart:
• terms of the abstract language of \( G \) derived trees correspond to the TAG derivation trees;
• terms of \( \Lambda(\Sigma_{trees}) \) that are in the object language of \( G \) derived trees correspond to the TAG derived trees;
• terms of \( \Lambda(\Sigma_{strings}) \) that are in the object language of \( G \) yield \( G \) derived trees correspond to the TAG generated language.

(12) and (13) illustrate these correspondences for the abstract term \( c_5 = c_{sleeps} I_5 (c_{seemingly} I_{VP}) c_{john} \) representing the derivation for the analysis of John seemingly sleeps.

\[
\begin{align*}
G_{\text{derived trees}}(c_{sleeps} I_5 (c_{seemingly} I_{VP}) c_{john}) \\
= \gamma_{sleeps} I (\gamma_{seemingly} I) \gamma_{john} \\
= S_2 (NP_1 John) (VP_2 (Adv_1 seemingly) (VP_1 (V_1 sleeps))) \\
\text{by (5), p. 545}
\end{align*}
\]

(13) \( G_{\text{yield}} \circ G_{\text{derived trees}}(c_{sleeps} I_5 (c_{seemingly} I_{VP}) c_{john}) \)

= John + (seemingly + sleeps) \quad \text{by (12) and (8) (p. 552)}

Remark 7 \((G_{\text{derived trees}} \text{ terms and description of trees})\). Let us have a look at the \((X \rightarrow X)\) type of the argument of an abstract constant of \( G_{\text{derived trees}} \) and at its interpretation. In \( c_{sleeps} \) for instance, the argument with type \((VP \rightarrow VP)\) is interpreted by the \( a \) variable of \( \gamma_{sleeps} \) (see Table 4 on page 554). The position of \( a \) in the term \( S (S_2 s (a (VP_1 (V_1 sleeps)))) \) makes it explicit that the result of \( a \) applied to \((VP_1 (V_1 sleeps))\), hence the latter term itself, is dominated by the second child of \( S_2 \) (the variable \( s \) being the first one). So, in some sense, the type \((VP \rightarrow VP)\) of \( a \) corresponds to the dominance constraint between the node where \( a \) occurs (here the second child of \( S_2 \)) and the root node of its argument (here \( VP_1 (V_1 sleeps) \)), as in the tree descriptions of Vijay-Shanker (1992): the root node of the argument of \( a \) is always dominated by the node where \( a \) occurs. In particular, replacing \( a \) by \( \lambda^o x.x \) corresponds to having these two nodes identified.

\[20\] There remain some reservations that Section 6 clears up, though. See Remark 8.
**Remark 8** (Generalized derivations and TAG derivation trees). It should be noted that \( G_{\text{derived trees}} \) and \( G_{\text{yield}} \circ G_{\text{derived trees}} \) are not second-order ACGs. It means that the polynomial parsing results do not directly apply. But we know that TAG parsing is polynomial. So what is happening here?

The answer is that while \( G_{\text{derived trees}} \) constrains the string language more than \( G_{\text{yield}} \) does, it does not constrain it enough to generate only the corresponding TAG language. There is indeed no difference between the type of the encoding of an elementary tree of root \( s \) with a substitution node \( s \) and the encoding of an auxiliary tree with root and foot nodes labelled by \( s \): it is \( s \rightarrow s \) in both cases.

The solution, that we develop in Section 6, is to further control \( \mathcal{A}(G_{\text{derived trees}}) \) with another ACG \( G_{\text{TAG}} \) such that \( \mathcal{A}(G_{\text{TAG}}) \subset \mathcal{A}(G_{\text{derived trees}}) \) just as \( G_{\text{derived trees}} \) allows us to control \( \mathcal{A}(G_{\text{yield}}) \). The general architecture is then the one that Figure 15 describes.

But while this additional control is necessary to have a faithful encoding of TAG, it is not necessary to provide the semantic interpretation of the derivation trees (and may somewhat obfuscate it). That is why we first present the syntax-semantic interface we propose and delay the final encoding of TAG (that corresponds to the one of de Groote 2002) to Section 6.

---

5 **SEMANTIC CONSTRUCTION**

In the previous section, we defined a signature \( \Sigma_{\text{derivations}} \) to represent derivation structures as terms of \( \Lambda(\Sigma_{\text{derivations}}) \). We now use this signature as a pivot to transfer these structures into semantic representations. From a practical point of view, as mentioned in Section 3.1, it
amounts to defining an ACG $\mathcal{G}_{\text{sem.}} = \langle \Sigma_{\text{derivations}}, \Sigma_{\text{logic}}, \mathcal{L}_{\text{sem.}}, s \rangle$ and composing it with $\mathcal{G}_{\text{derived trees}}$ thanks to the shared abstract vocabulary $\Sigma_{\text{derivations}}$. The object vocabulary $\Sigma_{\text{logic}}$ of this ACG is the vocabulary for defining the semantic representations. In this article, we use higher-order logic (and, more often than not, simply first-order logic). Other languages, such as description languages to express underspecified representations (Bos 1995; Egg et al. 2001), modal logic languages, etc. are possible as well. But we want to focus on how to build semantic representations rather than on the semantic modelling of some linguistic phenomenon itself.

5.1 A vocabulary for semantic representations: $\Sigma_{\text{logic}}$

We first define the object vocabulary $\Sigma_{\text{logic}} = \langle A_{\text{logic}}, C_{\text{logic}}, \tau_{\text{logic}} \rangle$ as in Table 5 with $A_{\text{logic}} = \{e, t\}$ the atomic types for entities and truth values respectively. As usual, we write the $\lambda$-term $\exists (\lambda x.P)$ as $\exists x.P$. The same, *mutatis mutandis*, holds for $\forall$. Note that, in this signature, we also use the non-linear implication, as a lot of semantic formulas (e.g., adjectives and quantifiers) use non linearity of entities. But we stay within the fragment of almost linear terms as only terms of atomic type are duplicated (see Definition 5 on page 537).21

| Logical constants | $\land : t \rightarrow t \rightarrow t$ | $\lor : t \rightarrow t \rightarrow t$ |
| Non-logical constants | $\Rightarrow : t \rightarrow t \rightarrow t$ | $\neg : t \rightarrow t$ |
| | $\exists : (e \rightarrow t) \rightarrow t$ | $\forall : (e \rightarrow t) \rightarrow t$ |

5.2 Generalized derivation-based interpretation

The first step in defining $\mathcal{G}_{\text{sem.}}$, to interpret the abstract vocabulary $\Sigma_{\text{derivations}}$ into types and terms built on the object vocabulary $\Sigma_{\text{logic}},$

21 For the sake of simplicity, we use simplified extensional types $e$ and $t$. A more accurate semantic would require, for instance, intensional types.
A syntax-semantics interface for TAG through ACG

Table 6:
Interpretation by $\mathcal{G}_{\text{sem.}}$ of the $\Sigma_{\text{derivations}}$ vocabulary

(a) Interpretation of the atomic types

$$
\begin{align*}
S & :=_{\text{sem.}} e \\
\text{NP} & :=_{\text{sem.}} (e \rightarrow t) \rightarrow t \\
\text{N} & :=_{\text{sem.}} e \rightarrow t \\
\text{VP} & :=_{\text{sem.}} e \rightarrow t \\
\text{WH} & :=_{\text{sem.}} (e \rightarrow t) \rightarrow t
\end{align*}
$$

(b) Interpretation of the constants

$$
\begin{align*}
c_{\text{John}} & :=_{\text{sem.}} \lambda^0 P P \text{John} \\
c_{\text{sleeps}} & :=_{\text{sem.}} \lambda^0 \text{adv}_s \text{adv}_v \text{subj.} \text{adv}_s (\text{subj} (\text{adv}_v (\lambda x. \text{sleep} x))) \\
c_{\text{seemingly}} & :=_{\text{sem.}} \lambda^0 \text{adv}_v \text{pred.} \text{adv}_v (\lambda x. \text{seemingly} (\text{pred} x)) \\
c_{\text{usually}} & :=_{\text{sem.}} \lambda^0 \text{adv}_v \text{pred.} \text{adv}_v (\lambda x. \text{usually} (\text{pred} x)) \\
c_{\text{hence}} & :=_{\text{sem.}} \lambda^0 \text{adv}_v \text{pred.} \text{adv}_v (\text{hence pred}) \\
I_S & :=_{\text{sem.}} \lambda^0 x. x \\
I_{\text{VP}} & :=_{\text{sem.}} \lambda^0 x. x
\end{align*}
$$

is to define the interpretation of the atomic types ($S$, $VP$...). We simply follow the standard interpretation of these syntactic types into the semantic types as proposed by Montague (1973). This results in the interpretation described in Table 6(a). The interpretation of the constants follows, as in Table 6(b). We do not repeat here the type of the constants $c_{\text{anchor}}$ of $\Sigma_{\text{derivations}}$, nor the constraint that the image of this type has to be the type of $\mathcal{G}_{\text{sem.}}(c_{\text{anchor}})$ (e.g., the type of $c_{\text{John}}$ is NP, hence $\mathcal{G}_{\text{sem.}}(c_{\text{John}}) : \mathcal{G}_{\text{sem.}}(\text{NP}) = (e \rightarrow t) \rightarrow t$). But the reader can check that this proviso holds.

We let the reader check that for our favourite example, $c_5 :=_{\text{sem.}} \text{seemingly} (\text{sleep John})$ as (14) shows.

$$
(14) \quad c_5 = c_{\text{sleeps}} I_S (c_{\text{seemingly}} I_{\text{VP}}) c_{\text{John}} :=_{\text{sem.}} \text{seemingly} (\text{sleep John})
$$

5.3 From derivation dependencies to semantic dependencies

We now turn to accounting for the mismatch between the dependencies as expressed in the derivation trees and in the logical semantic representations.

---

22The types now also use the intuitionistic implication $\rightarrow$. This is required when variables that are abstracted over appear more than once in the semantic recipes. This is in particular the case for entities in quantified formulas, or in the semantics of intersective adjectives (see next section).
5.3.1 Long-distance dependencies

The first mismatch we consider, in order to make explicit what exactly this mismatch refers to, relates to the classical examples (15–17).

(15) Paul claims John loves Mary.
(16) Mary, Paul claims John seems to love.
(17) Who does Peter think Paul claims John seems to love?

The TAG analysis relies on the elementary trees of Figure 16 and results in the derived tree and derivation tree of Figure 17 (p. 561) for (15). The mismatch appears in the contrast between the derivation tree where \( \alpha_{\text{loves}} \) scopes over \( \beta_{\text{claims}} \) whereas the opposite scoping is to be expected from a semantic point of view. A similar effect occurs with (16) as the derivation tree, shown in Figure 18(b) (p. 561), makes \( \alpha_{\text{to love}} \) scope over both \( \beta_{\text{claims}} \) and \( \beta_{\text{seems}} \), while semantically both...
should scope over the love predicate. Moreover, the derivation tree does not specify any scoping relation between the two auxiliary trees, whereas we expect claim to semantically scope over seem.

Finally, (17) and the derivation tree of Figure 19(b) (p. 562) illustrate how an element such as a wh-word can scope over a whole sentence and all its predicates while providing a semantic argument to the semantically “lowest” predicate (love).

To semantically account for these phenomena, we first extend $\Sigma_{\text{derivations}}$ and $\mathcal{G}_{\text{derived trees}}$ to represent the trees of Figure 16. Table 7 (p. 563) shows the new constants and their interpretations. The terms $c_{17}$, $c_{18}$, and $c_{19}$ in (18) represent the derivation trees of

\[ c_{17}, c_{18}, c_{19} \]
We now need to define the $g_{sem.}$ interpretation that provides the expected semantic dependencies. Table 8 (p. 563) shows the lexical semantics fulfilling the requirements. The constant $c_{loves}$ scopes over $c_{claims}$ in the term $c_{17} = c_{loves} (c_{claims} I_{S} I_{VP} c_{Paul}) I_{VP} c_{John} c_{Mary}$, as does $\alpha_{loves}$ over $\alpha_{claims}$ in the derivation tree of Figure 17(b). However, looking at $g_{sem.}(c_{loves})$ in Table 8, we observe that its first argument $adv_s$ scopes over the $love$ predicate. This argument actually corresponds to the meaning of the auxiliary tree adjoined at the $S$ node of $\alpha_{loves}$. When it is replaced by some actual value,
A syntax-semantics interface for TAG through ACG

\[ c_{who} := \text{NP} \]
\[ \gamma_{who} \triangleq \text{NP}_1 \text{ who } : T \]
\[ c_{loves} := (S \rightarrow (VP \rightarrow VP) \rightarrow NP \rightarrow NP \rightarrow S) \]
\[ \gamma_{loves} \triangleq \lambda^0 (a \rightarrow s \rightarrow s_2 \rightarrow S ((S_2 \rightarrow (a \rightarrow (VP_2 \rightarrow (V_1 \rightarrow (V_1 \rightarrow love))))))) \]
\[ c_{to love} := (S \rightarrow (VP \rightarrow VP) \rightarrow NP \rightarrow NP \rightarrow S) \]
\[ \gamma_{to love} \triangleq \lambda^0 (a \rightarrow s \rightarrow s_2 \rightarrow o \rightarrow S ((S_2 \rightarrow (a \rightarrow (VP_2 \rightarrow (V_1 \rightarrow to) \rightarrow (VP_1 \rightarrow (V_1 \rightarrow love))))))) \]
\[ c_{to love?} := (S \rightarrow (VP \rightarrow VP) \rightarrow NP \rightarrow NP \rightarrow S) \]
\[ \gamma_{to love?} \triangleq \lambda^0 (a \rightarrow s \rightarrow s_2 \rightarrow S ((S_2 \rightarrow (a \rightarrow (VP_2 \rightarrow (V_1 \rightarrow think)))))) \]
\[ c_{claims} := (S \rightarrow (VP \rightarrow VP) \rightarrow NP \rightarrow NP \rightarrow S) \]
\[ \gamma_{claims} \triangleq \lambda^0 (a \rightarrow s \rightarrow s_2 \rightarrow c \rightarrow S ((S_2 \rightarrow (a \rightarrow (VP_2 \rightarrow (V_1 \rightarrow claim)))))) \]
\[ c_{seems} := (VP \rightarrow VP) \rightarrow (VP \rightarrow VP) \]
\[ \gamma_{seems} \triangleq \lambda^0 (a \rightarrow s \rightarrow s_2 \rightarrow c \rightarrow S ((S_2 \rightarrow (a \rightarrow (VP_2 \rightarrow (V_1 \rightarrow seem)))))) \]
\[ c_{does think} := (S \rightarrow (VP \rightarrow VP) \rightarrow NP \rightarrow S) \]
\[ \gamma_{does think} \triangleq \lambda^0 (a \rightarrow s \rightarrow s_2 \rightarrow c \rightarrow S ((S_2 \rightarrow (a \rightarrow (VP_2 \rightarrow (V_1 \rightarrow think)))))) \]

Table 7: Interpretation by \( \mathcal{G}_{\text{derived trees}} \)

<table>
<thead>
<tr>
<th>derivation</th>
<th>sem. Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_{who} := \text{sem. } \lambda^0 \text{P.WHO} P</td>
<td></td>
</tr>
<tr>
<td>c_{loves} := \text{sem. } \lambda^0 \text{adv}_s \text{ adv}_v \text{ subj obj.adv}_s \text{ subj (adv}_v \text{ (adv}_v \text{ (x.obj (y.love x y))))}</td>
<td></td>
</tr>
<tr>
<td>c_{to love} := \text{sem. } \lambda^0 \text{adv}_s \text{ adv}_v \text{ subj obj.adv}_s \text{ subj (adv}_v \text{ (adv}_v \text{ (x.obj (y.love x y))))}</td>
<td></td>
</tr>
<tr>
<td>c_{to love?} := \text{sem. } \lambda^0 \text{adv}_s \text{ adv}_v \text{ wh subj.wh (adv}_v \text{ (subj (adv}_v \text{ (adv}_v \text{ (x.love x y))))}</td>
<td></td>
</tr>
<tr>
<td>c_{claims} := \text{sem. } \lambda^0 \text{adv}_s \text{ adv}_v \text{ subj comp.adv}_s \text{ subj (adv}_v \text{ (adv}_v \text{ (x.claim x comp))))}</td>
<td></td>
</tr>
<tr>
<td>c_{seems} := \text{sem. } \lambda^0 \text{mod pred.mod (x.seem x pred)</td>
<td></td>
</tr>
<tr>
<td>c_{does think} := \text{sem. } \lambda^0 \text{adv}_s \text{ adv}_v \text{ subj comp.adv}_s \text{ subj (adv}_v \text{ (adv}_v \text{ (x.think x comp))))}</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Interpretation by \( \mathcal{G}_{\text{sem.}} \) of the \( \Sigma_{\text{derivations}} \) vocabulary – long distance dependencies

[ 563 ]
for instance by the interpretation of \(c_{\text{claims}} I_S I_{VP} c_{Paul}\), the predicate in this actual value (here claim) then takes scope over love, achieving the desired effect. The same holds for the \(adv_{VP}\) argument.

However, in \(\mathcal{G}_{sem.}(c_{to\ love})\), the \(wh\) argument takes scope over the whole interpretation. This argument corresponds to the meaning of the constituent to be substituted at the \(WH\) node of \(\alpha_{to\ love}\) (see Figure 16), typically \(\lambda^x P . WHO P : (e \rightarrow t) \rightarrow t\), making \(WHO\) eventually take scope over all the other predicates.

Equation (19) shows that the \(\mathcal{G}_{sem.}\) interpretation builds the expected semantics with the required scope inversions. In terms of lexical semantics, the analysis and the account we propose are very close to the one proposed in synchronous TAG (Nesson 2009, p. 142).

\[
(19)
\]

For instance, \(c_{loves}\), \(c_{to\ love}\), and \(c_{to\ love?}\) are very close to each other. Building large scale grammars would require some factoring as can be done by lexical rules or meta-grammars (Candito 1996, 1999; Xia 2001; Xia et al. 2005; Crabbé et al. 2013). But in this article, we give the terms corresponding to the actual elementary trees that would be generated.

5.3.2 Quantification

We address in a similar way the mismatch between the scoping relation of verbal predicates over quantifiers in derivation trees and of quantifiers over verbal predicates in the logical semantic formulas. The trees of Figure 20 provide the TAG elementary trees for the TAG analysis for (20) (resp. for (21)) shown in Figure 21 (resp. in Figure 22).
(20) Everyone loves someone.
(21) Every man loves some woman.

\[
\begin{align*}
\alpha_{\text{someone}} &= \text{NP} \quad \alpha_{\text{everyone}} &= \text{NP} \\
\beta_{\text{some}} &= \text{NP} \quad \beta_{\text{every}} &= \text{NP} \\
\beta_{\text{some}} &= \text{Det} \quad N^* \\
\beta_{\text{every}} &= \text{Det} \quad N^* \\
\text{everyone} &\quad \text{loves} \quad \text{someone} \\
\text{man} &\quad \text{loves} \quad \text{woman}
\end{align*}
\]

Remark 10. We follow the standard TAG analyses for determiners (Abeillé 1990, 1993; XTAG Research Group 2001) where the latter adjoin on initial trees anchored by nouns, in order, in particular, to account for sequences of determiners (e.g., all these ideas) and mass nouns. While the auxiliary trees \(\beta_{\text{some}}\) and \(\beta_{\text{every}}\) of Figure 20 look unusual because the root node and the foot node do not have the same label, we can consider the label \(\text{NP}\) as a shorthand for the \(\text{NP}\) TAG category together with a positive \((\text{NP}[+])\) determiner feature. On the
other hand, the $N$ label is a shorthand for the $NP$ TAG category together with a negative ($NP[-]$) determiner feature. Kasper et al. (1995) and others (Rogers 1999; Kahane et al. 2000) already noted that the differences between the features on the root node and on the foot node could be reflected in allowing auxiliary trees to have different labels as root and foot nodes. While we discuss the modelling of features in TAG more generally in Section 7, the $NP$ and $N$ notations allow us to model the auxiliary trees of determiners with constants of the usual $N \rightarrow NP$ type (to be compared with a $NP[-] \rightarrow NP[+]$ type). While we could avoid introducing this distinction on the syntactic part of the TAG modelling, and have every node labelled with $N$, this distinction is semantically meaningful and records the different interpretation of $N$ (as $e \rightarrow t$) and $NP$ (as $(e \rightarrow t) \rightarrow t$) (see Table 6(a) on page 559).

The type of the constants modelling initial trees anchored by nouns has to be modified accordingly: it specifies that it requires an adjunction (an argument of type $(N \rightarrow NP)$) before turning the noun into a noun phrase $NP$. So the type of constants (e.g., $c_{man}$) modelling initial trees anchored by nouns (e.g., $a_{man}$) is: $(N \rightarrow NP) \rightarrow NP$. It corresponds to only keeping the constants that can indeed be used in actual derivations. For each noun, we could instead introduce two constants with the following types: $(NP[-] \rightarrow NP[-]) \rightarrow NP[-] = (N \rightarrow N) \rightarrow N$ and $(NP[-] \rightarrow NP[+]) \rightarrow NP[+] = (N \rightarrow NP) \rightarrow NP$, but since there is no other constant that uses $NP[-] = N$ as a type for its arguments (i.e., substitution nodes), we only keep the constant with the last type.

The derivation trees of Figure 21 and 22 are again such that the elementary tree of the verb predicate dominates the other elementary trees, while the respective scopes of their semantic contributions are in the reverse order. To show how this apparent mismatch can be dealt with, we extend $\Sigma_{derivations}$ with the constants of Table 9. This table also provides the interpretation of these constants by $\langle G_{derived trees} \rangle$, modelling the elementary trees of Figure 20. The terms of (22) belong to $\langle G_{derived trees} \rangle$ and represent the derivation trees of Figure 21.

\[ 24 \] This of course depends on the grammar. In any case, if there were such a constant, it would not allow for performing first the adjunction of a determiner on the noun.
A syntax-semantics interface for TAG through ACG

<table>
<thead>
<tr>
<th>Constants of $\Sigma_{derivations}$</th>
<th>Interpretation by $\mathcal{G}_{derived}$ trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{\text{man}}$ : $(N \rightarrow \text{NP}) \rightarrow \text{NP}$</td>
<td>$\lambda^n. d. (N_1 \text{man})$</td>
</tr>
<tr>
<td>$c_{\text{someone}}$ : NP</td>
<td>NP$_1$ someone</td>
</tr>
<tr>
<td>$c_{\text{everyone}}$ : NP</td>
<td>NP$_1$ everyone</td>
</tr>
<tr>
<td>$c_{\text{some}}$ : $N \rightarrow \text{NP}$</td>
<td>$\lambda^n. n. \text{NP}_2 (\text{Det}_1 \text{some}) n$</td>
</tr>
<tr>
<td>$c_{\text{every}}$ : $N \rightarrow \text{NP}$</td>
<td>$\lambda^n. n. \text{NP}_2 (\text{Det}_1 \text{every}) n$</td>
</tr>
</tbody>
</table>

Table 9: Interpretation of $\Sigma_{derivations}$ constants by $\mathcal{G}_{derived}$ trees

and 22. Equation (23) shows they are interpreted as the derived trees of the same figures.

$$c_{21} = c_{\text{loves}} I_S I_{VP} c_{\text{everyone}} c_{\text{someone}}$$

$$c_{22} = c_{\text{loves}} I_S I_{VP} (c_{\text{man}} c_{\text{every}}) (c_{\text{woman}} c_{\text{some}})$$

(23) $c_{21} : \text{derived trees} S_2 (\text{NP}_1 \text{everyone}) (\text{VP}_2 (V_1 \text{loves}) (\text{NP}_1 \text{someone}))$

$$c_{22} : \text{derived trees} S_2 (\text{NP}_2 (\text{Det}_1 \text{every}) (N_1 \text{man}))$$

$$(\text{VP}_2 (V_1 \text{loves}) (\text{NP}_2 (\text{Det}_1 \text{some}) (N_1 \text{woman})))$$

Then we extend $\mathcal{G}_{sem}$. with the interpretations of these new constants of $\Sigma_{derivations}$ as terms of $\Lambda(\Sigma_{logic})$ (Table 10). The semantic interpretations of the terms $c_{21}$ and $c_{22}$ are then as expected, as (24) shows.

$$c_{21} : \equiv_{sem.} \forall x. (\text{human} x) \Rightarrow (\exists y. (\text{human} y) \land (\text{love} x y))$$

$$c_{22} : \equiv_{sem.} \forall x. (\text{man} x) \Rightarrow (\exists y. (\text{woman} y) \land (\text{love} x y))$$

(24) $c_{21} : \equiv_{sem.} \forall x. (\text{human} x) \Rightarrow (\exists y. (\text{human} y) \land (\text{love} x y))$

$$(\text{man} x \Rightarrow (\exists y. (\text{woman} y) \land (\text{love} x y)))$$

Table 10: Interpretation by $\mathcal{G}_{sem.}$ of the $\Sigma_{derivations}$ vocabulary – quantification

This shows how to use the derivation tree as a pivot towards the semantic representation of an expression. The (lexical) semantic interpretation of the terms labelling the nodes of the derivation tree encodes, when necessary, the inversion of the scope of the elements. This is reminiscent of the transformation of derivation trees.
into semantic dependency graphs of Candito and Kahane (1998) or Kallmeyer and Kuhlmann (2012). To this end, the latter implements a tree transduction-based approach (macro-tree transduction). Maskharashvili and Pogodalla (2013) discuss the relation with the present approach, relying on the encoding of macro-tree transduction within second-order ACGs (Yoshinaka 2006).

**Remark 11.** There are several ways to get the object scope reading. So far, the relative scopes of the subject and the object are bound to the semantic interpretation of the verb (see the semantic interpretation of $c_{\text{loves}}$ in Table 8 on page 563). So a possibility consists in introducing a new constant $c_{\text{ows}}^{\text{loves}}$ whose semantic interpretation reverses the scope, as Equation (25) shows.

$$c_{\text{ows}}^{\text{loves}} := \text{sem.}$$

$$\lambda^o \text{adv}_5 \text{adv}_{vp} \text{subj} \text{obj}.\text{adv}_5 (\text{obj} (\lambda y. \text{subj} (\text{adv}_{vp} (\lambda x. \text{love} x y))))$$

Another possibility, that would go beyond what is introduced in this article, would be to use Multi-Component TAG (MCTAG: Weir 1988) as Williford (1993) proposes. In both cases, some care should be taken in order not to introduce spurious ambiguities. In Section 8.3, we provide another modelling that allows us to get this reading, and we relate it to other approaches.

5.3.3 Multiple adjunctions

The representation of TAG derivation trees as abstract terms of an ACG corresponds to the standard notion of derivation trees (Vijay-Shanker 1987). Schabes and Shieber (1994) call it dependent and advocate for an alternative independent notion. With dependent derivations, and in our approach, multiple adjunction on the same node is forbidden. So the analysis of (26), using the trees of Figure 23, requires first the adjunction of $\beta_{\text{big}}$ into $\beta_{\text{black}}$, and the adjunction of the result into $\alpha_{\text{dog}}$. Figure 25(a) shows the resulting derivation tree. On the other hand, the independent adjunction shown in Figure 25(b) only specifies that both adjectives adjoin at the N node of the initial tree, corresponding to both the derived tree of Figure 24(a) and the derived tree of Figure 24(b).

(26) big black dog
(27) black big dog
Schabes and Shieber (1994) present several arguments in favour of multiple adjunction for auxiliary trees encoding modification (as opposed to auxiliary trees encoding predication) and independent derivations. We only discuss here the semantic argument they provide.\textsuperscript{25} The main concern again has to do with the relation between derivation trees and semantic dependencies. The dependent derivation of Figure 25(a) reflects “cascaded modifications” of the head, rather

\textsuperscript{25}The two other main arguments relate to the addition of adjoining constraints and to the addition of statistical parameters. Adding the latter to ACGs as a general framework is ongoing work, and the effects on the particular case of the TAG into ACG encoding will be considered from this perspective (Huot 2017). The argument about adjoining constraints that fail to escape intervening adjunctions is not related to the syntax-semantics interface and deserves a discussion that is beyond the scope of this article. For instance, the example of the [+] determiner feature (Section 5.3.2) that can percolate from the determiner (outermost adjunction) to the noun, despite the intervening adjunctions of the adjectives, shows that selectional restrictions can be implemented with long-distance effects.
than more expected “separate modifications”, the latter being only available through multiple adjunction. We show that we can actually achieve this effect in our framework, without multiple adjunction, by specifying a semantic interpretation for adjectives that encodes such a behavior.

We consider the extension of $\Sigma_{\text{derivations}}$ with the constants of Table 11 and their interpretations by $G_{\text{derived trees}}$ and $G_{\text{sem.}}$ of Table 12. The types of the the constants modelling adjectives follow the types proposed for constants modelling nouns. The modification they introduce builds a $NP$ from a $N$, and can itself take a $(N \to NP)$ modification (adjunction) into account. Consequently, they are of type $(N \to NP) \to N \to NP$. As we did for the types of the constants modelling nouns, we could enumerate the possible types taking the determiner feature into account. Because the adjunction of an adjective does not change the determiner feature, its value at the root node of the auxiliary tree only depends on what is possibly adjoined to it. So we could have four constants with the following types: $(NP[-] \to NP[-]) \to (NP[-] \to NP[-])$, $(NP[-] \to NP[+]) \to (NP[-] \to NP[+])$, $(NP[+] \to NP[-]) \to (NP[+] \to NP[-])$, and $(NP[+] \to NP[+]) \to (NP[+] \to NP[+])$. But if we do not provide a term for a fake adjunction $(NP[-] \to NP[-])$, $(NP[+] \to NP[+])$, or $(NP[+] \to NP[-])$ (as in the example grammar we have), such terms can never be used in a $S$ derivation. So we only keep the constants that have type $(NP[-] \to NP[+]) \to (NP[-] \to NP[+]) = (N \to NP) \to N \to NP$.

Equation (28) shows the interpretations of the term $c_{25} : (N \to NP) \to NP$ (an expression missing a determiner of type $(N \to NP)$ to provide a $NP$) that encodes the derivation tree of Figure 25(a) (p. 569).

Table 11: $\Sigma_{\text{derivations}}$ additional constants

| $c_{\text{big}}$ |
| $(N \to NP) \to N \to NP$ |
| $c_{\text{black}}$ |
| $(N \to NP) \to N \to NP$ |
| $c_{\text{dog}}$ |
| $(N \to NP) \to NP$ |

Table 12: Interpretation by $G_{\text{sem.}}$ and $G_{\text{derived trees}}$ of the $\Sigma_{\text{derivations}}$ vocabulary - multiple adjunction

$\begin{align*}
\text{$c_{\text{big}}$} & \coloneqq_{\text{derived trees}} \lambda^0 a \ n \ a \ (N_2 \ (\text{Adj}_1 \ \text{big}) \ n) \\
\text{$c_{\text{black}}$} & \coloneqq_{\text{derived trees}} \lambda^0 a \ n \ a \ (N_2 \ (\text{Adj}_1 \ \text{black}) \ n) \\
\text{$c_{\text{dog}}$} & \coloneqq_{\text{derived trees}} \lambda^0 d \ d \ (N_1 \ \text{dog}) \\
\text{$c_{\text{big}}$} & \coloneqq_{\text{sem.}} \lambda^0 Q \ n.\lambda^0 q.Q \ (\lambda x. (n \ x) \wedge (\text{big} \ x)) \ q \\
\text{$c_{\text{black}}$} & \coloneqq_{\text{sem.}} \lambda^0 Q \ n.\lambda^0 q.Q \ (\lambda x. (n \ x) \wedge (\text{black} \ x)) \ q \\
\text{$c_{\text{dog}}$} & \coloneqq_{\text{sem.}} \lambda^0 Q.\lambda^0 q.Q \ \text{dog} \ q
\end{align*}$
The interpretation by $G_{sem}$ indeed provides a separate modification of the same variable $x$ as argument both of big and black (a similar account would also be available in synchronous TAG).

$$c_{25} = \lambda^o D. c_{dog} (c_{black} (c_{big} D))$$

(28) $\vdash_{\text{derived trees}} \lambda^o D. D \ (N_2 \ (\text{Adj}_1 \ \text{big}) \ (N_2 \ (\text{Adj}_1 \ \text{black}) \ (N_1 \ \text{dog})))$

$\vdash_{\text{sem.}} \lambda^o D. \lambda^o q. D \ (\lambda x. ((\text{big} \ x) \land (\text{black} \ x)) \land (\text{dog} \ x)) \ q$

**Remark 12.** By not introducing a constant $I_N : N \rightarrow NP$ in $\Sigma_{derivations}$, we require actual adjunctions of determiners (of type $N \rightarrow NP$), e.g., $c_{some}$ on nouns or on nouns modified by adjectives.

### 6 COMPLETING THE TAG INTO ACG ENCODING

So far, the abstract signatures we used, in particular $\Sigma_{derivations}$, introduce constants that are of order strictly greater than 2. This comes in particular from the modelling of auxiliary trees as functions (typically of type $X \rightarrow X$), hence from having constants of higher-order type modelling the ability of a tree to take an auxiliary tree as an argument. From a theoretical point of view, we know this encoding cannot faithfully model TAG: TAG languages are polynomially parsable, and 3rd-order ACGs can generate languages in NP. Remark 8 (p. 557) gives an example of an unexpected result of this encoding: there is no way to distinguish the $S \rightarrow S$ type of an abstract constant modelling an auxiliary tree of foot node $S$ from an abstract constant modelling an elementary tree of root $S$ with a substitution node $S$.

#### 6.1 A vocabulary for TAG derivations: the $\Sigma_{TAG}$ signature

In order to allow for the distinction between these types, we introduce atomic types (e.g., $S_A$) that will be interpreted as the modifier types of the constants modelling auxiliary trees. So in addition to the ACG $G_{\text{derived trees}} = \langle \Sigma_{derivations}, \Sigma_{trees}, \mathcal{L}_{\text{derived trees}}, S \rangle$, we also define a higher-order signature $\Sigma_{TAG} = \langle A_{TAG}, C_{TAG}, \tau_{TAG} \rangle$ such that $A_{TAG} = A_{derivations} \cup \bigcup_{X \in \text{derivations}} X_A$.

For any $\delta_{anchor} \in \mathcal{I} \cup A$ an elementary tree of $G$, $C_{anchor}$ is a constant in $C_{TAG}$ with type $X^1_A \rightarrow \cdots \rightarrow X^n_A \rightarrow Y^1 \rightarrow \cdots \rightarrow Y^m \rightarrow \alpha$ where:

- the $X^i_A$ are the labels of the $n$ internal nodes of $\delta_{anchor}$ labelled with a non-terminal where an adjunction is possible (by convention we use the breadth-first traversal);
• the $Y^i$ are the labels of the $m$ leaves of $\delta_{anchor}$ labelled with non-terminals, not counting the foot node if $\delta_{anchor}$ is an auxiliary tree, of $\delta_{anchor}$ (by convention, we use the left-right order traversal);

• let $Z$ be the label of the root node of $\delta_{anchor}$. $\alpha = Z$ if $\delta_{anchor} \in \mathcal{I}$ is an initial tree, and $\alpha = Z_A$ with $Z$ the label of the root node if $\delta_{anchor} \in \mathcal{A}$ is an auxiliary tree.

From the elementary trees of Figure 2 (p. 533), for instance, we get the constants typed as (29) shows.

\[
C_{sleeps} : S_A \rightarrow VP_A \rightarrow NP \rightarrow S
\]
\[
C_{John} : NP
\]
\[
C_{seemingly} : VP_A \rightarrow VP_A
\]

Moreover, for each non-terminal $X$ of the TAG grammar where an adjunction can occur, we also define $I_X : X_A$. These constants play a similar role as the $I_X$ constants in $\Sigma_{derivations}$: they are used when a TAG derivation does not involve adjunctions on sites where it would be possible to have them.

Then the set of typed constants of $\Sigma_{TAG}$ is $C_{TAG} = \{C_{anchor} | \delta_{anchor} \in \mathcal{I} \cup \mathcal{A}\} \cup \{I_X | X \in V^0\}$ and $\tau_{TAG}$ is the associated typing function defined as above. The typing provided by $\Sigma_{TAG}$ now distinguishes the type of the encoding of an elementary tree of root $S$ with a substitution node $S$ (type $S \rightarrow S$) and the encoding of an auxiliary tree with root and foot nodes labelled by $S$ (type $S_A$, see Remark 8, p. 557).

We now need to relate the terms of $\Lambda(\Sigma_{TAG})$ to the terms of $\Lambda(\Sigma_{derivations})$ by a suitable interpretation.

### 6.2 Interpreting $\Sigma_{TAG}$ into $\Lambda(\Sigma_{derivations})$: the $\mathcal{G}_{TAG}$ ACG

We now can relate $\Sigma_{TAG}$ and $\Lambda(\Sigma_{derivations})$ through a new ACG $\mathcal{G}_{TAG} = \langle \Sigma_{TAG}, \Sigma_{derivations}, \mathcal{L}_{TAG}, \Lambda \rangle$ where $\mathcal{L}_{TAG}$ is such that:

• for all $\alpha \in A_{TAG}$, if $\alpha = X_A$ then $\mathcal{L}_{TAG}(\alpha) = \mathcal{L}_{TAG}(X_A) = X'' \rightarrow X'$ with, most of the time, $X = X' = X''$ (see footnote 18 (p. 553) and Remark 13, next page), otherwise $\alpha = X \in A_{derivations}$ and $\mathcal{L}_{TAG}(\alpha) = \mathcal{L}_{TAG}(X) = X$;

• for all $C_{anchor} \in C_{TAG}$, $\mathcal{L}_{TAG}(C_{anchor}) = c_{anchor}$. 


By construction of the constants $c_{\text{anchor}} \in C_{\text{derivations}}$ (Section 4.3), and by construction of the constants $C_{\text{anchor}} \in C_{\text{TAG}}$, $\mathcal{L}_{\text{TAG}}$ is well defined.

Table 13 sums up the constants corresponding to the elementary trees introduced so far as well as their interpretations. Because constants are interpreted as constants, the terms of $\Lambda(\Sigma_{\text{TAG}})$ and their interpretations are isomorphic. However, some terms of $\Lambda(\Sigma_{\text{derivations}})$ have no antecedent in $\mathcal{L}_{\text{TAG}}$. For instance, the term

<table>
<thead>
<tr>
<th>Types and constants of $\Sigma_{\text{TAG}}$</th>
<th>Their interpretations in $\Lambda(\Sigma_{\text{derivations}})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>VP</td>
<td>VP</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>WH</td>
<td>WH</td>
</tr>
<tr>
<td>VP$_A$</td>
<td>VP $\rightarrow$ VP</td>
</tr>
<tr>
<td>S$_A$</td>
<td>S $\rightarrow$ S</td>
</tr>
<tr>
<td>N$_A$</td>
<td>N $\rightarrow$ NP</td>
</tr>
<tr>
<td>$c_{\text{john}}$</td>
<td>$c_{\text{john}}$</td>
</tr>
<tr>
<td>$c_{\text{sleeps}}$</td>
<td>($S \rightarrow S$) $\rightarrow$ (VP $\rightarrow$ VP) $\rightarrow$ NP $\rightarrow$ S</td>
</tr>
<tr>
<td>$c_{\text{seemingly}}$</td>
<td>($VP \rightarrow VP$) $\rightarrow$ VP $\rightarrow$ VP</td>
</tr>
<tr>
<td>$c_{\text{usually}}$</td>
<td>($VP \rightarrow VP$) $\rightarrow$ VP $\rightarrow$ VP</td>
</tr>
<tr>
<td>$c_{\text{hence}}$</td>
<td>($S \rightarrow S$) $\rightarrow$ S $\rightarrow$ S</td>
</tr>
<tr>
<td>$I_S$</td>
<td>$I_S$</td>
</tr>
<tr>
<td>$I_{VP}$</td>
<td>$I_{VP}$</td>
</tr>
<tr>
<td>$c_{\text{matters}}$</td>
<td>($VP \rightarrow VP$) $\rightarrow$ S $\rightarrow$ S</td>
</tr>
<tr>
<td>$c_{\text{who}}$</td>
<td>$c_{\text{who}}$</td>
</tr>
<tr>
<td>$c_{\text{loves}}$</td>
<td>($S \rightarrow S$) $\rightarrow$ (VP $\rightarrow$ VP) $\rightarrow$ NP $\rightarrow$ NP $\rightarrow$ S</td>
</tr>
<tr>
<td>$c_{\text{to love}}$</td>
<td>($S \rightarrow S$) $\rightarrow$ (VP $\rightarrow$ VP) $\rightarrow$ NP $\rightarrow$ NP $\rightarrow$ S</td>
</tr>
<tr>
<td>$c_{\text{to love?}}$</td>
<td>($S \rightarrow S$) $\rightarrow$ (VP $\rightarrow$ VP) $\rightarrow$ NP $\rightarrow$ NP $\rightarrow$ S</td>
</tr>
<tr>
<td>$c_{\text{claims}}$</td>
<td>($S \rightarrow S$) $\rightarrow$ (VP $\rightarrow$ VP) $\rightarrow$ NP $\rightarrow$ (S $\rightarrow$ S)</td>
</tr>
<tr>
<td>$c_{\text{seems}}$</td>
<td>($VP \rightarrow VP$) $\rightarrow$ (VP $\rightarrow$ VP)</td>
</tr>
<tr>
<td>$c_{\text{does think}}$</td>
<td>($S \rightarrow S$) $\rightarrow$ (VP $\rightarrow$ VP) $\rightarrow$ NP $\rightarrow$ (S $\rightarrow$ S)</td>
</tr>
<tr>
<td>$c_{\text{man}}$</td>
<td>($N \rightarrow NP$) $\rightarrow$ NP</td>
</tr>
<tr>
<td>$c_{\text{someone}}$</td>
<td>$c_{\text{someone}}$</td>
</tr>
<tr>
<td>$c_{\text{everyone}}$</td>
<td>$c_{\text{everyone}}$</td>
</tr>
<tr>
<td>$c_{\text{some}}$</td>
<td>$c_{\text{some}}$</td>
</tr>
<tr>
<td>$c_{\text{every}}$</td>
<td>$c_{\text{every}}$</td>
</tr>
<tr>
<td>$c_{\text{big}}$</td>
<td>($N \rightarrow NP$) $\rightarrow$ N $\rightarrow$ NP</td>
</tr>
<tr>
<td>$c_{\text{black}}$</td>
<td>($N \rightarrow NP$) $\rightarrow$ N $\rightarrow$ NP</td>
</tr>
<tr>
<td>$c_{\text{dog}}$</td>
<td>($N \rightarrow NP$) $\rightarrow$ NP</td>
</tr>
</tbody>
</table>
c_sleeps (c_matters I_VP) I_VP c_john : S ∈ \mathcal{A}(\mathcal{G}_{\text{derived trees}}), where c_matters : (\text{VP} \to \text{VP}) \to S \to S corresponds to the initial tree \alpha_{matters} of Figure 26, as in *To arrive on time matters considerably* (see XTAG Research Group 2001, Section 6.31), has no antecedent. This is because the type of c_matters I_VP : S \to S in \mathcal{G}_{\text{derived trees}}, while it encodes an initial tree, is the same as the type of a term encoding an adjunction on a S node (see Remark 8 p. 557). But this is not true anymore at the level of \mathcal{G}_{\text{TAG}} where C_{matters} I_VP : S \to S but the type of a term encoding an adjunction on a S node is now S_A.

So \Sigma_{\text{TAG}} allows us to add control on the admissible derivation structures that \Sigma_{\text{derivations}} can provide. The general architecture is now the one of Figure 15 (p. 557). Moreover, this architecture allows us to provide, by function composition, a semantic interpretation to the constants of \Sigma_{\text{TAG}}. Interestingly, this semantic interpretation derives from the more general constructions that \Sigma_{\text{derivations}} enables.

\Sigma_{\text{TAG}} strictly follows the abstract signature definition proposed by de Groote (2002) to encode the syntactic part of TAGs. The correctness of our encoding follows from the fact that the ACG \mathcal{G}_{\text{derived trees}} \circ \mathcal{G}_{\text{TAG}} we get by function composition is the ACG defined by de Groote (2002).

**Remark 13.** Because the modelling of adjunction is now controlled by the interpretation of the types X_A from \Sigma_{\text{TAG}}, we see that we can have more freedom in the type that is given in \Sigma_{\text{derivations}}. For instance, we can set N_A := \text{TAG} N \to \text{NP}. We can use even more complex interpretations if it helps explaining the semantic interpretation. For instance, in Section 7.2 we introduce a type S'_A := \text{TAG} (\text{NP} \to S) \to S to model control verbs.

We can now consider the ACG \mathcal{G}_{\text{yield}} \circ \mathcal{G}_{\text{derived trees}} \circ \mathcal{G}_{\text{TAG}}, that interprets terms of \Lambda(\Sigma_{\text{TAG}}) into strings, and the ACG \mathcal{G}_{\text{sem.}} \circ \mathcal{G}_{\text{TAG}}, that interprets terms of \Lambda(\Sigma_{\text{TAG}}) into logical formulas (see Figure 15,
Because $\Sigma_{\text{TAG}}$ is second-order, these two ACGs are second-order (while $\mathcal{G}_{\text{yield}} \circ \mathcal{G}_{\text{derived trees}}$ and $\mathcal{G}_{\text{sem.}}$ are not, since $\Sigma_{\text{derivations}}$ is not second-order). Hence the parsing result applies and we may parse terms with ACGs that have $\Sigma_{\text{TAG}}$ as abstract vocabulary, in particular with the ACG toolkit. The ACG example files we provide can, for instance, parse the string every + big + black + dog + usually + barks. It can also parse the logical formula $\forall x . ((\text{dog} x) \land (\text{black} x)) \land (\text{big} x)) \Rightarrow (\text{usually} (\text{bark} x))$. Note that, as a $\lambda$-term, a logical formula can generally not be replaced by a logically equivalent formula. This is an instance of the problem of logical-form equivalence (Shieber 1993) that will need to be addressed, for instance using sets of $\lambda$-terms as input (Kanazawa 2017, Section 4.2). More examples are available in the example files.

7 ADJOINING CONSTRAINTS AND FEATURES

It is part of the TAG formalism to specify if an internal node may, may not, or must receive any adjunction. The latter case is called an obligat\-ory adjoining (OA) constraint. In case an internal node can be subject to an adjunction operation, it is also possible to specify a restricted set of auxiliary trees, with relevant root and foot nodes, that can adjoin. This constraint is called a selective adjoining (SA) constraint. There are different ways to specify such constraints in TAG. One is to add features to the formalism. ACGs do not provide a concise way to express the abstract representation of type constraints that features offer. There have been some proposals with dependent types (de Groote and Maarek 2007; de Groote et al. 2007; Pompigne 2013), but the underlying calculus does not have the expected good properties. So selection restriction has to be expressed by introducing as many types as necessary (see Section 5.3.2 for determiners and Section 5.3.3 for adjectives). To avoid the drawback of a growing size of the grammar, the addition of features, in particular morpho-syntactic ones, to ACGs remains desirable.

Note, however, that we do not want to consider features that are used to model the syntax-semantics interface (Kallmeyer and Romero 2004, 2008), since we use the interpretation of derivation trees instead. We discuss the relation between the two approaches in Section 7.3.
Section 5.3.3 presents an instance of an adjoining constraint, namely an obligatory adjoining constraint. In order to form an NP, a determiner of type \((N \rightarrow NP)\) needs to be adjoined (directly or through adjectival modifications) into a noun. The obligatory nature of the adjunction is reflected by the fact that the abstract vocabulary does not provide any constant \(I_N : N \rightarrow NP\) simulating a fake adjunction.

7.2 Selective adjoining constraints

In Section 5.3.2, we saw an instance of using features in a TAG analysis: noun phrases can receive a determiner feature \([+]\) or \([-]\) indicating whether they are determined. The ACG way to account for this distinction instead consists in introducing different (atomic) types. This corresponds to specifying local adjunction constraints by enumeration as in TAG and contrary to Feature-based TAG (FTAG: Vijay-Shanker and Joshi 1988, 1991).

As noticed in Remark 13 (p. 574), the key here is to model auxiliary trees using an atomic type \(X_A\), so that the ACG is second-order, and to interpret this type as a functional type \(X'' \rightarrow X'\) of \(\Sigma_{derivations}\), without the actual requirement that \(X''\) and \(X'\) are atomic or that \(X'' = X' = X\). We illustrate such an encoding with the TAG analysis of control verbs.

TAG analyzes a sentence such as (30) with an adjunction of the subject control verb \(\text{wants}\) on the reduced clause \(\text{to sleep}\) as Figure 28 shows. Figure 27 presents the elementary trees of the control verb and of the infinitive clause. This is similar to representing infinitive clauses as clauses without subjects (Abeillé 2002). For the sake of simplicity, we directly represent such a clause as an elementary tree with a \(\text{PRO}\) node.

Control and adjunction are enforced using a control feature on the \(s\) root node of the complement tree (\(\text{control}\) in XTAG (XTAG Research Group 2001, p. 98), or some semantic index \(\text{id}x\) (Gardent and Parmentier 2005; Gardent 2008)) that is to be provided by the foot node of the auxiliary tree (the control verb) which is adjoined. Moreover, in the auxiliary tree, the control feature on the foot node is co-indexed with a control feature on the subject \(NP\) (for subject control, as in (30)) or on the object \(NP\) (for object control). Figure 28 shows
A syntax-semantics interface for TAG through ACG

\[ \beta_{\text{John}} = \begin{array}{c} \text{NP} \left[ \text{idx} = j \right] \\ \text{John} \end{array} \]

\[ \beta_{\text{wants}} = \begin{array}{c} \text{NP} \left[ \text{idx} = x \right] \quad \text{VP} \\ \text{wants} \end{array} \]

\[ \alpha_{\text{to sleep}} = \begin{array}{c} \text{S} \left[ \text{idx} = y \right] \\ \text{NP} \quad \text{VP} \\ \text{to sleep} \end{array} \]

Figure 27: Elementary trees for control verbs

Figure 28: TAG analysis of control

(a) Derived tree

(b) Derivation tree

the derived tree, where, by unification of the top and bottom features, we eventually get \( x = y = j \), and the derivation tree for (30).

(30) John wants to sleep

In our ACG encoding, we model \( S \) nodes with a control feature with the functional type \( \text{NP} \to S \), expressing that such a clause is missing its subject. Consequently, the type of the constant \( c_{\text{wants}} \) that models the auxiliary tree \( \beta_{\text{wants}} \) is \( (S \to S) \to (VP \to VP) \to \text{NP} \to (\text{NP} \to S) \to S \). The end part \( (\text{NP} \to S) \to S \) of this type corresponds to the functional interpretation of the conjunction of control verbs, modelled at the \( \Sigma_{\text{TAG}} \) level with the atomic type \( S' \). The difference between the type \( (\text{NP} \to S) \) of the argument and the type \( S \) of the result corresponds to the different feature set attached to the root node and to the foot node of the auxiliary tree.

Then we model the feature sharing between the subject \( \text{NP} \) and the \( S \) foot node of the control verb in the semantic interpretation of the latter, as the second line of Table 16 shows: the first argument \( x \) of \( \text{want} \ x \ (\text{pred} \ (\lambda^o P. P \ x)) \) also appears (type raised as \( (\lambda^o P. P \ x) \) as
the argument of \textit{pred}, the latter corresponding to the semantics of the infinitive clause without subject.

Let $C_{28}$ of $\Lambda(\Sigma_{\text{TAG}})$ in (31) represent the derivation tree of Figure 28, and let $c_{28}$ be its interpretation in $\Lambda(\Sigma_{\text{derivations}})$. We can further interpret $c_{28}$ in $\Lambda(\Sigma_{\text{trees}})$ (resp. in $\Lambda(\Sigma_{\text{logic}})$) in order to have a term representing the associated derived tree (resp. semantics).

\begin{align*}
C_{28} &= C_{\text{to sleep}} (C_{\text{wants}} I_S I_{VP} C_{\text{John}}) \\
&:= \text{derivations } c_{28} \\
(31)
\end{align*}

\begin{align*}
c_{28} &= c_{\text{to sleep}} (C_{\text{wants}} I_S I_{VP} C_{\text{John}}) \\
&:= \text{derived trees } S_2 (NP_1 \text{ John}) (VP_2 (V_1 \text{ wants}) \\
&\quad (S_2 (NP_1 (PRO_1 e)) (VP_2 (V_1 to) (VP_1 \text{ sleep})))) \\
&:= \text{sem. want john (sleep john)}
\end{align*}

\section{Feature sharing and semantic computation}

As the previous section shows, features in TAG are taken into account in the ACG encoding using the typing discipline on the one hand, and using the (semantic) interpretation on the other hand, in particular when some value has to be shared in order to express the modifications performed by adjunction operations.

Unification based approaches to semantic construction in TAG typically rely on feature sharing (Gardent and Kallmeyer 2003; Gardent and Parmentier 2005; Kallmeyer and Romero 2008) in order to...
compositionally build the semantic representation of a sentence. In our approach, the semantic representation results from the interpretation of the derivation tree.

However, Vijay-Shanker and Joshi (1988, p. 718) already noticed that “[t]his treatment [of variable instantiation on adjunction] can be obtained if we think of the auxiliary tree as corresponding to functions over feature structures (by λ-abstracting the variable corresponding to the feature structure for the tree that will appear below the foot node). Adjunction corresponds to applying this function to the feature structure corresponding to the subtree below the node where [it] takes place”. This is precisely the view we adopt here. While the typing experts control over the admissible derivation structures, the associated computations are managed using interpretations, to compute the derived trees as well as the logical formulas.

8 DERIVATION TREES AND SEMANTIC INTERPRETATIONS

Looking at Figure 15 (p. 557), we can consider each of the sets of λ-terms as independent combinatorial systems of the grammar architecture that Jackendoff (2002) describes: “Language comprises a number of independent combinatorial systems which are aligned with each other by means of a collection of interface systems. Syntax is among the combinatorial systems, but far from the only one”.

Among those systems, \( \Lambda(\Sigma_{\text{derivations}}) \) and \( \Lambda(\Sigma_{\text{TAG}}) \) play a central role as their structures are the ones that are interpreted as derived trees (and as strings, by functional composition) and as logical formulas. This is not the role of the syntactic trees of \( \Lambda(\Sigma_{\text{trees}}) \). This emphasises that the relevant syntactic algebra to provide compositional analyses for TAG, as was noticed very early, is not the one of derived trees, but the one of derivation trees. This section further explores the modelling power it provides.

In particular, the composition of the inverse of a function and a function defines the relation (the “interface”) between \( \Lambda(\Sigma_{\text{trees}}) \) and \( \Lambda(\Sigma_{\text{logic}}) \) as \( \varphi_{\text{sem.}} \circ \varphi_{\text{derived trees}}^{-1} \). In general, such a composition is not a function, allowing for relating a derived tree (even more a string) with several logical formulas, and vice versa. This follows the observation of Culicover and Jackendoff (2005) that “[t]he combinatorial principles of syntax and semantics are independent; there is no ‘rule-to-rule’
homomorphism. (...) [T]he mapping between syntactic and semantic combinatoriality is many-to-many”. However, we implement the many-to-many relation with homomorphisms and inverses of homomorphisms.

In this section, we illustrate the power of this architecture that makes derivation structures a full grammatical object with three phenomena: idioms, subordinating conjunctions with reduced clauses, and scope ambiguity. For idioms, we use the fact that derivation structures are first-class citizens of the formalism. While this could also be expressed in TAG (for instance following the interpretation of derivation trees provided by Shieber 1994), this naturally fits our architecture. For subordinating conjunctions, we rely on the fact that the typing of abstract terms does not need to stick to the tree structure, and in particular to the Gorn addresses, unlike derivation trees in TAG, extending the modelling capabilities. Finally, for scope ambiguity, we show how our approach can take into account analyses from other formalisms, such as categorial and type-logical grammars. We do this remaining in the ACG model, contrary to the TAG extension (TAG with cosubstitution, Barker 2010) to which it corresponds. Other examples that go beyond TAG capabilities are discussed in Section 9, in particular for discourse parsing.

8.1 Idioms

Because TAGs provide whole fragments of phrase structures, they can encode the rigid parts of idioms as well as the ones that are subject to possible modifications. Moreover, the role of the derivation structure as a bridge to semantic interpretation nicely captures the relation between a composed syntax and an atomic meaning. With the ACG encoding of TAG, Kobele (2012) shows that we can introduce a constant term that is interpreted as the combination (by adjunction or substitution) of several elementary trees. It goes beyond the previous approaches (Abeillé and Schabes 1989; Shieber and Schabes 1990; Abeillé 1995) in that the derived tree does not need to be an elementary tree of the grammar, but is instead the result of a partial derivation.

We illustrate this with (32):

(32) John kicked the bucket.
A syntax-semantics interface for TAG through ACG

Figure 29 presents the initial trees that allow us to analyze (32) as the literal expression, with a compositional meaning built out of the composition of the initial trees $\alpha_{\text{John}}$, $\alpha_{\text{kicked}}$, $\beta_{\text{the}}$, and $\alpha_{\text{bucket}}$. We actually syntactically analyze the idiomatic expression the same way, except that the combination of $\alpha_{\text{kicked}}$, $\beta_{\text{the}}$, and $\alpha_{\text{bucket}}$ is also considered as the interpretation of the constant $c_{\text{kicked the bucket}}$. We use $c_{\text{kicked the bucket}}$ in the idiomatic derivation in Figure 30(c) to stress that there is no corresponding elementary tree. The ACG abstract term we get really corresponds to this derivation, as the term $C_{30(c)}$ of Equation (33) shows. In both cases, the derived tree is the same (Figure 30(a)). However, the derivation trees differ, as Figure 30(b) and Figure 30(c) show.

$$\alpha_{\text{kicked}} = S / \backslash$$
$$NP \quad VP$$
$$NP \quad VP$$
$$V \quad NP$$
$$/ \quad \backslash$$
$$\text{kicked}$$

$$\alpha_{\text{bucket}} = N / \backslash$$
$$\text{bucket}$$
$$\text{Det} \quad N^*$$
$$/ \quad \backslash$$
$$\text{the}$$

$$\beta_{\text{the}} = \text{NP}$$

Figure 29:
Elementary trees to analyze
kick the bucket

(a) Derived tree
(b) Literal derivation
(c) Idiomatic derivation

Table 18 (p. 582) shows how the interpretation of the constant abstract term for the idiom is interpreted, syntactically (by $\mathcal{G}_{\text{derived trees}}$) but not semantically (by $\mathcal{G}_{\text{sem.}}$), as the interpretation of the partial derivation $\lambda^o s \ a \ subj \ c_{\text{kicked}} \ s \ a \ subj \ (c_{\text{bucket}} \ c_{\text{the}})$. Then, according to the lexicons of Tables 17, 18, and 19, (33), (34), (35), and (36) hold. They show that the two terms $C_{30(b)}$ and $C_{30(c)}$ have the same interpretations as derived tree by $\mathcal{G}_{\text{derived trees}}$. But they have two different interpretations as logical formulas by the ACG $\mathcal{G}_{\text{sem.}}$.

$$C_{30(b)} = C_{\text{kicked}} \ I_S \ I_{VP} \ C_{\text{John}} \ (C_{\text{bucket}} \ C_{\text{the}})$$

$$C_{30(c)} = C_{\text{kicked the bucket}} \ I_S \ I_{VP} \ C_{\text{John}}$$

Figure 30:
Derived tree and derivation trees for John kicked the bucket

(33)
Table 17: TAG elementary tree encoding as \( \Lambda(\Sigma_{trees}) \) terms

<table>
<thead>
<tr>
<th>( \gamma )</th>
<th>Terms of ( \Lambda(\Sigma_{trees}) )</th>
<th>Corresponding TAG elementary tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gamma_{kicked} )</td>
<td>( \Delta : (T \to T) \to (T \to T) \to T \to T )</td>
<td>( \alpha_{kicked} )</td>
</tr>
<tr>
<td>( \gamma_{bucket} )</td>
<td>( \Delta : (T \to T) \to T )</td>
<td>( \alpha_{bucket} )</td>
</tr>
<tr>
<td>( \gamma_{the} )</td>
<td>( \Delta : (T \to T) \to T )</td>
<td>( \beta_{the} )</td>
</tr>
<tr>
<td>( \gamma_{kicked the bucket} )</td>
<td>( \Delta : (T \to T) \to (T \to T) \to T \to T )</td>
<td>None: composed from ( \alpha_{kicked}, \beta_{the}, ) and ( \alpha_{bucket} )</td>
</tr>
</tbody>
</table>

Table 18: Constants of \( \Sigma_{derivations} \) and their interpretation by \( \mathcal{G}_{\text{derived trees}} \) and \( \mathcal{G}_{\text{sem.}} \)

<table>
<thead>
<tr>
<th>( c )</th>
<th>( \Delta : (s \to s) \to (vp \to vp) \to np \to np \to s )</th>
<th>( \Delta : \text{derived trees } \gamma_{kicked} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_{the} )</td>
<td>( \Delta : N \to NP )</td>
<td>( \Delta : \text{derived trees } \gamma_{the} )</td>
</tr>
<tr>
<td>( c_{bucket} )</td>
<td>( \Delta : (n \to np) \to np )</td>
<td>( \Delta : \text{derived trees } \gamma_{bucket} )</td>
</tr>
<tr>
<td>( c_{kicked the bucket} )</td>
<td>( \Delta : (s \to s) \to (vp \to vp) \to np \to s )</td>
<td>( \Delta : \text{derived trees } \gamma_{kicked the bucket} )</td>
</tr>
</tbody>
</table>

Table 19: Constants of \( \Sigma_{\text{TAG}} \) and their interpretation by \( \mathcal{G}_{\text{TAG}} \)

<table>
<thead>
<tr>
<th>( C )</th>
<th>( \Delta : S_{A} \to VP_{A} \to NP \to NP \to S )</th>
<th>( \Delta : \text{TAG } C_{kicked} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_{the} )</td>
<td>( \Delta : N_{A} )</td>
<td>( \Delta : \text{TAG } C_{the} )</td>
</tr>
<tr>
<td>( C_{bucket} )</td>
<td>( \Delta : N_{A} \to NP )</td>
<td>( \Delta : \text{TAG } C_{bucket} )</td>
</tr>
<tr>
<td>( C_{kicked the bucket} )</td>
<td>( \Delta : S_{A} \to VP_{A} \to NP \to S )</td>
<td>( \Delta : \text{TAG } C_{kicked the bucket} )</td>
</tr>
</tbody>
</table>
A syntax-semantics interface for TAG through ACG

\[
G_{\text{TAG}}(C_{30(b)}) = c_{\text{kicked}} I_\ell I_{\text{VP}} c_{\text{John}} (c_{\text{bucket}} c_{\text{the}})
\]

\[
G_{\text{TAG}}(C_{30(c)}) = c_{\text{kicked the bucket}} I_\ell I_{\text{VP}} c_{\text{John}}
\]

\[
G_{\text{derived trees}} \circ G_{\text{TAG}}(C_{30(b)}) = S_2 (\text{NP}_1 \text{John}) (\text{VP}_2 (V_1 \text{kicked}))
\]

\[
= G_{\text{derived trees}} \circ G_{\text{TAG}}(C_{30(c)})
\]

\[
G_{\text{sem.}} \circ G_{\text{TAG}}(C_{30(b)}) = \exists! x. (\text{bucket} x) \land (\text{kick john} x)
\]

\[
G_{\text{sem.}} \circ G_{\text{TAG}}(C_{30(c)}) = \text{die john}
\]

8.2 Subordinating conjunctions

We saw in Section 7.2 that infinitive clauses behave like clauses missing a subject. In this case, the matrix clause (control verb) adjoins on the infinitive clause. As the latter is an argument of the modifier, we could use an extra \( S' \) type that was interpreted as \( (\text{NP} \rightarrow S) \rightarrow S \) and make the modifier fill the semantic subject with its own subject.

In the case of subordinating conjunctions, as in (37), it is the subordinate clause that adjoins on the matrix clause and uses it as argument, as Figure 31 shows: the substitution node \( S \) is meant for the reduced infinitive clause, and the foot node for adjoining into the matrix clause. But if the latter is interpreted as a full proposition, there is no way to decompose it so that its subject also fills the semantic subject position of the subordinate clause.

(37) In order to arrive on time, a man left early

\[\begin{align*}
\alpha_{\text{to arrive}} = & \quad S \\
& \quad \text{NP} /
\\
& \quad \text{VP} /
\\
& \quad \text{V} \\
& \quad \text{I} \\
& \quad \text{to} \\
& \quad \text{arrive}
\\
\beta_{\text{in order}} = & \quad S \\
& \quad \text{S} /
\\
& \quad \text{S}^* /
\\
& \quad \text{Conj} \\
& \quad \text{I} \\
& \quad \text{in order}
\\
\alpha_{\text{left}} = & \quad S \\
& \quad \text{NP} /
\\
& \quad \text{VP} /
\\
& \quad \text{V} \\
& \quad \text{I} \\
& \quad \text{left}
\\
\beta_{\text{on time}} = & \quad \text{VP} \\
& \quad \text{Adv} \\
& \quad \text{I} \\
& \quad \text{on time}
\\
\beta_{\text{early}} = & \quad \text{VP} \\
& \quad \text{Adv} \\
& \quad \text{I} \\
& \quad \text{early}
\end{align*}\]

Figure 31: Auxiliary trees for subordinate conjuncts
Sylvain Pogodalla

The solution we propose uses the flexible link between the derivation and the derived trees. The constraints ACGs set on this link have to do with the type, not with the term (provided the typing is preserved). In particular:

- there is no need for an adjunction on a $S_n$ node of a term in $\Lambda(\Sigma_{trees})$ to be the image of a term (in $\Lambda(\Sigma_{derivations})$) of type $S$. We already used this feature;
- there is no need for an actual node in the derived tree to allow for an adjunction.

In order to implement the solution, terms for verbs such as $c_{left}$ in Table 20 have an additional argument of type $((NP \rightarrow S) \rightarrow (NP \rightarrow S))$ corresponding to the type of the auxiliary trees of subordinate clauses. The latter results for instance from the substitution of an infinitive clause of type $(NP \rightarrow S)$ into the term standing for the initial tree of a subordinating conjunction such as $c_{in\,order}$. We can consider this additional argument as an additional possibility to get an adjunction on the $S$ root node (the same node where a $S \rightarrow S$ adjunction is possible). As usual, in case no actual adjunction of a subordinate clause occurs, we use the $I_{NP\rightarrow S}$ constant which is interpreted (syntactically and semantically) as the identity function.

<table>
<thead>
<tr>
<th>Table 20: Constants of $\Sigma_{derivations}$ and their interpretation by $\mathcal{G}<em>{derived,trees}$ and $\mathcal{G}</em>{sem.}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{left}$</td>
</tr>
<tr>
<td>$: (S \rightarrow S) \rightarrow ((NP \rightarrow S) \rightarrow (NP \rightarrow S)) \rightarrow (VP \rightarrow VP) \rightarrow NP \rightarrow S$</td>
</tr>
<tr>
<td>$: derived,trees \quad \gamma_{left}$</td>
</tr>
<tr>
<td>$: sem. \lambda^0_s \cdot sub,a,subj,sub,(\lambda^0,subj^p,p,(\lambda^0,leave,x))),subj$</td>
</tr>
<tr>
<td>$c_{to,arrive}$</td>
</tr>
<tr>
<td>$: (VP \rightarrow VP) \rightarrow NP \rightarrow S$</td>
</tr>
<tr>
<td>$: derived,trees \quad \gamma_{to,arrive}$</td>
</tr>
<tr>
<td>$: sem. \lambda^0_p,adv,subj,(adv,p,(\lambda^0,arrive,x))$</td>
</tr>
<tr>
<td>$c_{in,order}$</td>
</tr>
<tr>
<td>$: (NP \rightarrow S) \rightarrow ((NP \rightarrow S) \rightarrow (NP \rightarrow S))$</td>
</tr>
<tr>
<td>$: derived,trees \quad \gamma_{in,order}$</td>
</tr>
<tr>
<td>$: sem. \lambda^0_p,Q,subj,(\lambda^0,goal,(P,(\lambda^0,p,x))),(Q,(\lambda^0,p,x)))$</td>
</tr>
<tr>
<td>$I_{NP\rightarrow S}$</td>
</tr>
<tr>
<td>$: (NP \rightarrow S) \rightarrow (NP \rightarrow S)$</td>
</tr>
<tr>
<td>$: derived,trees \quad \lambda^0,x\cdot x$</td>
</tr>
<tr>
<td>$: sem. \lambda^0,x\cdot x$</td>
</tr>
<tr>
<td>$c_{early}$</td>
</tr>
<tr>
<td>$: \rightarrow VP \rightarrow VP$</td>
</tr>
<tr>
<td>$: derived,trees \quad \lambda^0,x\cdot VP,p,x,(Adv_{1,early})$</td>
</tr>
<tr>
<td>$: sem. \lambda^0,p,\lambda,x\cdot early(p,x)$</td>
</tr>
<tr>
<td>$c_{on,time}$</td>
</tr>
<tr>
<td>$: \rightarrow VP \rightarrow VP$</td>
</tr>
<tr>
<td>$: derived,trees \quad \lambda^0,x\cdot VP,p,x,(Adv_{1,on,time})$</td>
</tr>
<tr>
<td>$: sem. \lambda^0,p,\lambda,x\cdot on,time(p,x)$</td>
</tr>
</tbody>
</table>
A syntax-semantics interface for TAG through ACG

We can observe in $\mathcal{G}_{\text{derived trees}}(c_{\text{left}}) = \gamma_{\text{left}}$ how the subordinate clause is inserted. The latter corresponds to the sub argument in $\gamma_{\text{left}}$ in Table 22 (p. 586). It takes as an argument the whole S rooted subtree over which the NP subject is abstracted (with $\lambda^0s'$) and the actual subject subj of the matrix clause. So it is the subordinate clause that is responsible for first applying the matrix clause to its subject before plugging in the resulting tree at the foot node. We can observe this behavior in $\beta_{\text{in order}}$: the sub argument corresponds to the infinitive subordinate clause to be substituted in $\beta_{\text{in order}}$, while the matrix argument corresponds to the matrix clause into which it adjoins and to which the subj argument is given, as the subterm (matrix subj) shows.

As before, the higher-order types at the level of $\Sigma_{\text{derivations}}$ are interpretations of atomic types of $\Sigma_{\text{TAG}}$. In particular, we introduce the atomic type $S''_A := \text{TAG} (\text{NP} \rightarrow S) \rightarrow \text{NP} \rightarrow S$ (resp. $S''_{\text{WS}} := \text{TAG} \text{NP} \rightarrow S$) for the reduced subordinate clauses (resp. for the infinitive clause that occurs in subordinate clauses) as Table 21 shows.

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S''_A$</td>
<td>$:= \text{TAG} (\text{NP} \rightarrow S) \rightarrow \text{NP} \rightarrow S$</td>
</tr>
<tr>
<td>$S_{\text{WS}}$</td>
<td>$:= \text{TAG} \text{NP} \rightarrow S$</td>
</tr>
<tr>
<td>$c_{\text{left}}$</td>
<td>$s_A \rightarrow s''_A \rightarrow \text{VP}_A \rightarrow \text{NP} \rightarrow S$</td>
</tr>
<tr>
<td>$c_{\text{to arrive}}$</td>
<td>$\text{VP}<em>A \rightarrow S</em>{\text{WS}}$</td>
</tr>
<tr>
<td>$c_{\text{in order}}$</td>
<td>$S_{\text{WS}} \rightarrow s''_A$</td>
</tr>
<tr>
<td>$I_{\text{S''}}$</td>
<td>$s''_A \rightarrow \text{INP} \rightarrow S$</td>
</tr>
<tr>
<td>$c_{\text{early}}$</td>
<td>$\text{VP}_A$</td>
</tr>
<tr>
<td>$c_{\text{on time}}$</td>
<td>$\text{VP}_A$</td>
</tr>
</tbody>
</table>

Table 21: Constants of $\Sigma_{\text{TAG}}$ and their interpretation by $\mathcal{G}_{\text{TAG}}$

With the lexicon of Tables 20, 21, and 22, we can build terms that correspond to the derivation and derived trees of Figure 32 as (38), (39), and (40) show.\footnote{Note that all terms corresponding to initial trees where the adjunction of a subordinate clause can occur should have the extra argument added. For the sake of simplicity, only $c_{\text{left}}$ and $c_{\text{left}}$ are modified here.} We compute the semantic interpretation as in (41).
## Table 22: TAG elementary tree encoding as $\Lambda(\Sigma_{trees})$ terms

<table>
<thead>
<tr>
<th>Terms of $\Lambda(\Sigma_{trees})$</th>
<th>Corresponding TAG elementary tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma_{left} = \lambda^{o}S\ sub\ a\ subj\ obj.$</td>
<td>$\alpha_{left}$</td>
</tr>
<tr>
<td>$S\ (sub\ (\lambda^{o}s^{'}s_{2}\ s^{'}\ ((v_{1}\ left))))\ subj$</td>
<td></td>
</tr>
<tr>
<td>$(T \to T)\to ((T \to T) \to (T \to T)) \to T \to T \to T$</td>
<td></td>
</tr>
<tr>
<td>$\gamma_{to\ arrive} = \lambda^{o}a\ s.s_{2}\ (NP_{1}\ s)\ (a\ (VP_{2}\ (v_{1}\ to)\ (VP_{1}\ arrive)))$</td>
<td>$\alpha_{to\ arrive}$</td>
</tr>
<tr>
<td>$(T \to T) \to T \to T$</td>
<td></td>
</tr>
<tr>
<td>$\gamma_{in\ order} = \lambda^{o}subj\ matrix\ subj.$</td>
<td>$\beta_{in\ order}$</td>
</tr>
<tr>
<td>$s_{2}\ (S_{2}\ (conj_{1}\ in\ order)\ (sub\ (PRO_{1}\ e)))\ (matrix\ subj)$</td>
<td></td>
</tr>
<tr>
<td>$(T \to T) \to (T \to T) \to T \to T$</td>
<td></td>
</tr>
</tbody>
</table>

(a) Derived tree

(b) Derivation tree

Figure 32: Derived tree and derivation tree for *In order to arrive on time, a man left early*

(38) $C_{32} = C_{left} IS (C_{in\ order} (C_{to\ arrive} C_{on\ time})) C_{early} (C_{man} C_{a})$

(39) $g_{TAG}(C_{32}) = c_{left} IS (c_{in\ order} (c_{to\ arrive} c_{on\ time})) c_{early} (c_{man} c_{a})$

(40) $g_{derived\ trees} \circ g_{TAG}(C_{32}) =$

$S_{2}\ (S_{2}\ (conj_{1}\ in\ order)\ (S_{2}\ (NP_{1}\ (PRO_{1}\ e))) (VP_{2})$

$(VP_{2}\ (v_{1}\ to)\ (VP_{1}\ arrive)) (Adv_{1}\ on\ time)))))$

$(S_{2}\ (NP_{2}\ (Det_{1}\ a)\ (N_{1}\ man))$

$(VP_{2}\ (VP_{1}\ (v_{1}\ left))\ (Adv_{1}\ early)))$
A syntax-semantics interface for TAG through ACG

\[ \mathcal{G}_{\text{sem.}} \circ \mathcal{G}_{\text{TAG}}(C_{32}) = \exists x.(\text{man } x) \land (\text{goal(on_time(arrive } x))(\text{early(leave } x))) \]

Because \( \mathcal{G}_{\text{TAG}} \) is still second-order, parsing is available. Parsing the logical term \( t_{32}^{\text{logic}} \) (see (42)) results in the term \( t_{32} : S \) of \( \mathcal{A}(\mathcal{G}_{\text{sem.}} \circ \mathcal{G}_{\text{TAG}}) \). This is the same term of \( \mathcal{A}(\mathcal{G}_{\text{yield}} \circ \mathcal{G}_{\text{derived trees}} \circ \mathcal{G}_{\text{TAG}}) \) that we get when parsing \( t_{32}^{\text{string}} \).

\[ t_{32}^{\text{logic}} = \exists x.(\text{man } x) \land (\text{goal(on_time(arrive } x))(\text{early(leave } x))) \]

\[ t_{32}^{\text{string}} = \text{in } + \text{order } + \text{to } + \text{arrive } + \text{on } + \text{time } + \text{a } + \text{man } + \text{left } + \text{early} \]

\[ t_{32} = C_{\text{left}} I_{5} (C_{\text{in order}} (C_{\text{to arrive } C_{\text{on time}}}) C_{\text{early}} (C_{\text{man } C_{a}}) \]

8.3 Scope ambiguity and non-functional form-meaning relation

The phenomena we have modelled so far make far use of derivation structures (either in \( \Lambda(\Sigma_{\text{derivations}}) \) or in \( \Lambda(\Sigma_{\text{TAG}}) \)) that are very close (homomorphic) to TAG derivation trees. As we can see in Figure 15 (p. 557), the relation between TAG derivations as terms of \( \Lambda(\Sigma_{\text{TAG}}) \) and terms of \( \Lambda(\Sigma_{\text{logic}}) \) is functional (encoded by the composition \( \mathcal{G}_{\text{sem.}} \circ \mathcal{G}_{\text{TAG}} \)). The non-functional relation is between terms of \( \Lambda(\Sigma_{\text{trees}}) \) and terms of \( \Lambda(\Sigma_{\text{logic}}) \) (encoded by the relation \( \mathcal{G}_{\text{sem.}} \circ \mathcal{G}^{-1}_{\text{derived trees}} \)). So because there are two derivation trees for \textit{John kicked the bucket}, there are two possible semantic interpretations. But with only one derivation tree for \textit{every man loves some woman}, there is only one possible semantic interpretation. A possible solution to this problem is to use an underspecified representation formalism instead of higher-order logic to represent the semantics, as Pogodalla (2004a) proposes.

We present here another solution. It uses the power of higher-order typing of the abstract terms in order to provide TAGs with a relation between TAG derivation trees and meanings that is \textit{not functional}. Nevertheless, our grammatical architecture only appeals to homomorphisms. We introduce an abstract vocabulary \( \Sigma_{\text{CoTAG}} \) and two ACGs. The first one, \( \mathcal{G}_{\text{CoTAG}} \), maps terms of \( \Lambda(\Sigma_{\text{CoTAG}}) \) to terms of \( \Lambda(\Sigma_{\text{TAG}}) \), i.e., TAG derivation trees. The second one, \( \mathcal{G}_{\text{co-sem.}} \), maps terms of \( \Lambda(\Sigma_{\text{CoTAG}}) \) to terms of \( \Lambda(\Sigma_{\text{logic}}) \). It then provides a relation between \( \Lambda(\Sigma_{\text{TAG}}) \) and \( \Lambda(\Sigma_{\text{logic}}) \) as \( \mathcal{G}_{\text{co-sem.}} \circ \mathcal{G}^{-1}_{\text{CoTAG}} \) as Figure 33 (p. 588) shows. The derivation tree (in \( \Lambda(\Sigma_{\text{TAG}}) \)) of \textit{every man loves some woman}, for
instance, will have two antecedents in $\Lambda(\Sigma_{\text{CoTAG}})$, hence two semantic interpretations.

The idea is to use the type-raising methods of categorial and type-logical grammars. So, corresponding to a term $C_{\text{everyone}} : \text{NP}$ in $\Sigma_{\text{TAG}}$, we have a term $L_{\text{everyone}} : (\text{NP} \rightarrow \text{S}) \rightarrow \text{S}$ in $\Sigma_{\text{CoTAG}}$ such that $\mathfrak{g}_{\text{CoTAG}}(L_{\text{everyone}}) = \lambda^0 \text{P.P} C_{\text{everyone}}$. More generally, whenever a term of type $A$ occurring within a constituent of type $B$ can take scope over this term, we associate to $C_{\text{scoping}} : A_1 \rightarrow \ldots \rightarrow A_n \rightarrow A$ in $\Sigma_{\text{TAG}}$ a term $L_{\text{scoping}} : A_1 \rightarrow \ldots \rightarrow A_n \rightarrow (A \rightarrow B) \rightarrow B$ in $\Sigma_{\text{CoTAG}}$ such that $\mathfrak{g}_{\text{CoTAG}}(L_{\text{scoping}}) = \lambda^0 x_1 \ldots x_n \lambda^0 \text{P.P} (C_{\text{scoping}} x_1 \ldots x_n)$. For other lexical items $C_{\text{lex.item}} : \alpha$, we have $L_{\text{lex.item}} : \alpha$ such that $\mathfrak{g}_{\text{CoTAG}}(L_{\text{lex.item}}) = C_{\text{lex.item}}$. And for any atomic type $A$, $\mathfrak{g}_{\text{CoTAG}}(A) = A$. Table 23 exemplifies the approach for quantified noun phrases (note that proper
A syntax-semantics interface for TAG through ACG

nouns, for instance, are not type-raised). Atomic types in $\Sigma_{CoTAG}$ are the same as in $\Sigma_{TAG}$. With $L_{21}^{sws}$ and $L_{21}^{ows}$ as defined in (45) and (46) respectively, we indeed have $\mathcal{G}_{CoTAG}(L_{21}^{sws}) = \mathcal{G}_{CoTAG}(L_{21}^{ows})$, i.e., two different abstract terms of $\Lambda(\Sigma_{CoTAG})$ that are mapped to the same term of $\Lambda(\Sigma_{TAG})$ (TAG derivation tree).

\begin{equation}
L_{21}^{sws} = (L_{man} L_{every})
(\lambda^o x. (L_{woman} L_{some}) (\lambda^o y. L_{loves} I_S I_{VP} x y))
\end{equation}

\begin{equation}
\mathcal{G}_{CoTAG}(L_{21}^{sws}) = (\lambda^o P.P (C_{man} C_{every})) (\lambda^o x. (\lambda^o P.P (C_{woman} C_{some})))
(\lambda^o y. C_{loves} I_S I_{VP} x y))
= C_{loves} I_S I_{VP} (C_{man} C_{every}) (C_{woman} C_{some})
\end{equation}

(46)

\begin{equation}
L_{21}^{ows} = (L_{woman} L_{some})
(\lambda^o y. (L_{man} L_{every}) (\lambda^o x. L_{loves} I_S I_{VP} x y))
\end{equation}

\begin{equation}
\mathcal{G}_{CoTAG}(L_{21}^{ows}) = (\lambda^o P.P (C_{woman} C_{some})) (\lambda^o y. (\lambda^o P.P (C_{man} C_{every})))
(\lambda^o x. C_{loves} I_S I_{VP} x y))
= C_{loves} I_S I_{VP} (C_{man} C_{every}) (C_{woman} C_{some})
\end{equation}

In order to get two semantic interpretations from the two abstract terms of $\Lambda(\Sigma_{CoTAG})$, we need to directly provide them with a semantic lexicon. For if we keep on interpreting them through $\Sigma_{TAG}$ and $\Sigma_{derivations}$, because the two terms $L_{21}^{sws}$ and $L_{21}^{ows}$ are interpreted as a single term in $\Lambda(\Sigma_{TAG})$, we would still get a single interpretation. In other words, we do not want the diagram of Figure 33 to commute.

Table 24 defines the $\mathcal{G}_{co-sem}$ interpretation into terms of $\Lambda(\Sigma_{logic})$. Contrary to $\mathcal{G}_{sem}$, where NPs are interpreted with the higher-order type $(e \rightarrow t) \rightarrow t$, because quantified noun phrases are given the type $(NP \rightarrow S) \rightarrow S$ in $\Sigma_{CoTAG}$, we now interpret NP as $e$. All the other interpretations, in particular for verbs, are defined accordingly.\footnote{Note, however, that, because at the abstract level we only have linear types, in order to allow for non linearity at the object level, we have to uniformly interpret $\rightarrow$ as $\Rightarrow$, so that the image of $(NP \rightarrow S) \rightarrow S$ is $(e \rightarrow t) \rightarrow t$. Another possibility would be to use the exponential connectives of linear logic.}

We can now compute the semantic interpretation of $L_{21}^{sws}$ and $L_{21}^{ows}$ by $\mathcal{G}_{co-sem}$. Equations (47) and (48) show that these two terms are
Table 24: Constants of $\Sigma_{CoTAG}$ and their interpretation by $\mathcal{G}_{co-sem}$.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{\text{CoTAG}}$</td>
<td>$:=\text{co-sem.}$</td>
</tr>
<tr>
<td>$\text{NP} , := \text{co-sem.} , e$</td>
<td></td>
</tr>
<tr>
<td>$\text{VP} , := \text{co-sem.} , e \rightarrow t$</td>
<td></td>
</tr>
<tr>
<td>$L_{\text{John}}$</td>
<td>$:=\text{co-sem.} , \lambda x , \text{(human x)} \land (Q x)$</td>
</tr>
<tr>
<td>$L_{\text{loves}}$</td>
<td>$:=\text{co-sem.} , \lambda x , \text{(love x x') \land (love x x')}$</td>
</tr>
<tr>
<td>$L_{\text{someone}}$</td>
<td>$:=\text{co-sem.} , \lambda x , \text{(human x)} \land (Q x)$</td>
</tr>
<tr>
<td>$L_{\text{every}}$</td>
<td>$:=\text{co-sem.} , \lambda x , \text{(man x) \land (man x')} \land (Q x)$</td>
</tr>
<tr>
<td>$L_{\text{man}}$</td>
<td>$:=\text{co-sem.} , \lambda x , \text{det.det man}$</td>
</tr>
<tr>
<td>$L_{\text{woman}}$</td>
<td>$:=\text{co-sem.} , \lambda x , \text{det.det woman}$</td>
</tr>
</tbody>
</table>

mapped onto two logical formulas corresponding to the subject wide scope reading on the one hand, and to the object wide scope reading on the other hand.

\begin{align}
G_{co-sem}(L_{sws}^{21}) &= \forall x. (\text{man x}) \Rightarrow (\exists x'. (\text{woman x'}) \land (\text{love x x'})) \\
G_{co-sem}(L_{ows}^{21}) &= \exists x. (\text{woman x}) \land (\forall x'. (\text{man x'}) \Rightarrow (\text{love x' x}))
\end{align}

This approach to scope ambiguity, first proposed by Pogodalla (2007b,a), is used by Kobele and Michaelis (2012) to provide an ACG formalization of the cosubstitution operation for TAG (Barker 2010). This also makes explicit Barker’s (2010) claim that “cosubstitution is a version of the continuation-based approaches to scope-taking […]”. And, indeed, the type ($\text{NP} \rightarrow \text{S} \rightarrow \text{S}$) corresponds to making the continuation of a noun phrase (i.e., its scope) part of its interpretation.

$G_{CoTAG}$ is not a second-order ACG. In this particular case, because of lexicalization, we know that parsing is decidable, but it can be complex. Salvati (2007) presents a lexicalized third-order ACG whose membership problem reduces to an NP-complete problem. There is currently no implementation of parsing for such grammars in the ACG toolkit. The identification of fragments that are both linguistically relevant and computationally tractable is ongoing work.

This extension with more abstract levels can also be used to model (non-local) MCTAG. And one level more can control MCTAG (similar to the control that $G_{TAG}$ adds on $\mathcal{G}(G_{\text{derived trees}})$) so that it stays within the polynomially parsable languages of set-local MCTAG (Pogodalla 2009).
Moving to ACGs to encode TAGs and to build TAG semantic representations offers several advantages. First, we saw in Section 3.2 that we can benefit from parsing algorithms and optimization techniques grounded in well-established fields such as type theory and Datalog.

A second advantage, also concerning the parsing algorithms, is to offer an inherently reversible framework (Dymetman 1994). Kanazawa’s Datalog reduction (2007; 2017) indeed makes no hypothesis on the object language: it can be a language of strings, of trees, or of any kind of (almost linear) $\lambda$-terms. In the latter case, it can represent the usual logical semantic formulas. While in NLP parsing usually refers to building a parse structure (or a semantic term) from a string representation and generation (or surface realization) refers to building a string from a semantic representation, they both rely on ACG parsing (i.e., recovering the abstract structure from an object term), and the algorithms are the same.

This constitutes an important difference between the ACG approach and the synchronous approaches to semantic construction. If both are based on (or can be reformulated using) a tree transduction, the latter does not offer a built-in transformation to $\beta$-reduced terms (which may definitely not be trees but rather graphs) at the semantic level. When parsing strings, synchronous grammars (Nesson and Shieber 2006; Nesson 2009) build semantic trees that correspond to $\lambda$-terms before the $\beta$-reduction. Processing such trees to produce the $\beta$-reduced form is straightforward. However, the inverse process, the one that is of interest for generation, is not. It actually corresponds to the morphism inversion found in ACG parsing. The ACG framework tells us that this inversion is possible, and, for the second-order case, it has actually been implemented.

Koller and Kuhlmann (2011, 2012) also propose parsing by morphism inversion using interpreted regular tree grammars, and their approach completely fits the synchronous approach. But, as for synchronous TAG, this formalism is not well-suited to dealing with semantics represented with logical formulas. To parse a term $t$ requires that the set of trees that are interpreted as $t$ is regular. For instance, if the string algebra comes with the 2-ary concatenation operation, this
set is the set of all the bracketings of the string to parse (Koller and Kuhlmann 2011) (the string algebra Koller and Kuhlmann 2012 propose for the TAG encoding is different, in order to keep the complexity bound for parsing low). Applying the same approach to logical representations based on $\lambda$-calculus would mean representing all terms that are $\beta$-equivalent to the term we want to parse by a regular tree grammar. It is not clear how this can be done.

Semantic representation with $\lambda$-terms, and the ACG type-theoretic settings more generally, 28 also provides tight links with formal logical semantics. The various grammatical formalisms ACGs can encode may be linked to various semantic theories. This concerns both semantic theories, such as event semantics (Davidson 2001) (for a type-theoretic account, see Blom et al. 2012) or dynamic semantics (Kamp and Reyle 1993; Groenendijk and Stokhof 1991) (for a type-theoretic semantics account, see de Groote 2006; Martin and Pollard 2014), 29 and phenomena at the syntax-semantics interface where approaches based on underspecification (Pogodalla 2004a,b) or based on type theory and higher-order logic (Pogodalla 2007b,a; Kobele and Michaelis 2012) can be expressed.

Because they only use unification, the unification-based approaches to TAG semantics (Gardent and Kallmeyer 2003; Kallmeyer and Romero 2004, 2008) do not easily extend to higher-order semantics: only conjunctions of propositions are allowed, and no application. A first consequence is that the actual representation language needs to be embedded into a reified logical language (typically a labelled underspecified representation language). For instance, the semantics of an adverb adjoining to a $VP$ node cannot be represented as a function from $(e \to t)$ to $(e \to t)$. It is represented as a proposition expressing that some property holds of a label which gets its value by unification with the label corresponding to the semantics of the $VP$ in the verb initial tree. When dealing with higher-order representations, as for dynamic semantics of discourse (de Groote 2006; Martin and Pollard

---

28 This includes other categorial grammars (van Benthem 1986; Carpenter 1997; Steedman 2001; Steedman and Balridge 2011).

29 Note however that the semantic calculi are somewhat extended with additional operators and then do not fulfill the requirements allowing for reversibility. This is a research program on its own.
it becomes awkward to assign values of arguments with unification and to compute the semantic representation by $\beta$-reduction.

Moreover, ACGs uniformly deal with the interpretation of derivation trees, either as strings, derived trees, or semantic representations. Consequently, the same parsing algorithms apply. This is not the case for the unification-based approaches, and the reversibility of the grammars is not ensured.

Another benefit of the ACG approach to the syntax-semantics interface over the synchronous TAG or over the unification-based approach is that, by construction, it is compositional, and the homomorphism requirement between the syntactic and the semantic categories holds. For instance, in synchronous TAG, in a pair of syntactic and semantic trees, it is possible to link a node $X$ (in the syntactic tree) with a node of type $\alpha$ (in the semantic tree), while having a pair of auxiliary trees whose syntactic tree has a foot and a root node labelled by $X$, but the nodes in the semantic trees are labelled by $\beta \neq \alpha$, yielding semantic trees that are not well-typed. A similar thing can happen in unification-based approaches if the semantic features to be unified are not the same. This is not possible in ACG and the ACG toolkit would raise a typing error, in the same way that statically typed programming languages ensure type-safeness.

Finally, the modularity of the ACG framework allows us to look at TAG and TAG variants as fragments of a larger class of grammars. For instance, Multi-Component TAG (MCTAG: Weir 1988) can also be described using a similar architecture (Pogodalla 2009). It is also possible to add operations in addition to substitution and adjunction that would otherwise be difficult, if not impossible, to express as TAG (or MCTAG) operations. Such operations can be used in order to link a TAG phrase grammar with a TAG discourse grammar without requiring an intermediate processing step (Danlos et al. 2015, 2016), contrary to D-LTAG (Webber and Joshi 1998; Forbes et al. 2003; Webber 2004; Forbes-Riley et al. 2006) or D-STAG (Danlos 2009, 2011). But, provided the encoding remains in the second-order ACG class, these grammars remain reversible and there is no need to design new parsing algorithms.
We have presented a model of the syntax-semantics interface for TAGs hinging on the ACG framework. We demonstrated, with the help of classical TAG syntax-semantics examples and new modellings, that this framework offers a lot of flexibility and expressiveness. In particular, we built on the modular properties of ACG that result from the two notions of composition between grammars it provides. These composition modes have been used for the syntax-semantics interface on the one hand, and for restricting the derivations to actual TAG derivations using a second-order ACG on the other hand. This allowed us to apply the ACG parsing results and to make the grammar reversible so that both parsing and syntactic realization are available.

Moreover, we showed that new modellings can be proposed that extend the standard TAG analyses without the requirement of designing new parsing algorithms. This was illustrated with phenomena such as idioms and subordinating conjunctions. We also showed what we can bring into TAG accounts from type-logical frameworks, such as the modelling of scope ambiguities.

This shows how relevant ACGs are as models of the syntax-semantics interface in general, and for TAG in particular. Relying on the work we have presented here, we can consider modelling other standard extensions of TAGs, such as MCTAG. We can also consider relating TAG to other type-theoretic modellings of semantic phenomena, e.g., discourse, knowledge and beliefs, time, etc. Finally, we believe this can give a new perspective on ways to model phenomena which are challenging to model otherwise in TAG, such as coordination.

REFERENCES


Anne Abeillé (1993), Les nouvelles syntaxes, Armand Colin.

Anne Abeillé (1995), The flexibility of French idioms: a representation with Lexicalised Tree Adjoining Grammar, in Martin Everaert, Erik-Jan van der


Sylvain Pogodalla

Pierre BOURREAU and Sylvain SALVATI (2011), A Datalog recognizer for almost affine λ-CFGs, in Makoto KANAZAWA, András KORNAI, Marcus KRACHT, and Hiroyuki SEKI, editors, The mathematics of language, volume 6878 of Lecture notes in computer science, pp. 21–38, Springer, 10.1007/978-3-642-23211-4_2.


Bob CARPENTER (1997), Type-logical semantics, The MIT Press.


Benoît CRABBÉ (2005), Grammatical development with XMG, in Blache et al. (2005), pp. 84–100, 10.1007/11422532_6.


Peter W. CULICOVER and Ray JACKENDOFF (2005), Simpler syntax, Oxford University Press.


Philippe de Groote and Sylvain Pogodalla (2004), On the expressive power of Abstract Categorial Grammars: representing context-free formalisms,


Robert Frank, editor (2002), Proceedings of the Sixth International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6), Università di Venezia, ACL anthology: W02-22.


Claire Gardent and Yannick Parmentier (2005), Large scale semantic construction for Tree Adjoining Grammars, in Blache et al. (2005), pp. 131–146, 10.1007/11422532_9.
A syntax-semantics interface for TAG through ACG


Sylvain Pogodalla


A syntax-semantics interface for TAG through ACG


Sylvain Pogodalla (2007b), Generalizing a proof-theoretic account of scope ambiguity, in 7th International Workshop on Computational Semantics (IWCS-7), Tilburg, Netherlands, HAL open archive: inria-00112898.

Sylvain Pogodalla (2009), Advances in Abstract Categorial Grammars: language theory and linguistic modeling. ESSLLI 2009 Lecture Notes, Part II, HAL open archive: hal-00749297.


Owen Rambow and Matthew Stone, editors (2004), Proceedings of the 7th International Workshop on Tree Adjoining Grammar and Related Formalisms (TAG + 7), Simon Fraser University, Vancouver, BC, Canada, ACL anthology: W04-3300.


A syntax-semantics interface for TAG through ACG


Sylvain Pogodalla


A syntax-semantics interface for TAG through ACG


Fei Xia, Chung-hye Han, Martha Palmer, and Aravind K. Joshi (2000), Comparing lexicalized treebank grammars extracted from Chinese, Korean, and English corpora, in *Second Chinese Language Processing Workshop*, pp. 52–59, Association for Computational Linguistics, Hong Kong, China, 10.3115/1117769.1117778, ACL anthology: W00-1208.


This work is licensed under the Creative Commons Attribution 3.0 Unported License. http://creativecommons.org/licenses/by/3.0/