Abstract Categorial Grammars
ESSLLI Notes
(2005, Edinburgh, Scotland)

Philippe de Groote
Sylvain Pogodalla

Contents

1 Towards Abstract Categorial Grammars
Ph. de Groote, 2001 **3**
1.1 Introduction . . . 3
1.2 Definition of a multiplicative kernel . . . 3
1.2.1 Types, signature, and λ-terms . . . 4
1.2.2 Vocabulary, lexicon, grammar, and language . . . 5
1.2.3 Example . . . 6
1.3 Three computational paradigms . . . 7
1.3.1 Applicative paradigm . . . 7
1.3.2 Deductive paradigm . . . 7
1.3.3 Transductive paradigm . . . 8
1.4 Relating ACGs to other grammatical formalisms . . . 8
1.4.1 Context-free grammars . . . 8
1.4.2 Regular grammars and rational transducers . . . 9
1.4.3 Tree adjoining grammars . . . 9
1.5 Beyond the multiplicative fragment . . . 10
1.5.1 Additives . . . 10
1.5.2 Exponentials . . . 10
1.5.3 Quantifiers . . . 11
1.6 Grammars as first-class citizen . . . 11
1.7 Conclusion . . . 11
Bibliography . . . 11

2 Tree-Adjoining Grammars as Abstract Categorial Grammars
Ph. de Groote, 2002 **13**
2.1 Introduction . . . 13
2.2 Abstract Categorial Grammars . . . 13
2.3 Representing Tree-Adjoining Grammars . . . 15
2.4 Example . . . 16
2.5 Extracting the string languages . . . 17
2.6 Expressing adjoining constraints . . . 17
Bibliography . . . 18

3 On the Complexity of Higher-Order Matching in the Linear λ-Calculus
S. Salvati and Ph. de Groote, 2003 **19**
3.1 Introduction . . . 19
3.2 Higher-order matching in the linear λ-calculus . . . 20
3.3 1-Neg-sat . . . 21
3.4 Reduction of 1-neg-sat . . . 22
3.5 Correction of the reduction . . . 23
3.6 Related results . . . 26
Bibliography . . . 27

4 On the expressive power of Abstract Categorial Grammars: Representing context-free formalisms Ph. de Groote and S. Pogodalla, 2004 29
4.1 Introduction . . . 29
4.2 Abstract Categorial Grammars . . . 30
4.3 Strings as linear λ-terms . . . 32
4.4 Three context-free formalisms . . . 32
4.4.1 Context-free string grammars . . . 32
4.4.2 Linear context-free tree grammars . . . 33
4.4.3 Linear context-free rewriting systems . . . 34
4.5 Specifying context-free derivations . . . 34
4.6 Composition as first-order susbtitution . . . 35
4.7 Composition as second-order susbtitution . . . 36
4.8 Composition as third-order substitution . . . 38
4.9 Conclusions . . . 40
Bibliography . . . 40

5 Computing Semantic Representation: Towards ACG Abstract Terms as Derivation Trees S. Pogodalla, 2004a 43
5.1 ACG principles . . . 44
5.2 TAGs as ACGs . . . 44
5.3 Semantic representation for TAGs as ACGs . . . 46
5.3.1 Quantification . . . 47
5.3.2 Adverbs . . . 48
5.3.3 Raising verbs . . . 49
5.3.4 Verbs with phrasal arguments . . . 49
5.3.5 Wh-questions . . . 50
5.3.6 Control verbs . . . 51
Bibliography . . . 52

6 Further Readings 55
Bibliography . . . 55

Bibliography 57
Chapter 1 . . . 57
Chapter 2 . . . 58
Chapter 3 . . . 58
Chapter 4 . . . 59
Chapter 5 . . . 60
Chapter 6 . . . 61

Chapter 1

Towards Abstract Categorial Grammars

Ph. de Groote, 2001

We introduce a new categorial formalism based on intuitionistic linear logic. This formalism, which derives from current type-logical grammars, is abstract in the sense that both syntax and semantics are handled by the same set of primitives. As a consequence, the formalism is reversible and provides different computational paradigms that may be freely composed together.

1.1 Introduction

Type-logical grammars offer a clear cut between syntax and semantics. On the one hand, lexical items are assigned syntactic categories that combine via a categorial logic akin to the Lambek calculus (Lambek 1958). On the other hand, we have so-called semantic recipes, which are expressed as typed λ-terms. The syntax-semantics interface takes advantage of the Curry-Howard correspondence, which allows semantic readings to be extracted from categorial deductions (van Benthem 1986). These readings rely upon a homomorphism between the syntactic categories and the semantic types.

The distinction between syntax and semantics is of course relevant from a linguistic point of view. This does not mean, however, that it must be wired into the computational model. On the contrary, a computational model based on a small set of primitives that combine via simple composition rules will be more flexible in practice and easier to implement.

In the type-logical approach, the syntactic contents of a lexical entry is outlined by the following patern:

<*atom*> : <*syntactic category*>

On the other hand, the semantic contents obeys the following scheme:

<*λ-term*> : <*semantic type*>

This asymmetry may be broken by:

1. allowing λ-terms on the syntactic side (atomic expressions being, after all, particular cases of λ-terms),
2. using the same type theory for expressing both the syntactic categories and the semantic types.

The first point is a powerfull generalization of the usual scheme. It allows λ-terms to be used at a syntactic level, which is an approach that has been advocated by (Oehrle 1994). The second point may be satisfied by dropping the non-commutative (and non-associative) aspects of categorial logics. This implies that, contrarily to the usual categorial approaches, word order constraints cannot be expressed at the logical level. As we will see this apparent loss in expressive power is compensated by the first point.

1.2 Definition of a multiplicative kernel

In this section, we define an elementary grammatical formalism based on the ideas presented in the introduction. This elementary formalism is founded on the multiplicative fragment of linear logic (Girard 1987). For this reason,

we call it a *multiplicative kernel*. Possible extensions based on other fragments of linear logic are discussed in Section 1.5.

1.2.1 Types, signature, and λ-terms

We first introduce the mathematical apparatus that is needed in order to define our notion of an abstract categorial grammar.

Let A be a set of atomic types. The set $\mathscr{T}(A)$ of *linear implicative types* built upon A is inductively defined as follows:

1. if $a \in A$, then $a \in \mathscr{T}(A)$;
2. if $\alpha, \beta \in \mathscr{T}(A)$, then $(\alpha \multimap \beta) \in \mathscr{T}(A)$.

We now introduce the notion of a *higher-order linear signature*. It consists of a triple $\Sigma = \langle A, C, \tau \rangle$, where:

1. A is a finite set of atomic types;
2. C is a finite set of constants;
3. $\tau : C \to \mathscr{T}(A)$ is a function that assigns to each constant in C a linear implicative type in $\mathscr{T}(A)$.

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

1. if $c \in C$, then $c \in \Lambda(\Sigma)$;
2. if $x \in X$, then $x \in \Lambda(\Sigma)$;
3. if $x \in X$, $t \in \Lambda(\Sigma)$, and x occurs free in t exactly once, then $(\lambda x. t) \in \Lambda(\Sigma)$;
4. if $t, u \in \Lambda(\Sigma)$, and the sets of free variables of t and u are disjoint, then $(t\, u) \in \Lambda(\Sigma)$.

$\Lambda(\Sigma)$ is provided with the usual notion of capture avoiding substitution, α-conversion, and β-reduction as defined in (Barendregt 1984).

Given a higher-order linear signature $\Sigma = \langle A, C, \tau \rangle$, each linear λ-term in $\Lambda(\Sigma)$ may be assigned a linear implicative type in $\mathscr{T}(A)$. This type assignment obeys an inference system whose judgements are sequents of the following form:

$$\Gamma \vdash_{\Sigma} t : \alpha$$

where:

1. Γ is a finite set of λ-variable typing declarations of the form '$x : \beta$' (with $x \in X$ and $\beta \in \mathscr{T}(A)$), such that any λ-variable is declared at most once;
2. $t \in \Lambda(\Sigma)$;
3. $\alpha \in \mathscr{T}(A)$.

The axioms and inference rules are the following:

$$\vdash_{\Sigma} c : \tau(c) \quad \text{(cons)}$$

$$x : \alpha \vdash_{\Sigma} x : \alpha \quad \text{(var)}$$

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

$$\frac{\Gamma \vdash_{\Sigma} t : (\alpha \multimap \beta) \quad \Delta \vdash_{\Sigma} u : \alpha}{\Gamma, \Delta \vdash_{\Sigma} (t\, u) : \beta} \quad \text{(app)}$$

1.2.2 Vocabulary, lexicon, grammar, and language

We now introduce the abstract notions of a vocabulary and a lexicon, on which the central notion of an abstract categorial grammar is based.

A *vocabulary* is simply defined to be a higher-order linear signature.

Given two vocabularies $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$, a *lexicon* $\mathscr{L}$ from Σ_1 to Σ_2 (in notation, $\mathscr{L} : \Sigma_1 \to \Sigma_2$) is defined to be a pair $\mathscr{L} = \langle F, G \rangle$ such that:

1. $F : A_1 \to \mathscr{T}(A_2)$ is a function that interprets the atomic types of Σ_1 as linear implicative types built upon A_2;

2. $G : C_1 \to \Lambda(\Sigma_2)$ is a function that interprets the constants of Σ_1 as linear λ-terms built upon Σ_2;

3. the interpretation functions are compatible with the typing relation, *i.e.*, for any $c \in C_1$, the following typing judgement is derivable:
$$\vdash_{\Sigma_2} G(c) : \hat{F}(\tau_1(c)),$$
where $\hat{F}$ is the unique homomorphic extension of F.

As stated in Clause 3 of the above definition, there exists a unique type homomorphism $\hat{F} : \mathscr{T}(A_1) \to \mathscr{T}(A_2)$ that extends F. Similarly, there exists a unique λ-term homomorphism $\hat{G} : \Lambda(\Sigma_1) \to \Lambda(\Sigma_2)$ that extends G. In the sequel, when '$\mathscr{L}$' will denote a lexicon, it will also denote the homorphisms $\hat{F}$ and $\hat{G}$ induced by this lexicon. In any case, the intended meaning will be clear from the context.

Condition 3, in the above definition of a lexicon, is necessary and sufficient to ensure that the homomorphisms induced by a lexicon commute with the typing relations. In other terms, for any lexicon $\mathscr{L} : \Sigma_1 \to \Sigma_2$ and any derivable judgement
$$x_0 : \alpha_0, \ldots, x_n : \alpha_n \vdash_{\Sigma_1} t : \alpha$$
the following judgement
$$x_0 : \mathscr{L}(\alpha_0), \ldots, x_n : \mathscr{L}(\alpha_n) \vdash_{\Sigma_2} \mathscr{L}(t) : \mathscr{L}(\alpha)$$
is derivable. This property, which is reminiscent of Montague's homomorphism requirement (Montague 1970b), may be seen as an abstract realization of the compositionality principle.

We are now in a position of giving the definition of an abstract categorial grammar.

An abstract categorial grammar (ACG) is a quadruple $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ where:

1. $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ are two higher-order linear signatures; Σ_1 is called the *abstract vocabulary* and Σ_2 is called the *object vocabulary*;

2. $\mathscr{L} : \Sigma_1 \to \Sigma_2$ is a lexicon from the abstract vocabulary to the object vocabulary;

3. $s \in \mathscr{T}(A_1)$ is a type of the abstract vocabulary; it is called the *distinguished type* of the grammar.

Any ACG generates two languages, an abstract language and an object language. The *abstract language* generated by $\mathscr{G}$ ($\mathcal{A}(\mathscr{G})$) is defined as follows:
$$\mathcal{A}(\mathscr{G}) = \{t \in \Lambda(\Sigma_1) \mid \ \vdash_{\Sigma_1} t : s \text{ is derivable}\}$$

In words, the abstract language generated by $\mathscr{G}$ is the set of closed linear λ-terms, built upon the abstract vocabulary Σ_1, whose type is the distinguished type s. On the other hand, the *object language* generated by $\mathscr{G}$ ($\mathcal{O}(\mathscr{G})$) is defined to be the image of the abstract language by the term homomorphism induced by the lexicon $\mathscr{L}$:
$$\mathcal{O}(\mathscr{G}) = \{t \in \Lambda(\Sigma_2) \mid \exists u \in \mathcal{A}(\mathscr{G}).\ t = \mathscr{L}(u)\}$$

It may be useful of thinking of the abstract language as a set of abstract grammatical structures, and of the object language as the set of concrete forms generated from these abstract structures. Section 1.4 provides examples of ACGs that illustrate this interpretation.

1.2.3 Example

In order to exemplify the concepts introduced so far, we demonstrate how to accomodate the PTQ fragment of Montague (1973). We concentrate on Montague's famous sentence:

John seeks a unicorn (1)

For the purpose of the example, we make the two following assumptions:

1. the formalism provides an atomic type '*string*' together with a binary associative operator '$+$' (that we write as an infix operator for the sake of readability);
2. we have the usual logical connectives and quantifiers at our disposal.

We will see in Section 1.4 and 1.5 that these two assumptions, in fact, are not needed.

In order to handle the syntactic part of the example, we define an ACG ($\mathscr{G}_{12}$). The first step consists in defining the two following vocabularies:

$$\begin{aligned}\Sigma_1 = \langle\ & \{n, np, s\}, \{J, S_{re}, S_{dicto}, A, U\},\\ & \{J \mapsto np,\ S_{re} \mapsto (np \multimap (np \multimap s)),\\ & \ S_{dicto} \mapsto (np \multimap (np \multimap s)),\\ & \ A \mapsto (n \multimap np),\ U \mapsto n\}\ \rangle\end{aligned}$$

$$\begin{aligned}\Sigma_2 = \langle\ & \{string\}, \{\mathbf{John}, \mathbf{seeks}, \mathbf{a}, \mathbf{unicorn}\},\\ & \{\mathbf{John} \mapsto string,\ \mathbf{seeks} \mapsto string,\\ & \ \mathbf{a} \mapsto string, \mathbf{unicorn} \mapsto string\}\ \rangle\end{aligned}$$

Then, we define a lexicon $\mathscr{L}_{12}$ from the abstract vocabulary Σ_1 to the object vocabulary Σ_2:

$$\begin{aligned}\mathscr{L}_{12} = \langle\ & \{n \mapsto string, np \mapsto string,\\ & \ s \mapsto string\},\\ & \{J \mapsto \mathbf{John},\\ & \ S_{re} \mapsto \lambda x.\, \lambda y.\, x + \mathbf{seeks} + y,\\ & \ S_{dicto} \mapsto \lambda x.\, \lambda y.\, x + \mathbf{seeks} + y,\\ & \ A \mapsto \lambda x.\, \mathbf{a} + x,\\ & \ U \mapsto \mathbf{unicorn}\}\ \rangle\end{aligned}$$

Finally we have $\mathscr{G}_{12} = \langle \Sigma_1, \Sigma_2, \mathscr{L}_{12}, s\rangle$.

The semantic part of the example is handled by another ACG ($\mathscr{G}_{13}$), which shares with $\mathscr{G}_{12}$ the same abstract language. The object language of this second ACG is defined as follows:

$$\begin{aligned}\Sigma_3 = \langle\ & \{e, t\},\\ & \{\textsc{john}, \textsc{try-to}, \textsc{find}, \textsc{unicorn}\},\\ & \{\textsc{john} \mapsto e,\\ & \ \textsc{try-to} \mapsto (e \multimap ((e \multimap t) \multimap t)),\\ & \ \textsc{find} \mapsto (e \multimap (e \multimap t)),\\ & \ \textsc{unicorn} \mapsto (e \multimap t)\}\ \rangle\end{aligned}$$

Then, a lexicon from Σ_1 to Σ_3 is defined:

$$\begin{aligned}\mathscr{L}_{13} = \langle\ & \{n \mapsto (e \multimap t), np \mapsto ((e \multimap t) \multimap t),\\ & \ s \mapsto t\},\\ & \{J \mapsto \lambda P.\, P\, \textsc{john},\\ & \ S_{re} \mapsto\\ & \quad \lambda P.\, \lambda Q.\, Q\, (\lambda x.\, P\\ & \qquad (\lambda y.\, \textsc{try-to}\, y\, (\lambda z.\, \textsc{find}\, z\, x))),\\ & \ S_{dicto} \mapsto\\ & \quad \lambda P.\, \lambda Q.\, P\\ & \qquad (\lambda x.\, \textsc{try-to}\, x\\ & \qquad\quad (\lambda y.\, Q\, (\lambda z.\, \textsc{find}\, y\, z))),\\ & \ A \mapsto \lambda P.\, \lambda Q.\, \exists x.\, P\, x \wedge Q\, x,\\ & \ U \mapsto \lambda x.\, \textsc{unicorn}\, x\}\ \rangle\end{aligned}$$

This allows the ACG $\mathscr{G}_{13}$ to be defined as $\langle \Sigma_1, \Sigma_3, \mathscr{L}_{13}, s \rangle$.

The abstract language shared by $\mathscr{G}_{12}$ and $\mathscr{G}_{13}$ contains the two following terms:

$$S_{re}\, J\, (A\, U) \quad (2) \qquad S_{dicto}\, J\, (A\, U) \quad (3)$$

The syntactic lexicon $\mathscr{L}_{12}$ applied to each of these terms yields the same image. It β-reduces to the following object term:

$$\mathbf{John} + \mathbf{seeks} + \mathbf{a} + \mathbf{unicorn}$$

On the other hand, the semantic lexicon $\mathscr{L}_{13}$ yields the *de re* reading when applied to (2):

$$\exists x.\, \text{UNICORN}\, x \wedge \text{TRY-TO JOHN}\, (\lambda z.\, \text{FIND}\, z\, x)$$

and it yields the *de dicto* reading when applied to (3):

$$\text{TRY-TO JOHN}\, (\lambda y.\, \exists x.\, \text{UNICORN}\, x \wedge \text{FIND}\, y\, x)$$

Our handling of the two possible readings of (1) differs from the type-logical account of Morrill (1994) and Carpenter (1997). The main difference is that our abstract vocabulary contains two constants corresponding to *seek*. Consequently, we have two distinct entries in the semantic lexicon, one for each possible reading. This is only a matter of choice. We could have adopt Morrill's solution (which is closer to Montague original analysis) by having only one abstract constant S together with the following type assignment:

$$S \mapsto (np \multimap (((np \multimap s) \multimap s) \multimap s))$$

Then the types of J and A, and the two lexicons should be changed accordingly. The semantic lexicon of this alternative solution would be simpler. The syntactic lexicon, however, would be more involved, with entries such as:

$$S \mapsto \lambda x.\, \lambda y.\, x + \mathbf{seeks} + y\, (\lambda z.\, z)$$
$$A \mapsto \lambda x.\, \lambda y.\, y\, (\mathbf{a} + x)$$

1.3 Three computational paradigms

Compositional semantics associates meanings to utterances by assigning meanings to atomic items, and by giving rules that allows to compute the meaning of a compound unit from the meanings of its parts. In the type logical approach, following the Montagovian tradition, meanings are expressed as typed λ-terms and combine via functional application.

Dalrymple, Lamping, Pereira and Saraswat (1995) offer an alternative to this applicative paradigm. They present a deductive approach in which linear logic is used as a glue language for assembling meanings. Their approach is more in the tradition of logic programming.

The grammatical framework introduced in the previous section realizes the compositionality principle in a abstract way. Indeed, it provides compositional means to associate the terms of a given language to the terms of some other language. Both the applicative and deductive paradigms are available.

1.3.1 Applicative paradigm

In our framework, the applicative paradigm consists simply in computing, according to the lexicon of a given grammar, the object image of an abstract term. From a computational point of view it amounts to performing substitution and β-reduction.

1.3.2 Deductive paradigm

The deductive paradigm, in our setting, answers the following problem: does a given term, built upon the object vocabulary of an ACG, belong to the object language of this ACG. It amounts to a kind of proof-search that has been described by Merenciano and Morrill (1997) and by Pogodalla (2000). This proof-search relies on linear higher-order matching, which is a decidable problem (de Groote 2000).

1.3.3 Transductive paradigm

The example developped in Section 1.2.3 suggests a third paradigm, which is obtained as the composition of the applicative paradigm with the deductive paradigm. We call it the transductive paradigm because it is reminiscent of the mathematical notion of transduction (see Section 1.4.2). This paradigm amounts to the transfer from one object language to another object language, using a common abstract language as a pivot.

1.4 Relating ACGs to other grammatical formalisms

In this section, we illustrate the expressive power of ACGs by showing how some other families of formal grammars may be subsumed. It must be stressed that we are not only interested in a weak form of correspondence, where only the generated languages are equivalent, but in a strong form of correspondence, where the grammatical structures are preserved.

First of all, we must explain how ACGs may manipulate strings of symbols. In other words, we must show how to encode strings as linear λ-terms. The solution is well known: it suffices to represent strings of symbols as compositions of functions. Consider an arbitrary atomic type $*$, and define the type '*string*' to be $(* \multimap *)$. Then, a string such as 'abbac' may be represented by the linear λ-term $\lambda x.\, a\,(b\,(b\,(a\,(c\,x))))$, where the atomic strings 'a', 'b', and 'c' are declared to be constants of type $(* \multimap *)$. In this setting, the empty word (ϵ) is represented by the identity function ($\lambda x.\, x$) and concatenation ($+$) is defined to be functional composition ($\lambda f.\, \lambda g.\, \lambda x.\, f\ (g\, x)$), which is indeed an associative operator that admits the identity function as a unit.

1.4.1 Context-free grammars

Let $G = \langle T, N, P, S\rangle$ be a context-free grammar, where T is the set of terminal symbols, N is the set of non-terminal symbol, P is the set of rules, and S is the start symbol. We write $\mathcal{L}(G)$ for the language generated by G. We show how to construct an ACG $\mathscr{G}_G = \langle \Sigma_1, \Sigma_2, \mathscr{L}, S\rangle$ corresponding to G.

The abstract vocabulary $\Sigma_1 = \langle A_1, C_1, \tau_1\rangle$ is defined as follows:

1. The set of atomic types A_1 is defined to be the set of non-terminal symbols N.
2. The set of constants C_1 is a set of symbols in 1-1-correspondence with the set of rules P.
3. Let $c \in C_1$ and let '$X \to \omega$' be the rule corresponding to c. τ_1 is defined to be the function that assigns the type $[\![\omega]\!]_X$ to c, where $[\![\cdot]\!]_X$ obeys the following inductive definition:
   (a) $[\![\epsilon]\!]_X = X$;
   (b) $[\![Y\omega]\!]_X = (Y \multimap [\![\omega]\!]_X)$, for $Y \in N$;
   (c) $[\![a\omega]\!]_X = [\![\omega]\!]_X$, for $a \in T$.

The definition of the object vocabulary $\Sigma_2 = \langle A_2, C_2, \tau_2\rangle$ is as follows:

1. A_2 is defined to be $\{*\}$.
2. The set of constants C_2 is defined to be the set of terminal symbols T.
3. τ_2 is defined to be the function that assigns the type '*string*' to each $c \in C_2$.

It remains to define the lexicon $\mathscr{L} = \langle F, G\rangle$:

1. F is defined to be the function that interprets each atomic type $a \in A_1$ as the type '*string*'.
2. Let $c \in C_1$ and let '$X \to \omega$' be the rule corresponding to c. G is defined to be the function that interprets c as $\lambda x_1 \ldots . \lambda x_n.\, |\omega|$, where $x_1 \ldots x_n$ is the sequence of λ-variables occurring in $|\omega|$, and $|\cdot|$ is inductively defined as follows:
   (a) $|\epsilon| = \lambda x.\, x$;
   (b) $|Y\omega| = y + |\omega|$, for $Y \in N$, and where y is a fresh λ-variable;
   (c) $|a\omega| = a + |\omega|$, for $a \in T$.

It is then easy to prove that $\mathscr{G}_G$ is such that:

1. the abstract language $\mathcal{A}(\mathscr{G}_G)$ is isomorphic to the set of parse-trees of G.
2. the language generated by G coincides with the object language of $\mathscr{G}_G$, *i.e.*, $\mathcal{O}(\mathscr{G}_G) = \mathcal{L}(G)$.

For instance consider the CFG whose production rules are the following:

$$\begin{array}{l} S \rightarrow \epsilon, \\ S \rightarrow aSb, \end{array}$$

which generates the language $a^n b^n$. The corresponding ACG has the following abstract language, object language, and lexicon:

$$\begin{array}{rl} \Sigma_1 = \langle & \{S\}, \{A, B\}, \\ & \{A \mapsto S, B \mapsto ((S \multimap S)\} \ \rangle \\ \Sigma_2 = \langle & \{*\}, \{a, b\}, \\ & \{a \mapsto \mathit{string}, b \mapsto \mathit{string}\} \ \rangle \\ \mathscr{L} = \langle & \{S \mapsto \mathit{string}\}, \\ & \{A \mapsto \lambda x.\, x, B \mapsto \lambda x.\, a + x + b\} \ \rangle \end{array}$$

1.4.2 Regular grammars and rational transducers

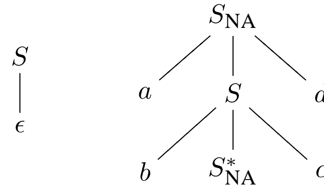
Regular grammars being particular cases of context-free grammars, they may be handled by the same construction. The resulting ACGs (which we will call “regular ACGs” for the purpose of the discussion) may be seen as finite state automata. The abstract language of a regular ACG correspond then to the set of accepting sequences of transitions of the corresponding automaton, and its object language to the accepted language.

More interestingly, rational transducers may also be accomodated. Indeed, two regular ACGs that shares the same abstract language correspond to a regular language homomorphism composed with a regular language inverse homomorphism. Now, after Nivat’s theorem (Nivat 1965), any rational transducer may be represented as such a bimorphism.

1.4.3 Tree adjoining grammars

The construction that allows to handle the tree adjoining grammars of Joshi (Joshi and Schabes 1997) may be seen as a generalization of the construction that we have described for the context-free grammars. Nevertheless, it is a little bit more involved. For instance, it is necessary to triplicate the non-terminal symbols in order to distinguish the initial trees from the auxiliary trees.

We do not have enough room in this paper for giving the details of the construction. We will rather give an example. Consider the TAG with the following initial tree and auxiliary tree:



It generates the non context-free language $a^n b^n c^n d^n$. This TAG may be represented by the ACG,

$$\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, S \rangle$$

where:

$$\begin{array}{rl} \Sigma_1 = \langle & \{S, S', S''\}, \{A, B, C\}, \\ & \{A \mapsto ((S'' \multimap S') \multimap S), \\ & \ B \mapsto (S'' \multimap ((S'' \multimap S') \multimap S')), \\ & \ C \mapsto (S'' \multimap S')\} \ \rangle \\ \Sigma_2 = \langle & \{*\}, \{a, b, c, d\}, \\ & \{a \mapsto \mathit{string}, b \mapsto \mathit{string}, \\ & \ c \mapsto \mathit{string}, d \mapsto \mathit{string}\} \ \rangle \end{array}$$

$$\begin{aligned}\mathscr{L} = \langle\ &\{S \mapsto \mathit{string}, S' \mapsto \mathit{string},\\ &\quad S'' \mapsto \mathit{string}\},\\ &\{A \mapsto \lambda f.\, f\,(\lambda x.\, x),\\ &\quad B \mapsto \lambda x.\, \lambda g.\, a + g\,(b + x + c) + d,\\ &\quad C \mapsto \lambda x.\, x\}\ \rangle\end{aligned}$$

One of the keystones in the above translation is to represent an adjunction node A as a functional parameter of type $A'' \multimap A'$. Abrusci, Fouqueré and Vauzeilles (1999) use a similar idea in their translation of the TAGs into non-commutative linear logic.

1.5 Beyond the multiplicative fragment

The linear λ-calculus on which we have based our definition of an ACG may be seen as a rudimentary functional programming language. The results in Section 1.4 indicate that, in theory, this rudimentary language is powerful enough. Nevertheless, in practice, it would be useful to increase the expressive power of the multiplicative kernel defined in Section 1.2 by providing features such as records, enumerated types, conditional expressions, etc.

From a methodological point of view, there is a systematic way of considering such extensions. It consists of enriching the type system of the formalism with new logical connectives. Indeed, each new logical connective may be interpreted, through the Curry-Howard isomorphism, as a new type constructor. Nonetheless, the possible additional connectives must satisfy the following requirements:

1. to be provided with introduction and elimination rules that satisfy Prawitz's inversion principle (Prawitz 1965). The resulting system must be strongly normalizable;

2. the resulting term language (or at least an interesting fragment of it) must have a decidable matching problem.

The first requirement ensures that the new types come with appropriate data constructors and discriminators, and that the associated evaluation rule terminates. This is mandatory for the applicative paradigm of Section 1.3. The second requirement ensures that the deductive paradigm (and consequently the transductive paradigm) may be fully automated.

The other connectives of linear logic are natural candidates for extending the formalism. In particular, they all satisfy the first requirement. On the other hand, the satisfaction of the second requirement is, in most of the cases, an open problem.

1.5.1 Additives

The additive connectives of linear logic '$\&$' and '$\oplus$' corresponds respectively to the cartesian product and the disjoint union. The cartesian product allows records to be defined. The disjoint union, together with the unit type '1', allows enumerated types and case analysis to be defined. Consequently, the additive connectives offer a good theoretical ground to provide ACG with feature structures.

1.5.2 Exponentials

The exponentials of linear logic are modal operators that may be used to go beyond linearity. In particular, the exponential '$!$' allows the intuitionistic implication '$\rightarrow$' to be defined, which corresponds to the possibility of dealing with non-linear λ-terms. A need for such non-linear λ-terms is already present in the example of Section 1.2.3. Indeed, the way of getting rid of the second assumption we made at the beginning of section 1.2.3 is to declare the logical symbols (*i.e.*, the existential quantifier and the conjunction that occurs in the interpretation of A in Lexicon $\mathscr{L}_{13}$) as constants of the object vocabulary Σ_3. Then, the interpretation of A would be something like:

$$\lambda P.\, \lambda Q.\, \textsc{exists}\,(\lambda x.\, \textsc{and}\,(P\,x)\,(Q\,x)).$$

Now, this expression must be typable, which is not possible in a purely linear framework. Indeed, the λ-term to which EXISTS is applied is not linear (there are two occurrences of the bound variable x). Consequently, EXISTS must be given $((e \rightarrow t) \multimap t)$ as a type.

1.5.3 Quantifiers

Quantifiers may also play a part. Uses of first-order quantification, in a type logical setting, are exemplified by Morrill (1994), Moortgat (1996), and Ranta (1994). As for second-order quantification, it allows for polymorphism.

1.6 Grammars as first-class citizen

The difference we make between an *abstract vocabulary* and an *object vocabulary* is purely conceptual. In fact, it only makes sense relatively to a given lexicon. Indeed, from a technical point of view, any vocabulary is simply a higher-order linear signature. Consequently, one may think of a lexicon $\mathscr{L}_{12} : \Sigma_1 \rightarrow \Sigma_2$ whose object language serves as abstract language of another lexicon $\mathscr{L}_{23} : \Sigma_2 \rightarrow \Sigma_3$. This allows lexicons to be sequentially composed. Moreover, one may easily construct a third lexicon $\mathscr{L}_{13} : \Sigma_1 \rightarrow \Sigma_3$ that corresponds to the sequential composition of $\mathscr{L}_{23}$ with $\mathscr{L}_{12}$. From a practical point of view, this means that the sequential composition of two lexicons may be compiled. From a theoretical point of view, it means that the ACGs form a category whose objects are vocabularies and whose arrows are lexicons. This opens the door to a theory where operations for constructing new grammars from other grammars could be defined.

1.7 Conclusion

This paper presents the first steps towards the design of a powerful grammatical framework based on a small set of computational primitives. The fact that these primitives are well known from programming theory renders the framework suitable for an implementation. A first prototype is currently under development.

Bibliography

Abrusci, M., Fouqueré, C. and Vauzeilles, J.: 1999, Tree-adjoining grammars in a fragment of the Lambek calculus, *Computational Linguistics* **25**(2), 209–236.

Barendregt, H. P.: 1984, *The lambda calculus, its syntax and semantics*, North-Holland. Revised edition.

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

Dalrymple, M., Lamping, J., Pereira, F. and Saraswat, V.: 1995, Linear logic for meaning assembly, *in* G. Morrill and R. Oehrle (eds), *Proceedings of Formal Gramma*, pp. 75–93.

de Groote, P.: 2000, Linear higher-order matching is np-complete, *in* L. Bachmair (ed.), *Rewriting Techniques and Applications, RTA'00*, Vol. 1833 of *LNCS*, Springer, pp. 127–140.

de Groote, P.: 2001, Towards abstract categorial grammars, *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155.

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

Joshi, A. K. and Schabes, Y.: 1997, *Tree-adjoining grammars*, Vol. 3 of *G. Rozenberg and A. Salomaa, editors, Handbook of formal languages*, Springer, chapter 2.

Lambek, J.: 1958, The mathematics of sentence structure, *American Mathematical Monthly* **65**(3), 154–170.

Merenciano, J. M. and Morrill, G. V.: 1997, Generation as deduction on labelled proof nets, *in* C. Retoré (ed.), *Proceedings of LACL-96*, Vol. 1328 of *LNAI*, Springer.

Montague, R.: 1970a, English as a formal language, *Linguaggi nella Società e nella Tecnica*, Edizioni di Comunità, Milan, pp. 189–224.

Montague, R.: 1970b, Universal grammar, *Theoria* **36**, 373–398.

Montague, R.: 1973, The proper treatment of quantification in ordinary english, *in* J. Hintikka (ed.), *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*, D. Reidel Publishing Co., Dordrecht, Holland, pp. 221–242. Reprinted in *Formal Philosophy*, by Richard Montague, Yale University Press, New Haven, CT, 1974, pp. 247–270.

Montague, R.: 1974, *Formal Philosophy: Selected Papers of Richard Montague*, Yale University Press, New Haven, CT. edited and with an introduction by Richmond Thomason.

Moortgat, M.: 1996, Categorial type logics, *in* J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, Elsevier Science Publishers, Amsterdam, pp. 93–177.

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

Nivat, M.: 1965, Transduction des langages de chomsky, *Annales de l'Institut Fourier* **18**, 339–455.

Oehrle, R. T.: 1994, Term-labeled categorial type systems, *Linguistic and Philosophy* **17**.

Pentus, M.: 1993, Lambek grammars are context free, *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Montreal, Canada, pp. 429–433.

Pogodalla, S.: 2000, Generation, lambek calculus, montague's semantics and semantic proof nets, *proceedings of the International Conference on Computational Linguistics*.
*http://www.xrce.xerox.com/Publications/Attachments/2000-006/pogodalla-coling2000.pdf

Prawitz, D.: 1965, *Natural Deduction, A Proof-Theoretical Study*, Almqvist & Wiksell, Stockholm.

Ranta, A.: 1994, *Type Theoretical Grammar*, Oxford University Press.

van Benthem, J.: 1986, *Essays in Logical Semantics*, Reidel, Dordrecht.

Chapter 2

Tree-Adjoining Grammars as Abstract Categorial Grammars

Ph. de Groote, 2002

2.1 Introduction

We recently introduced abstract categorial grammars (ACGs) (de Groote 2001) as a new categorial formalism based on Girard linear logic (Girard 1987). This formalism, which derives from current type-logical grammars (Carpenter 1997, Moortgat 1996, Morrill 1994, Oehrle 1994), offers some novel features:

- Any ACG generates two languages, an abstract language and an object language. The abstract language may be thought as a set of abstract grammatical structures, and of the object language as the set of concrete forms generated from these abstract structures. Consequently, one has a direct control on the parse structures of the grammar.
- The langages generated by the ACGs are sets of linear λ-terms. This may be seen as a generalization of both string-langages and tree-langages.
- ACGs are based on a small set of mathematical primitives that combine via simple composition rules. Consequently, the ACG framework is rather flexible.

Abstract categorial grammars are not intended as yet another grammatical formalism that would compete with other established formalisms. It should rather be seen as the kernel of a grammatical framework — in the spirit of (Ranta 2002) — in which other existing grammatical models may be encoded. This paper illustrates this fact by showing how tree-adjoining grammars (Joshi and Schabes 1997) may be embedded in abstract categorial grammars.

This embedding exemplifies several features of the ACG framework:

- The fact that the basic objects manipulated by an ACG are λ-terms allows higher-order operations to be defined. Typically, tree-adjunction is such a higher-order operation (Abrusci et al. 1999, Joshi and Kulick 1997, Mönnich 1997).
- The flexibility of the framework allows the embedding to be defined in two stages. A first ACG allows the tree langage of a given TAG to be generated. The abstract language of this first ACG corresponds to the derivation trees of the TAG. Then, a second ACG allows the corresponding string language to be extracted. The abstract language of this second ACG corresponds to the object language of the first one.

2.2 Abstract Categorial Grammars

This section defines our notion of an abstract categorial grammar. We first introduce the notions of *linear implicative types*, *higher-order linear signature*, *linear λ-terms* built upon a higher-order linear signature, and *lexicon*.

Let A be a set of atomic types. The set $\mathscr{T}(A)$ of *linear implicative types* built upon A is inductively defined as follows:

1. if $a \in A$, then $a \in \mathscr{T}(A)$;
2. if $\alpha, \beta \in \mathscr{T}(A)$, then $(\alpha \multimap \beta) \in \mathscr{T}(A)$.

A *higher-order linear signature* consists of a triple $\Sigma = \langle A, C, \tau \rangle$, where:

1. A is a finite set of atomic types;
2. C is a finite set of constants;
3. $\tau : C \to \mathscr{T}(A)$ is a function that assigns to each constant in C a linear implicative type in $\mathscr{T}(A)$.

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

1. if $c \in C$, then $c \in \Lambda(\Sigma)$;
2. if $x \in X$, then $x \in \Lambda(\Sigma)$;
3. if $x \in X$, $t \in \Lambda(\Sigma)$, and x occurs free in t exactly once, then $(\lambda x.\, t) \in \Lambda(\Sigma)$;
4. if $t, u \in \Lambda(\Sigma)$, and the sets of free variables of t and u are disjoint, then $(t\, u) \in \Lambda(\Sigma)$.

$\Lambda(\Sigma)$ is provided with the usual notion of capture avoiding substitution, α-conversion, and β-reduction as defined in (Barendregt 1984).

Given a higher-order linear signature $\Sigma = \langle A, C, \tau \rangle$, each linear λ-term in $\Lambda(\Sigma)$ may be assigned a linear implicative type in $\mathscr{T}(A)$. This type assignment obeys an inference system whose judgements are sequents of the following form:

$$\Gamma \vdash_\Sigma t : \alpha$$

where:

1. Γ is a finite set of λ-variable typing declarations of the form '$x : \beta$' (with $x \in X$ and $\beta \in \mathscr{T}(A)$), such that any λ-variable is declared at most once;
2. $t \in \Lambda(\Sigma)$;
3. $\alpha \in \mathscr{T}(A)$.

The axioms and inference rules are the following:

$$\vdash_\Sigma c : \tau(c) \quad \text{(cons)}$$

$$x : \alpha \vdash_\Sigma x : \alpha \quad \text{(var)}$$

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

$$\frac{\Gamma \vdash_\Sigma t : (\alpha \multimap \beta) \quad \Delta \vdash_\Sigma u : \alpha}{\Gamma, \Delta \vdash_\Sigma (t\, u) : \beta} \quad \text{(app)}$$

Given two higher-order linear signatures $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$, a lexicon $\mathscr{L} : \Sigma_1 \to \Sigma_2$ is a realization of Σ_1 into Σ_2, i.e., an interpretation of the atomic types of Σ_1 as types built upon A_2 together with an interpretation of the constants of Σ_1 as linear λ-terms built upon Σ_2. These two interpretations must be such that their homomorphic extensions commute with the typing relations. More formally, a *lexicon* $\mathscr{L}$ from Σ_1 to Σ_2 is defined to be a pair $\mathscr{L} = \langle F, G \rangle$ such that:

1. $F : A_1 \to \mathscr{T}(A_2)$ is a function that interprets the atomic types of Σ_1 as linear implicative types built upon A_2;

2. $G : C_1 \to \Lambda(\Sigma_2)$ is a function that interprets the constants of Σ_1 as linear λ-terms built upon Σ_2;

3. the interpretation functions are compatible with the typing relation, *i.e.*, for any $c \in C_1$, the following typing judgement is derivable:

$$\vdash_{\Sigma_2} G(c) : \hat{F}(\tau_1(c)),$$

   where $\hat{F}$ is the unique homomorphic extension of F.

We are now in a position of defining the notion of abstract categorial grammar. An *abstract categorial grammar* is a quadruple $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ where:

1. Σ_1 and Σ_2 are two higher-order linear signatures; they are called the *abstract vocabulary* and the *object vocabulary*, respectively ;

2. $\mathscr{L} : \Sigma_1 \to \Sigma_2$ is a lexicon from the abstract vocabulary to the object vocabulary;

3. s is an atomic type of the abstract vocabulary; it is called the *distinguished type* of the grammar.

The *abstract language* generated by $\mathscr{G}$ ($\mathcal{A}(\mathscr{G})$) is defined as follows:

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

In words, the abstract language generated by $\mathscr{G}$ is the set of closed linear λ-terms, built upon the abstract vocabulary Σ_1, whose type is the distinguished type s. On the other hand, the *object language* generated by $\mathscr{G}$ ($\mathcal{O}(\mathscr{G})$) is defined to be the image of the abstract language by the term homomorphism induced by the lexicon $\mathscr{L}$:

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

2.3 Representing Tree-Adjoining Grammars

In this section, we explain how to construct an abstract categorial grammar that generates the same tree langage as a given tree-adjoining grammar.

Let $G = \langle \Sigma, N, I, A, S \rangle$ be a tree-adjoining grammar, where Σ, N, I, A, and S are the set of terminal symbols, the set of non-terminal symbols, the set of initial trees, the set of auxiliary tree, and the distinguished non-terminal symbol, respectively. We associate to G an ACG $\mathscr{G}^G = \langle \Sigma_1^G, \Sigma_2^G, \mathscr{L}^G, s^G \rangle$ as follows.

The set of atomic types of Σ_1^G is made of two copies of the set of non-terminal symbols. Given $\alpha \in N$, we write α_S and α_A for the two corresponding atomic types. Then, we associate a constant

$$c_T : \gamma_{1A} \multimap \cdots \gamma_{mA} \multimap \beta_{1S} \multimap \cdots \beta_{nS} \multimap \alpha_S$$

to each initial tree T whose root node is labelled by α, whose substitution nodes are labeled by $\beta_1, \ldots, \beta_n$, and whose interior nodes are labeled by $\gamma_1, \ldots, \gamma_m$. Similarly, we associate a constant

$$c_{T'} : \gamma_{1A} \multimap \cdots \gamma_{mA} \multimap \beta_{1S} \multimap \cdots \beta_{nS} \multimap \alpha_A \multimap \alpha_A$$

to each auxiliary tree T' whose root node is labelled by α, whose substitution nodes are labeled by $\beta_1, \ldots, \beta_n$, and whose interior nodes are labeled by $\gamma_1, \ldots, \gamma_m$. Finally, we also associate to each non-terminal symnbol $\alpha \in N$, a constant I_α of type α_A. This concludes the specification of the abstract vocabulary.

The object vocabulary Σ_2^G allows labelled trees to be represented. Its set of atomic types contains only one element : τ (for *tree*). Then, its set of constants consists in:

1. constants of type τ corresponding to the terminal symbols of G;

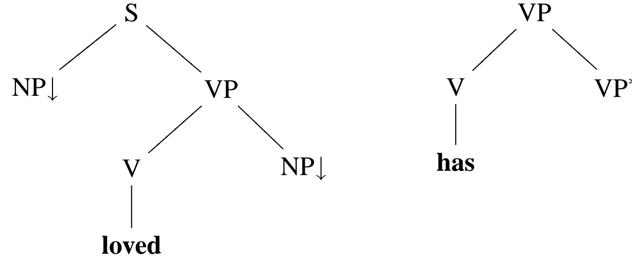
2. for each non-terminal symbol α, constants

$$\alpha_i : \underbrace{\tau \multimap \cdots \tau}_{i \text{ times}} \multimap \tau$$

   for $1 \leq i \leq k$, where k is the maximal branching of the interior nodes labelled with α that occur in the initial and auxiliary trees of G.

Clearly, the terms of type τ that can be built by means of the above set of constants correspond to trees whose frontier nodes are terminal symbols and whose interior nodes are labelled with non-terminal symbols.

It remains to define the lexicon $\mathscr{L}^G$. The rough idea is to represent the initial trees as trees (i.e., terms of type τ) and the auxiliary trees as functions over trees (i.e., terms of type $\tau \multimap \tau$). Consequently, for each $\alpha \in N$, we let $\mathscr{L}^G(\alpha_S) = \tau$ and $\mathscr{L}^G(\alpha_A) = \tau \multimap \tau$. Accordingly, the susbstitution nodes will be represented as first-order λ-variables of type τ , and the adjunction nodes as second-order λ-variables of type $\tau \multimap \tau$. The object representation of the elementary trees is then straightforward. Consider, for instance, the following initial tree and auxiliary tree:



According to our construction, the two abstract constants corresponding to these trees have the following types:

$$C_{\text{loved}} : \mathrm{S}_A \multimap \mathrm{VP}_A \multimap \mathrm{V}_A \multimap \mathrm{NP}_S \multimap \mathrm{NP}_S \multimap \mathrm{S}_S \quad \text{and} \quad C_{\text{has}} : \mathrm{VP}_A \multimap \mathrm{V}_A \multimap \mathrm{VP}_A \multimap \mathrm{VP}_A$$

Then, the realization of these two constants is as follows:

$$\begin{array}{rcl} \mathscr{L}^G(C_{\text{loved}}) & = & \lambda F.\, \lambda G.\, \lambda H.\, \lambda x.\, \lambda y.\, F\, (\mathrm{S}_2\, x\, (G\, (\mathrm{VP}_2\, (H\, (\mathrm{V}_1\, \textbf{loved}))\, y))) \\ \mathscr{L}^G(C_{\text{has}}) & = & \lambda F.\, \lambda G.\, \lambda H.\, \lambda x.\, F\, (\mathrm{VP}_2\, (G\, (\mathrm{V}_1\, \textbf{has}))\, (H\, x)) \end{array}$$

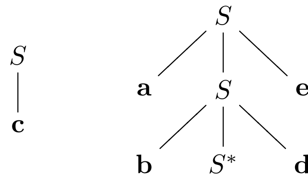
In order to derive actual trees, the second-order variables should eventually disappear. The abstract constants I_α have been introduced to this end. Consequently they are realized by the identity function, i.e., $\mathscr{L}^G(I_\alpha) = \lambda x.\, x$.

Finally, the distinguished type of $\mathscr{G}^G$ is defined to be S_S. This completes the definition of the ACG $\mathscr{G}^G$ associated to a TAG G. Then, the following proposition may be easily established.

Proposition 2.1. *Let G be a TAG. The tree-language generated by G is isomorphic to the object language of the ACG $\mathscr{G}^G$ associated to G.*

2.4 Example

Consider the TAG with the following initial tree and auxiliary tree:



It generates a non context-free language whose intersection with the regular language $a^*b^*c\,d^*e^*$ is $a^nb^nc\,d^ne^n$. According to the construction of Section 3, this TAG may be represented by the ACG, $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, S \rangle$, where:

$$\begin{array}{l} \Sigma_1 = \langle\ \{S_S, S_A\}, \{c_i, c_a, I\}, \\ \quad \{c_i \mapsto (S_A \multimap S_S), \\ \quad c_a \mapsto (S_A \multimap (S_A \multimap (S_A \multimap S_A))), \\ \quad I \mapsto S_A\}\ \rangle \end{array}$$

$$\begin{array}{l} \Sigma_2 = \langle\ \{\tau\}, \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, S_1, S_3\}, \\ \quad \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e} \mapsto \tau, \\ \quad S_1 \mapsto (\tau \multimap \tau), \\ \quad S_3 \mapsto (\tau \multimap (\tau \multimap (\tau \multimap \tau)))\}\ \rangle \end{array}$$

$$
\begin{aligned}
\mathscr{L} = \langle\ \{&S_S \mapsto \tau,\\
&S_A \mapsto (\tau \multimap \tau)\},\\
\{&c_i \mapsto \lambda f.\, f\,(S_1\,\mathbf{c}),\\
&c_a \mapsto \lambda f.\, \lambda g.\, \lambda h.\, \lambda x.\, f\,(S_3\,\mathbf{a}\,(g\,(S_3\,\mathbf{b}\,(h\,x)\,\mathbf{d}))\,\mathbf{e}),\\
&I \mapsto \lambda x.\, x\}\ \rangle
\end{aligned}
$$

2.5 Extracting the string languages

There is a canonical way of representing strings as linear λ-terms. It consists of encoding a string of symbols as a composition of functions. Consider an arbitrary atomic type σ, and define the type '*string*' to be $(\sigma \multimap \sigma)$. Then, a string such as '$abbac$' may be represented by the linear λ-term:

$$\lambda x.\, a\,(b\,(b\,(a\,(c\,x)))),$$

where the atomic strings 'a', 'b', and 'c' are declared to be constants of type $(\sigma \multimap \sigma)$. In this setting, the empty word is represented by the identity function:

$$\epsilon \triangleq \lambda x.\, x$$

and concatenation is defined to be functional composition:

$$\alpha + \beta \triangleq \lambda\alpha.\, \lambda\beta.\, \lambda x.\, \alpha\,(\beta\,x),$$

which is indeed an associative operator that admits the identity function as a unit.

This allows a second ACG, $\mathscr{G}'^{G}$, to be defined. Its abstract vocabulary is the object vocabulary Σ_2^G of $\mathscr{G}^G$. Its object vocabulary allows string of terminal symbols to be represented. Its lexicon interprets each constant of type τ as an atomic string, and each constant α_i as a concatenation operator. This second ACG, $\mathscr{G}'^{G}$, extracts the yields of the trees. Then, by composing $\mathscr{G}^G$ with $\mathscr{G}'^{G}$, one obtains an ACG which generates the same string-language as G.

Let us continue the example of Section 4. The second ACG, $\mathscr{G}' = \langle \Sigma_1', \Sigma_2', \mathscr{L}', S' \rangle$, is defined as follows:

$$\Sigma_1' = \Sigma_2$$

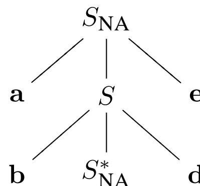
$$
\begin{aligned}
\Sigma_2' = \langle\ &\{\sigma\}, \{a, b, c, d, e\},\\
&\{a, b, c, d, e \mapsto (\sigma \multimap \sigma)\}\ \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{L}' = \langle\ \{&\tau \mapsto (\sigma \multimap \sigma)\},\\
\{&\mathbf{a} \mapsto \lambda x.\, a\,x,\\
&\mathbf{b} \mapsto \lambda x.\, b\,x,\\
&\mathbf{c} \mapsto \lambda x.\, c\,x,\\
&\mathbf{d} \mapsto \lambda x.\, d\,x,\\
&\mathbf{e} \mapsto \lambda x.\, e\,x,\\
&S_1 \mapsto \lambda f.\, \lambda x.\, f\,x,\\
&S_3 \mapsto \lambda f.\, \lambda g.\, \lambda h.\, f\,(g\,(h\,x))\}\ \rangle
\end{aligned}
$$

2.6 Expressing adjoining constraints

Adjunction, which is enabled by second-order variables at the object level, is explicitly controlled at the abstract level by means of types. This typing discipline may be easily refined in order to express adjoining constraints such as selective, null, or obligatory adjunction.

Consider again the TAG given in Section 4. By adding the following null adjunction constraints on its auxiliary tree:



one obtains a grammar that generates exactly the non context-free language $a^n b^n c\, d^n e^n$. These constraints may be expressed in a simple and natural way. It suffices to exclude the constrained nodes from the arguments of the λ-term corresponding to the auxiliary tree. This gives the following modified ACG:

$$\begin{aligned}\Sigma_1 = \langle\ & \{S_S, S_A\}, \{c_i, c_a, I\},\\ & \{c_i \mapsto (S_A \multimap S_S),\\ & \ \ c_a \mapsto (S_A \multimap S_A),\\ & \ \ I \mapsto S_A\}\ \rangle\end{aligned}$$

$$\begin{aligned}\Sigma_2 = \langle\ & \{\tau\}, \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, S_1, S_3\},\\ & \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e} \mapsto \tau,\\ & \ \ S_1 \mapsto (\tau \multimap \tau),\\ & \ \ S_3 \mapsto (\tau \multimap (\tau \multimap (\tau \multimap \tau)))\}\ \rangle\end{aligned}$$

$$\begin{aligned}\mathscr{L} = \langle\ & \{S_S \mapsto \tau,\\ & \ \ S_A \mapsto (\tau \multimap \tau)\},\\ & \{c_i \mapsto \lambda f.\, f\,(S_1\,\mathbf{c}),\\ & \ \ c_a \mapsto \lambda f.\, \lambda x.\, S_3\,\mathbf{a}\,(f\,(S_3\,\mathbf{b}\,x\,\mathbf{d}))\,\mathbf{e},\\ & \ \ I \mapsto \lambda x.\, x\}\ \rangle\end{aligned}$$

The other kinds of adjunction constraints may be expressed in a similar way.

Bibliography

Abrusci, M., Fouqueré, C. and Vauzeilles, J.: 1999, Tree-adjoining grammars in a fragment of the Lambek calculus, *Computational Linguistics* **25**(2), 209–236.

Barendregt, H. P.: 1984, *The lambda calculus, its syntax and semantics*, North-Holland. Revised edition.

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

de Groote, P.: 2001, Towards abstract categorial grammars, *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155.

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

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

Joshi, A. K. and Kulick, S.: 1997, Partial proof trees as building blocks for a categorial grammar, *Linguistics and Philosophy* **20**(6), 637–667.

Joshi, A. K. and Schabes, Y.: 1997, *Tree-adjoining grammars*, Vol. 3 of *G. Rozenberg and A. Salomaa, editors, Handbook of formal languages*, Springer, chapter 2.

Mönnich, U.: 1997, Adjunction as substitution, *in* G.-J. Kruijff, G. Morrill and R. Oehrle (eds), *Proceedings of Formal Grammar*, pp. 169–178.

Moortgat, M.: 1996, Categorial type logics, *in* J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, Elsevier Science Publishers, Amsterdam, pp. 93–177.

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

Oehrle, R. T.: 1994, Term-labeled categorial type systems, *Linguistic and Philosophy* **17**.

Ranta, A.: 2002, Grammatical Framework: A type-theoretical grammar formalism, *Journal of Functional Programming* . To appear. Manuscript available at `http://www.cs.chalmers.se/articles/gf-jfp.ps.gz`.

Chapter 3

On the Complexity of Higher-Order Matching in the Linear λ-Calculus

S. Salvati and Ph. de Groote, 2003

We prove that second-order matching in the linear λ-calculus with linear occurrences of the unknowns is NP-complete. This result shows that context matching and second-order matching in the linear λ-calculus are, in fact, two different problems.

3.1 Introduction

Higher-order unification, which consists in solving a syntactic equations between two simply-typed λ-term (modulo β, or modulo $\beta\eta$), is undecidable (Huet 1973), even in the second-order case (Goldfarb 1981). Consequently, several restrictions of the problem have been introduced and studied in the literature(see (Dowek 2001) for a survey).

Higher-order matching is such a restriction. It consists in solving equations whose right-hand sides do not contain any unknown. This problem, which is indeed simpler, has been shown to be decidable in the second-order case (Huet 1976), in the third-order case (Dowek 1994), and in the fourth order case (Padovani 1996). Starting form the sixth-order case, higher-order matching modulo β is undecidable (Loader 2003). On the other-hand the decidability of higher-order matching modulo $\beta\eta$ is still open.

Another restriction consists in studying unification in the linear λ-calculus where every λ-abstraction $\lambda x.\, M$ is such that M contains exactly one free occurrence of x. This problem, in the second-order case, is related to context unification (Comon 1998a, Comon 1998b). It has been studied by Levy under the name of *linear second-order unification* (Levy 1996). Nevertheless, its decidability is still open. On the other hand, the more restricted problem of higher-order matching in the linear λ-calculus has been shown to be decidable (de Groote 2000). In fact, it is NP-complete, which is also the case of context matching (Schmidt-Schauß and Stuber 2001). A related problem consists in deciding whether a matching problem between simply typed λ-terms admits a linear solution. This more general problem is also decidable (Dougherty and Wierzbicki 2002)

Finally, several other restrictions concern the way the unknowns occur in the equation. In particular, another notion of linearity appears in the literature. Indeed, linear unification (or matching) also designates equations whose unknowns occur only once.

In this paper, we will be concerned with these two different notions of linearity (equations between linear λ-terms, or linear occurrences of the unknowns). In order not to confuse them, we will speak of *matching in the linear λ-calculus*, in the first case, and we will use the expression *linear matching* for the second case. This question of vocabulary being settled, we may state our main result: *linear second-order matching in the linear λ-calculus is NP-complete.* Such a complexity, at first sight, might be surprising. Indeed, it seems to be folklore that context matching and second-order matching in the linear λ-calculus are equivalent problems. Our result, however, shows that this is not quite the case. Indeed, linear context matching is polynomial (Schmidt-Schauß and Stuber 2001).

The paper is organized as follows. Section 3.2 contains prerequisite basic notions, and defines precisely what is linear second-order matching in the linear λ-calculus. In Section 3.3, we define a variant of the satifiability

problem (which we call 1-neg-sat), and we prove its NP-completeness. Section 3.4 shows how to reduce 1-neg-sat to linear second-order matching in the linear λ-calculus, and Section 3.5 proves the correctness of the reduction. Finally, in Section 3.6, we state some related results.

3.2 Higher-order matching in the linear λ-calculus

This section reviews basic definitions and fixes the notations that we use in the sequel of the paper.

Definition 3.1. *Let $\mathscr{A}$ be a finite set, the elements of which are called* atomic types. *The set $\mathscr{F}$ of* linear functional types *built upon $\mathscr{A}$ is defined according to the following grammar:*

$$\mathscr{F} \quad ::= \quad \mathscr{A} \mid (\mathscr{F} \multimap \mathscr{F}).$$

■

We let the lowercase Greek letters $(\alpha, \beta, \gamma, \ldots)$ range over $\mathscr{F}$, and we adopt the usual convention that $\alpha_1 \multimap \alpha_2 \multimap \cdots \alpha_n \multimap \alpha$ stands for $\alpha_1 \multimap (\alpha_2 \multimap (\cdots (\alpha_n \multimap \alpha) \cdots))$, and we write $\alpha^n \multimap \beta$ for:

$$\underbrace{\alpha \multimap \ldots \multimap \alpha}_{n\times} \multimap \beta$$

The order of such a linear functional type is defined as usual:

$$\begin{aligned}&\text{order}(a) = 1 \text{ if } a \in \mathscr{A}\\&\text{order}(\alpha \multimap \beta) = \max(\text{order}(\alpha) + 1, \text{order}(\beta))\end{aligned}$$

Then, the notion of raw λ-term is defined as follows.

Definition 3.2. *Let $(\Sigma_\alpha)_{\alpha\in\mathscr{F}}$ be a family of pairwise disjoint finite sets indexed by $\mathscr{F}$, whose almost every member is empty. Let $(\mathscr{X}_\alpha)_{\alpha\in\mathscr{F}}$ and $(\mathscr{Y}_\alpha)_{\alpha\in\mathscr{F}}$ be two families of pairwise disjoint countably infinite sets indexed by $\mathscr{F}$, such that $(\bigcup_{\alpha\in\mathscr{F}} \mathscr{X}_\alpha) \cap (\bigcup_{\alpha\in\mathscr{F}} \mathscr{Y}_\alpha) = \varnothing$. The set $\mathscr{T}$ of raw λ-terms is defined according to the following grammar:*

$$\mathscr{T} \quad ::= \quad \Sigma \mid \mathscr{X} \mid \mathscr{Y} \mid \lambda\mathscr{X}.\mathscr{T} \mid (\mathscr{T}\,\mathscr{T}),$$

where $\Sigma = \bigcup_{\alpha\in\mathscr{F}} \Sigma_\alpha$, $\mathscr{X} = \bigcup_{\alpha\in\mathscr{F}} \mathscr{X}_\alpha$, and $\mathscr{Y} = \bigcup_{\alpha\in\mathscr{F}} \mathscr{Y}_\alpha$. ■

In this definition, Σ is the set of constants, $\mathscr{X}$ is the set of *λ-variables*, and $\mathscr{Y}$ is the set of *meta-variables* or *unknowns*. We let the lowercase roman letters $(\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots)$ range over the constants, the lowercase italic letters $(x, y, z, \ldots)$ range over the λ-variables, the uppercase bold letters $(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \ldots)$ range over the unknowns, and the uppercase italic letters $(M, N, O, \ldots)$ range over the λ-terms.

We write $h(M_1, \ldots, M_n)$ for a λ-term of the form $((\ldots(h\,M_1)\ldots)\,M_n)$, where h is either a constant, a λ-variable, or a meta-variable. Given such a term, h is called the head of the term.

The notions of free and bound occurrences of a λ-variable are defined as usual, and we write $\mathrm{FV}(M)$ for the set of λ-variables that occur free in a λ-term M. Finally, a λ-term that does not contain any meta-variable is called a *pure λ-term*.

We then define the notion of term of the linear λ-calculus..

Definition 3.3. *The family $(\mathscr{T}_\alpha)_{\alpha\in\mathscr{F}}$ of sets of terms of the linear λ-calculus is inductively defined as follows:*

1. *if* $\mathrm{a} \in \Sigma_\alpha$ *then* $\mathrm{a} \in \mathscr{T}_\alpha$*;*
2. *if* $\mathbf{X} \in \mathscr{Y}_\alpha$ *then* $\mathbf{X} \in \mathscr{T}_\alpha$*;*
3. *if* $x \in \mathscr{X}_\alpha$ *then* $x \in \mathscr{T}_\alpha$*;*
4. *if* $x \in \mathscr{X}_\alpha$*,* $M \in \mathscr{T}_\beta$*, and* $x \in \mathrm{FV}(M)$*, then* $\lambda x.\,M \in \mathscr{T}_{(\alpha\multimap\beta)}$*;*
5. *if* $M \in \mathscr{T}_{(\alpha\multimap\beta)}$*,* $N \in \mathscr{T}_\alpha$*, and* $\mathrm{FV}(M) \cap \mathrm{FV}(N) = \varnothing$*, then* $(M\,N) \in \mathscr{T}_\beta$*.* ■

Clauses 4 and 5 imply that any term $\lambda x.\,M$ of the linear λ-calculus is such that there is exactly one free occurrence of x in M. Remark, on the other hand, that constants and unknowns may occur several times in the same linear λ-term.

We define the set of terms of the linear λ-calculus to be $\bigcup_{\alpha\in\mathscr{F}}\mathscr{T}_\alpha$. One easily proves that the sets $(\mathscr{T}_\alpha)_{\alpha\in\mathscr{F}}$ are pairwise disjoint. Consequently, we may define the type term M to be the unique linear type α such that $M\in\mathscr{T}_\alpha$. This allows the order of a term to be defined as the order of its type. In particular, we will speak about the order of a meta-variable.

The notions of α-conversion, η and β-reduction are defined as usual. In particular, we write $\twoheadrightarrow_\beta$ for the relation of β-reduction, and $=_\beta$ for the relation of β-conversion.

We let $M[x{:=}N]$ denote the usual capture-avoiding substitution of a λ-variable by a λ-term. Similarly, $M[\mathbf{X}{:=}N]$ denotes the capture-avoiding substitution of a meta-variable by a λ-term. We abbreviate expressions like $M[x_1{:=}N_1]\cdots[x_n{:=}N_n]$ as $M[x_i{:=}N_i]_{i=1}^n$.

We now give a precise definition of the matching problem with which we are concerned.

Definition 3.4. *A* matching problem in the linear λ-calculus *is a pair of terms of the linear λ-calculus $\langle M,N\rangle$ of the same type such that N is pure (i.e., does not contain any meta-variable).*

Such a problem admits a solution if and only if there exists a substitution $(\mathbf{X}_i{:=}O_i)_{i=1}^n$ such that

$$M[\mathbf{X}_i{:=}O_i]_{i=1}^n =_\beta N$$

where $\{\mathbf{X}_1,\dots,\mathbf{X}_n\}$ is the set of meta-variables that occur in M. ■

In the above definition, we have taken the relation of β-conversion to be the notion of equality between λ-terms. Nevertheless, all the results we will establish remain valid when taking the relation of $\eta\beta$-conversion as the notion of equality.

In the sequel of this paper, a pair of λ-terms $\langle M,N\rangle$ obeying the conditions of the above definition will also be called a *syntactic equation*. The order of such an equation is defined to be the maximum of the orders of the meta-variables occurring in left-hand side the equation. Finally, such an equation is said to be linear if the meta-variables occurring in its left-hand side occur only once.

In this paper, we will mainly be concerned with linear second-order matching in the linear λ-calculus, i.e, the problem of solving a linear second-order syntactic equation between terms of the linear λ-calculus.

3.3 1-Neg-sat

In this section, we define a variant of the satisfiability problem, due to Kilpeläinen (Kilpeläinen and Mannila 1995).

We first remind the reader of some basic definitions. Given a finite set $\mathcal{A}=\{a_1,\dots,a_n\}$ of boolean variables, a literal is defined to be either a boolean variable a_i or its negation $\neg a_i$. A clause is a finite set of literals, and a satisfiability problem consists in a finite set of clauses. A positive literal a_i is satisfied by a valuation $\eta:\mathcal{A}\to\{0,1\}$ if and only if $\eta(a_i)=1$, and a negative literal $\neg a_i$ is satisfied if and only if $\eta(a_i)=0$, in which case we also write $\eta(\neg a_i)=1$. Then, a satisfiability problem $\mathcal{C}$ admits a solution if and only if there exists a valuation $\eta:\mathcal{A}\to\{0,1\}$ such that for all $C\in\mathcal{C}$ there exists $l\in C$ with $\eta(l)=1$. As well known, satisfiability is NP-complete (Cook 1971).

We now introduce the variant of the satisfiability problem that we call *1-neg-sat*.

Definition 3.5. *Let $\mathcal{A}$ be a finite set of boolean variables, and $\mathcal{C}$ be a finite set of clauses over $\mathcal{A}$. $\mathcal{C}$ is called a 1-neg-sat problem if and only if for all $a\in\mathcal{A}$, there exists exactly one $C\in\mathcal{C}$ such that $\neg a\in C$.* ■

The next result is due to Kilpeläinen (Kilpeläinen and Mannila 1995).

Lemma 3.1. *1-Neg-sat is NP-complete.*

Proof. We show that any satisfiability problem can be reduced to a 1-neg-sat-problem.

Let $\mathcal{C}$ be a finite set of clauses over the set of boolean variables $\mathcal{A}=\{a_1,\dots,a_n\}$. We introduce a set $\mathcal{B}=\{b_1,\dots,b_n\}$ of fresh boolean variables, and we define $\mathcal{D}$ to be the set of clauses $\bigcup_{i=1}^n\{\{a_i,b_i\},\{\neg a_i,\neg b_i\}\}$. Clearly, any valuation η that satisfies $\mathcal{D}$ is such that $\eta(a_i)=0$ if and only if $\eta(b_i)=1$. Conversely, any valuation η such that $\eta(a_i)=0$ if and only if $\eta(b_i)=1$ satisfies $\mathcal{D}$.

Then we define $\mathcal{C}^*$ as the set of clauses obtained from $\mathcal{C}$ by replacing each occurrence of $\neg a_i$ by b_i. By construction, $\mathcal{C}^*\cup\mathcal{D}$ is a 1-neg-sat problem. Moreover, any valuation that satisfies this problem satisfies $\mathcal{C}$. Conversely, given a valuation η that satisfies $\mathcal{C}$, the valuation η' such that $\eta'(a_i)=\eta(a_i)$ and $\eta'(b_i)=\neg\eta(a_i)$ satisfies $\mathcal{C}^*\cup\mathcal{D}$. □

3.4 Reduction of 1-neg-sat

In this section we show how to associate to any 1-neg-sat problem a linear second order syntactic equation in the linear λ-calculus.

Let $\mathcal{C} = \{C_1, \ldots, C_m\}$ be a 1-neg-sat problem defined over the set of boolean variables $\mathcal{A} = \{a_1, \ldots, a_n\}$. We define $\mathrm{neg} : \mathcal{A} \to \mathcal{C}$ to be the function such that:

$$\mathrm{neg}(a_i) = C_j \quad \text{if and only if} \quad \neg a_i \in C_j.$$

Similarly, we define $\mathrm{pos} : \mathcal{A} \to \mathscr{P}(\mathcal{C})$ such that:

$$\mathrm{pos}(a_i) = \{C_j \in \mathcal{C} \mid a_i \in C_j\}.$$

For each $i \in \{1, \ldots, n\}$, let m_i be the cardinality of $\mathrm{pos}(a_i)$, and define $\psi_i : \{1, \ldots, m_i\} \to \{1, \ldots, m\}$ to be a function such that:

$$\mathrm{pos}(a_i) = \{C_{\psi_i(1)}, \ldots, C_{\psi_i(m_i)}\}.$$

In case $m_i = 0$, by convention, ψ_i is defined to be the empty function.

Now, let o be an atomic type. In order to define the syntactic equation associated to $\mathcal{C}$, we introduce the following constants and meta-variables:

1. a constant a of type o;
2. a constant f of type $o^n \multimap o$;
3. for each clause $C \in \mathcal{C}$, a constant $\overline{C}$ of type $o^m \multimap o$;
4. a meta-variable $\mathbf{X}$ of type $o^m \multimap o$;
5. m^2 meta-variables $\mathbf{X}_{11}, \ldots, \mathbf{X}_{1m}, \ldots, \mathbf{X}_{m1}, \ldots, \mathbf{X}_{mm}$ of type o.

For $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, m\}$, we define the following terms:

$$R_{ij} = \begin{cases} \overline{C_{\psi_i(j)}}(a, \ldots, a) \text{ if } j \leq m_i \\ a \text{ otherwise.} \end{cases}$$

Then, for $i \in \{1, \ldots, n\}$, we define:

$$R_i = \overline{\mathrm{neg}(a_i)}(R_{i1}, \ldots, R_{im})$$

Finally, the syntactic equation $\langle L_{\mathcal{C}}, R_{\mathcal{C}} \rangle$, associated to $\mathcal{C}$, is defined as follows:

$$L_{\mathcal{C}} = \mathbf{X}(\overline{C_1}(\mathbf{X}_{11}, \ldots, \mathbf{X}_{1m}), \ldots, \overline{C_m}(\mathbf{X}_{m1}, \ldots, \mathbf{X}_{mm}))$$

$$R_{\mathcal{C}} = f(R_1, \ldots, R_n)$$

Let us illustrate the above reduction by an example. Consider the following clauses:

$$C_1 = \{a_1\}, c_2 = \{a_1, a_2\}, C_3 = \{\neg a_1, \neg a_2\}$$

to which we associate the constants c_1, c_2, and c_3 of type $o \multimap o \multimap o \multimap o$, respectively. We have $\mathrm{neg}(a_1) = C_3$, $\mathrm{neg}(a_2) = C_3$, $\mathrm{pos}(a_1) = \{C_1, C_2\}$, and $\mathrm{pos}(a_2) = \{C_2\}$. Consequently, the terms R_{ij} are the following:

$$\begin{array}{lll} R_{11} = c_1(a, a, a) & R_{12} = c_2(a, a, a) & R_{13} = a \\ R_{21} = c_2(a, a, a) & R_{22} = a & R_{23} = a \end{array}$$

Hence, the syntactic equation associated to this 1-neg-sat problem is as follows:

$$L_{\mathcal{C}} = \mathbf{X}(c_1(\mathbf{X}_{11}, \mathbf{X}_{12}, \mathbf{X}_{13}),\ c_2(\mathbf{X}_{21}, \mathbf{X}_{22}, \mathbf{X}_{23}),\ c_3(\mathbf{X}_{31}, \mathbf{X}_{32}, \mathbf{X}_{33}))$$

$$R_{\mathcal{C}} = f(c_3(c_1(a, a, a), c_2(a, a, a), a),\ c_3(c_2(a, a, a), a, a))$$

The intuition behind this reduction is the following. If the syntactic equation admits a solution, the term substituted for $\mathbf{X}$ must be of the form:

$$\lambda x_1 \ldots x_m . f(S_1, \ldots, S_n)$$

where each term S_i is either some λ-variable x_k, or some application of the form:

$$\overline{\mathrm{neg}(a_i)}(S_{i1}, \ldots, S_{im}).$$

The first case corresponds to a boolean variable a_i such that $\eta(a_i) = 0$, while the second case corresponds to a boolean variable a_i such that $\eta(a_i) = 1$.

Back to our example, one sees that the given equation admits the following solution:

$$\begin{cases} \mathbf{X} & := & \lambda x_1\, x_2\, x_3.\, f(c_3(x_1, x_2, a),\, x_3) \\ \mathbf{X}_{31} & := & c_2(a, a, a) \\ \mathbf{X}_{ij} & := & a \text{ for } i \neq 3 \text{ and } j \neq 1 \end{cases}$$

which corresponds, indeed, to the only valuation that satisfies $\mathcal{C}$, namely, the valuation η such that $\eta(a_1) = 1$ and $\eta(a_2) = 0$.

3.5 Correction of the reduction

Consider again a 1-neg-sat problem $\mathcal{C} = \{C_1, \ldots, C_m\}$ defined over the set of boolean variables $\mathcal{A} = \{a_1, \ldots, a_n\}$, and let the λ-terms R_{ij}, R_i, $L_{\mathcal{C}}$, and $R_{\mathcal{C}}$ be defined as in the previous section.

We first prove that the syntactic equation $\langle L_{\mathcal{C}}, R_{\mathcal{C}}\rangle$ admits a solution whenever $\mathcal{C}$ is satisfiable. To this end, suppose that $\mathcal{C}$ is satisfied by a valuation η. Consequently, there exists a choice function ϕ that picks in each clause a literal that this satisfied by η. More precisely, we defined $\phi : \{1, \ldots, m\} \to \{1, \ldots, n\}$ to be a function such that

$$\text{either } \eta(a_{\phi(i)}) = 1 \text{ and } a_{\phi(i)} \in C_i, \text{ or } \eta(a_{\phi(i)}) = 0 \text{ and } \neg a_{\phi(i)} \in C_i.$$

Remark that this function is such that if $\phi(i) = \phi(j)$ and $\eta(a_{\phi(i)}) = 0$ then $i = j$. This is due to the constraint that a negative literal occurs in only one clause.

Let $\{x_1, \ldots, x_m\}$ be a set of λ-variables. We define the family of terms S_i, for $i \in \{1, \ldots, n\}$, as follows:

$$S_i := \begin{cases} x_j \ \text{ if } \eta(a_i) = 0 \text{ and } j \text{ such that } \phi(j) = i \text{ exists} \\ \overline{\mathrm{neg}(a_i)}(S_{i1}, \ldots, S_{im}) \ \text{ otherwise} \end{cases}$$

where, in the second case, the family of terms S_{ij} is the following:

$$S_{ij} := \begin{cases} x_k \ \text{ if } \eta(a_i) = 1 \text{ and } k \text{ such that } \phi(k) = i \text{ and } R_{ij} = \overline{C_k}(a, \ldots, a) \text{ exists} \\ R_{ij} \ \text{ otherwise} \end{cases}$$

Finally, we define:

$$S = \lambda x_1 \ldots x_m . f(S_1, \ldots, S_n)$$

As we will show, the above term is the main ingredient of a solution to the syntactic equation $\langle L_{\mathcal{C}}, R_{\mathcal{C}}\rangle$. In order to establish this fact, we first prove that S is indeed a λ-term of the linear λ-calculus.

Lemma 3.2. *For all $i \in \{1, \ldots, m\}$, $x_i \in \mathrm{FV}(S_{\phi(i)})$.*

Proof. We proceed by case analysis, according to the value of $\eta(a_{\phi(i)})$.

Suppose that $\eta(a_{\phi(i)}) = 0$. Then, by definition, we have that $S_{\phi(i)} = x_i$. Hence, $x_i \in \mathrm{FV}(S_{\phi(i)})$.

On the other hand, when $\eta(a_{\phi(i)}) = 1$, we have, by definition of ϕ, that $C_i \in \mathrm{pos}(a_{\phi(i)})$. Consequently, there exists j such that $R_{\phi(i)j} = \overline{C_i}(a, \ldots, a)$. Therefore, by definition, $S_{\phi(i)j} = x_i$, which implies $x_i \in \mathrm{FV}(S_{\phi(i)})$. □

Lemma 3.3. *If $x_i \in \mathrm{FV}(S_j)$ then $\phi(i) = j$, for any $i \in \{1, \ldots, m\}$ and any $j \in \{1, \ldots, n\}$.*

Proof. An immediate consequence of the definition of the family of terms R_{ij}. □

Lemma 3.4. *For all $i \in \{1, \ldots, n\}$ and all $C_k \in \mathrm{pos}(a_i)$, there exists exactly one $j \in \{1, \ldots, m\}$ such that $R_{ij} = \overline{C_k}(a, \ldots, a)$.*

Proof. An immediate consequence of the definition of the family of terms R_{ij}. □

Lemma 3.5. *$S = \lambda x_1 \dots x_m.f(S_1, \dots, S_n)$ is a term of the linear λ-calculus.*

Proof. We have to prove that each of the λ-variables $x_1, \dots, x_m$ has exactly one occurrence in $f(S_1, \dots, S_n)$. By Lemma 3.2, we know that $x_1, \dots, x_m \in \mathrm{FV}(f(S_1, \dots, S_n))$. Hence, it remains to show that for any $i \in \{1, \dots, m\}$, x_i occurs at most once in $f(S_1, \dots, S_n)$. By Lemma 3.3, this amounts to prove that for any $j \in \{1, \dots, n\}$, x_i occurs at most once in S_j. So, suppose that $x_i \in \mathrm{FV}(S_j)$. Then, either $x_i = S_j$, or $x_i = S_{jk}$ for some k. In the second case, k is such that $R_{jk} = \overline{C_i}(a, \dots, a)$. Hence, it is unique by Lemma 3.4. Therefore, in both cases, there is only one occurence of x_i in S_j. □

It appears in the proof of Lemma 3.2 that for all $i \in \{1, \dots, m\}$ either there exists $k \in \{1, \dots, n\}$ such that $x_i = S_k$, or there exists $k \in \{1, \dots, n\}$ and $l \in \{1, \dots, m\}$ such that $x_i = S_{kl}$. This fact allows the family of terms T_{ij} (for $i, j \in \{1, \dots, m\}$) to be defined as follows:

$$T_{ij} = \begin{cases} R_{kj} \text{ if } k \text{ such that } x_i = S_k \text{ exists} \\ a \text{ if } k \text{ and } l \text{ such that } x_i = S_{kl} \text{ exist} \end{cases}$$

It is immediate that these terms are terms of the linear λ-calculus.

We are now in a position of establishing that the syntactic equation $\langle L_{\mathcal{C}}, R_{\mathcal{C}} \rangle$ admits a solution provided that $\mathcal{C}$ is satisfiable.

Proposition 3.1. *Let $\mathcal{C}$ be a 1-neg-sat problem, and $\langle L_{\mathcal{C}}, R_{\mathcal{C}} \rangle$ be the associated syntactic equation. If $\mathcal{C}$ is satisfiable, then $\langle L_{\mathcal{C}}, R_{\mathcal{C}} \rangle$ admits a solution.*

Proof. The fact that $\mathcal{C}$ is satisfiable allows the terms S, and T_{ij} to be defined, and we prove that

$$L_{\mathcal{C}}[\mathbf{X} := S][\mathbf{X}_{ij} := T_{ij}]_{i=1}^{m}{}_{j=1}^{m} \twoheadrightarrow_\beta R_{\mathcal{C}}$$

Indeed, we have:

$$\begin{aligned} & L_{\mathcal{C}}[\mathbf{X} := S][\mathbf{X}_{ij} := T_{ij}]_{i=1}^{m}{}_{j=1}^{m} \\ & \quad = S(\overline{C_1}(T_{11}, \dots, T_{1m}), \dots \overline{C_m}(T_{m1}, \dots, T_{mm})) \\ & \quad \twoheadrightarrow_\beta f(S_1, \dots, S_n)[x_j := \overline{C_j}(T_{j1}, \dots, T_{jm})]_{j=1}^{m} \end{aligned}$$

Then, it remains to show that for all $i \in \{1, \dots, n\}$:

$$S_i[x_j := \overline{C_j}(T_{j1}, \dots, T_{jm})]_{j=1}^{m} = R_i$$

There are two cases:

1. $S_i = x_k$, for some $k \in \{1, \dots, m\}$.

   In this case, we have that $T_{kl} = R_{il}$, for all $l \in \{1, \dots, m\}$. We also have $\eta(a_i) = 0$ and $\phi(k) = i$, which implies that $\mathrm{neg}(a_i) = C_k$. Consequently:

$$\begin{aligned} x_k[x_j := \overline{C_j}(T_{j1}, \dots, T_{jm})]_{j=1}^{m} &= x_k[x_k := \overline{C_k}(R_{i1}, \dots, R_{im})] \\ &= \overline{C_k}(R_{i1}, \dots, R_{im}) \\ &= \overline{\mathrm{neg}(a_i)}(R_{i1}, \dots, R_{im}) \\ &= R_i \end{aligned}$$

2. $S_i = \overline{\mathrm{neg}(a_i)}(S_{i1}, \dots, S_{im})$.

   In this case, it is sufficient to show that for all $k \in \{1, \dots, m\}$:

$$S_{ik}[x_j := \overline{C_j}(T_{j1}, \dots, T_{jm})]_{j=1}^{m} = R_{ik}$$

   There are two subcases. In the case $S_{ik} = x_l$, for some $l \in \{1, \dots, m\}$, we have that $R_{ik} = \overline{C_l}(a, \dots, a)$ and $T_{kj} = a$, for all $j \in \{1, \dots, m\}$. Therefore:

$$\begin{aligned} x_l[x_j := \overline{C_j}(T_{j1}, \dots, T_{jm})]_{j=1}^{m} &= x_l[x_l := \overline{C_l}(a, \dots, a)] \\ &= \overline{C_l}(a, \dots, a) \\ &= R_{ik} \end{aligned}$$

   Otherwise, we have $S_{ik} = R_{ik}$, and the desired property follows immediately.

□

It remains to prove that $\mathcal{C}$ is satisfiable whenever $\langle L_\mathcal{C}, R_\mathcal{C}\rangle$ admits a solution. We first establish a technical lemma concerning the form of the possible solutions of $\langle L_\mathcal{C}, R_\mathcal{C}\rangle$.

Lemma 3.6. *If the equation $\langle L_\mathcal{C}, R_\mathcal{C}\rangle$ admits a solution then the variable $\mathbf{X}$ is substituted by a term of the form*

$$\lambda x_1 \dots x_m . f(U_1, \dots, U_n)$$

where the terms U_i are such that:

1. *either $U_i = x_k$ (for some $k \in \{1, \dots, m\}$), in which case $neg(a_i) = C_k$,*
2. *or $U_i = \overline{\mathrm{neg}(a_i)}(U_{i1}, \dots, U_{im})$ where the terms U_{ij} are such that:*
   (a) *either $U_{ij} = x_k$ (for some $k \in \{1, \dots, m\}$), in which case $C_k \in \mathrm{pos}(a_i)$,*
   (b) *or $U_{ij} = R_{ij}$.*

Proof. Suppose that

$$\begin{cases} \mathbf{X} & = & U \\ \mathbf{X}_{ij} & = & V_{ij} \end{cases}$$

is a solution to the syntactic equation $\langle L_\mathcal{C}, R_\mathcal{C}\rangle$. Then, we must have:

$$U(\overline{C_1}(V_{11}, \dots, V_{1m}), \dots \overline{C_m}(V_{m1}, \dots, V_{mm})) \quad \twoheadrightarrow_\beta \quad f(R_1, \dots, R_n)$$

This implies that U is indeed of the form

$$\lambda x_1 \dots x_m . f(U_1, \dots, U_n)$$

where for all $i \in \{1, \dots, n\}$:

$$U_i[x_j := \overline{C_j}(V_{j1}, \dots, V_{jm})]_{j=1}^m \quad \twoheadrightarrow_\beta \quad R_i$$

Now, the head of each U_i is either some λ-variable x_k or some constant. In the first case, $U_i = x_k$, and we must have that

$$\overline{C_k}(V_{k1}, \dots, V_{km}), \quad = \quad R_i$$

which implies that $\mathrm{neg}(a_i) = C_k$. In the second case, the head of U_i must be the head of R_i, which implies that U_i is of the form

$$\overline{\mathrm{neg}(a_i)}(U_{i1}, \dots, U_{im})$$

Moreover, we must have that

$$U_{ik}[x_j := \overline{C_j}(V_{j1}, \dots, V_{jm})]_{j=1}^m \quad \twoheadrightarrow_\beta \quad R_{ik}$$

Now, if the head of U_{ik} is some λ-variable x_l, we must have $U_{ik} = x_l$, and:

$$\overline{C_l}(V_{l1}, \dots, V_{lm}) = R_{ik}$$

This implies that $C_l \in \mathrm{pos}(a_i)$. Otherwise, we have

$$U_{ik} = R_{ik}.$$

□

We are now in a position of proving the second half of our reduction result.

Proposition 3.2. *Let $\mathcal{C}$ be a 1-neg-sat problem, and $\langle L_\mathcal{C}, R_\mathcal{C}\rangle$ be the associated syntactic equation. If $\langle L_\mathcal{C}, R_\mathcal{C}\rangle$ admits a solution, then $\mathcal{C}$ is satisfiable.*

Proof. According to Lemma 3.6, if $\langle L_\mathcal{C}, R_\mathcal{C}\rangle$ admits a solution, then te term U substituted for $\mathbf{X}$ is of the form

$$\lambda x_1 \dots x_m . f(U_1, \dots, U_n)$$

where:

1. either $U_i = x_k$, for some $k \in \{1, \dots, m\}$,
2. or $U_i = \overline{\mathrm{neg}(a_i)}(U_{i1}, \dots, U_{im})$.

We define a valuation η as follows:

$$\eta(a_i) = \begin{cases} 0 \text{ if } U_i = x_j \text{ for some } j \in \{1, \dots, m\} \\ 1 \text{ otherwise} \end{cases}$$

Now, for every clause C_j such that $x_j = U_i$, for some $i \in \{1, \dots, n\}$, we have, by Lemma 3.6, that $\mathrm{neg}(a_i) = C_j$, i.e., $\neg a_i \in C_j$. Consequently, these clauses are satisfied by η.

As for the other clauses C_k, since U is a term of the linear λ-calculus, there must exists some term U_{ij} such that $U_{ij} = x_k$. In this case, according to Lemma 3.6, $C_k \in \mathrm{pos}(a_i)$, i.e., $a_i \in C_k$. Consequently, these clauses are also satisfied by η. □

As a consequence of Lemma 3.1, and Propositions 3.1, and 3.2, we get the main theorem of this paper.

Theorem 3.1. *Linear second-order matching in the linear λ-calculus is NP-complete.*

3.6 Related results

The main difference between a context and a linear second-order λ-term is that the latter has the ability of rearranging its arguments in any order. This explains why linear second-order matching in the linear λ-calculus is NP-complete while linear context matching is not. Nevertheless, this difference is not significant when the arguments of the second-order meta-variable do not contain any first-order meta-variable. Consider a second-order equation of the form:

$$\mathbf{X}(T_1, \dots, T_n) = T$$

where $T_1, \dots, T_n$, and T are first-order pure linear λ-terms (such an equation is called an interpolation equation). It is not difficult to see that it may be solved in polynomial time. Indeed, it amounts to check whether the union of the multisets of the subterms of $T_1, \dots, T_n$ is included in the multiset of the subterms of T.

This polynomiality result, which is quite specific, cannot be generalized. Indeed, as we prove in the next proposition, interpolation in third-order case is NP-complete.

Proposition 3.3. *Third-order interpolation in the linear λ-calculus is NP-complete.*

Proof. The proof consists in a reduction of 1-Neg-sat that we obtain by reducing the equation $\langle L_{\mathcal{C}}, R_{\mathcal{C}} \rangle$ (as defined in section 3.4) to a third-order interpolation equation

Let $\mathcal{C} = \{C_1, \dots, C_m\}$ be a 1-neg-sat problem defined over the set of boolean variables $\mathcal{A} = \{a_1, \dots, a_n\}$. We build a third-order interpolation equation $\langle L, R \rangle$ which has a solution if and only if the equation $\langle L_{\mathcal{C}}, R_{\mathcal{C}} \rangle$. From Propositions 3.1 and 3.2, this is equivalent to say that $\langle L, R \rangle$ has a solution if and only if $\mathcal{C}$ is satisfiable. Therefore, from Lemma 3.1, we obtain that third-order interpolation is NP-complete problem.

We first define

$$\begin{cases} L = Y(\lambda x_1 \dots x_m . C_1(x_1, \dots, x_m), \dots, \lambda x_1 \dots x_m . C_m(x_1, \dots, x_m)) \\ R = R_{\mathcal{C}} \end{cases}$$

Then it remains to prove that $\langle L, R_{\mathcal{C}} \rangle$ has a solution if and only if $\langle L_{\mathcal{C}}, R_{\mathcal{C}} \rangle$ has a solution.

Suppose $\langle L_{\mathcal{C}}, R_{\mathcal{C}} \rangle$ has a solution :

$$\begin{cases} \mathbf{X} = U \\ \mathbf{X}_{ij} = V_{ij} \end{cases}$$

then the term

$$S = \lambda y_1 \dots y_m . U(y_1(V_{11}, \dots, V_{1m}), \dots, y_m(V_{m1}, \dots, V_{mm}))$$

is a solution of $\langle L, R_{\mathcal{C}} \rangle$. Indeed:

$$\begin{array}{ll} S(\lambda x_1 \dots x_m . C_1(x_1, \dots, x_m), \dots, \lambda x_1 \dots x_m . C_m(x_1, \dots, x_m)) & \twoheadrightarrow_\beta \\ U(C_1(V_{11}, \dots, V_{1m}), \dots, C_m(V_{m1}, \dots, V_{mm})) & \twoheadrightarrow_\beta \quad L_{\mathcal{C}} \end{array}$$

Conversely if $\langle L, R_{\mathcal{C}}\rangle\ [Y := S]$, then $S = \lambda y_1 \dots y_m.S'$ and one can find terms of linear λ-calculus U, V_{11}, $\dots$, V_{1m},$\dots$,V_{m1}, $\dots$, V_{mm} such that:

$$U(y_1(V_{11}, \dots, V_{1m}), \dots, y_m(V_{m1}, \dots, V_{mm})) \twoheadrightarrow_\beta S'$$

and then

$$\begin{cases} \mathbf{X} & = & U \\ \mathbf{X}_{ij} & = & V_{ij} \end{cases}$$

is obviously a solution of $\langle L_{\mathcal{C}}, R_{\mathcal{C}}\rangle$. □

Bibliography

Comon, H.: 1998a, Completion of rewrite systems with membership constraints. Part I: Deduction rules, *Journal of Symbolic Computation* **25**(4), 397–420.

Comon, H.: 1998b, Completion of rewrite systems with membership constraints. Part II: Constraint solving, *Journal of Symbolic Computation* **25**(4), 421–454.

Cook, S. A.: 1971, The complexity of theorem-proving procedures, *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, ACM Press, New York, NY, USA, pp. 151–158.

de Groote, P.: 2000, Linear higher-order matching is np-complete, *in* L. Bachmair (ed.), *Rewriting Techniques and Applications, RTA'00*, Vol. 1833 of *LNCS*, Springer, pp. 127–140.

Dougherty, D. J. and Wierzbicki, T.: 2002, A decidable variant of higher order matching, *RTA '02: Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, Springer-Verlag, London, UK, pp. 340–351.

Dowek, G.: 1994, Third order matching is decidable, *Annals of Pure and Applied Logic* **69**(2–3), 135–155.

Dowek, G.: 2001, Higher-order unification and matching, *in* A. Robinson and A. Voronkov (eds), *Handbook of Automated Reasoning*, Vol. 2, Elsevier Science, chapter 16.

Goldfarb, W. D.: 1981, The undecidability of the second-order unification problem, *Theoretical Computer Science* **13**(2), 225–230.

Huet, G. P.: 1973, The undecidability of unification in third order logic, *Information and Control* **22**(3), 257–267.

Huet, G. P.: 1976, *Résolution d'équations dans les langages d'ordre 1, 2, $\dots$, ω*, PhD thesis, Université Paris, 7.

Kilpeläinen, P. and Mannila, H.: 1995, Ordered and unordered tree inclusion, *SIAM J. Comput.* **24**(2), 340–356.

Levy, J.: 1996, Linear second-order unification, *in* H. Ganzinger (ed.), *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA-96)*, Vol. 1103 of *LNCS*, Springer-Verlag, Berlin, pp. 332–346.

Loader, R.: 2003, Higher order β matching is undecidable, *Logic Journal of the IGPL* **11**(1), 51–68.

Padovani, V.: 1996, *Filtrage d'ordre supérieur*, PhD thesis, Université de Paris 7.

Salvati, S. and De Groote, P.: 2003, On the complexity of higher-order matching in the linear λ-calculus, *in* R. Nieuwenhuis (ed.), *International Conference on Rewriting Techniques and Applications - RTA'2003, Valencia, Spain*, Vol. 2706 of *Lecture notes in Computer Science*, pp. 234–245.

Schmidt-Schauß, M. and Stuber, J.: 2001, On the complexity of linear and stratified context matching problems, *Technical Report A01-R-411*, LORIA.

Chapter 4

On the expressive power of Abstract Categorial Grammars: Representing context-free formalisms

Ph. de Groote and S. Pogodalla, 2004

We show how to encode context-free string grammars, linear context-free tree grammars, and linear context-free rewriting systems as Abstract Categorial Grammars. These three encodings share the same constructs, the only difference being the interpretation of the composition of the production rules. It is interpreted as a first-order operation in the case of context-free string grammars, as a second-order operation in the case of linear context-free tree grammars, and as a third-order operation in the case of linear context-free rewriting systems. This suggest the possibility of defining an Abstract Categorial Hierarchy.

4.1 Introduction

Abstract Categorial Grammars (ACGs) (de Groote 2001) are a new categorial formalism based on Girard linear logic (Girard 1987). This formalism, which sticks to the spirit of current type-logical grammars (Carpenter 1997, Moortgat 1996, Morrill 1994, Oehrle 1994), offers the following features:

- Every ACG generates two languages, an abstract language and an object language. The abstract language may be seen as a set of abstract grammatical structures, and of the object language as the set of concrete forms generated from these abstract structures. Consequently, one has a direct control on the parse structures of the grammar.

- The langages generated by the ACGs are sets of linear λ-terms, which generalizes both string-langages and tree-langages.

- ACGs are based on a small set of mathematical primitives that combine via simple composition rules. Consequently, ACGs offer a rather flexible framework.

Abstract Categorial Grammars are not intended to be yet another grammatical formalism that would compete with other well-established formalisms. They should rather be seen as the kernel of a grammatical framework — in the spirit of (Ranta 2004) — in which other existing grammatical models may be encoded. In this paper, we illustrate this fact by exploring the expressive power of ACGs. We show how to encode three context-free formalisms (namely, context-free string grammars, linear context-free tree grammars, and linear context-free rewriting systems) as ACGs.

The paper is organized as follows. In the next section, we introduce the notion of Abstract Categorial Grammar. Section 4.3 gives a natural encoding of strings as linear λ-terms. In Section 4.4, we remind the reader of the definitions of a context-free string grammar, a linear context-free tree grammar, and a linear context-free rewritng system. In Section 4.5, we explain how to encode context-free derivations. Then, Sections 4.6, 4.7 and 4.8 give

the encodings of context-free string grammars, linear context-free tree grammars, and linear context-free rewriting systems, respectively. Finally, we conclude in Section 4.9.

4.2 Abstract Categorial Grammars

This section gives the definition of an Abstract Categorial Grammar, which is based on the notions of *linear implicative types*, *higher-order linear signature*, and *linear λ-terms* built upon a higher-order linear signature.

Let A be a set of atomic types. The set $\mathscr{T}(A)$ of *linear implicative types* built upon A is inductively defined as follows:

1. if $a \in A$, then $a \in \mathscr{T}(A)$;
2. if $\alpha, \beta \in \mathscr{T}(A)$, then $(\alpha \multimap \beta) \in \mathscr{T}(A)$.

We use the usual convention of right association of the parentheses, i.e., we write $\alpha \multimap \beta \multimap \gamma \multimap \delta$ for $(\alpha \multimap (\beta \multimap (\gamma \multimap \delta)))$. We also write $\alpha^n \multimap \beta$ for

$$\underbrace{\alpha \multimap \cdots \multimap \alpha}_{n\times} \multimap \beta.$$

A *higher-order linear signature* consists of a triple $\Sigma = \langle A, C, \tau \rangle$, where:

1. A is a finite set of atomic types;
2. C is a finite set of constants;
3. $\tau : C \to \mathscr{T}(A)$ is a function that assigns to each constant in C a linear implicative type in $\mathscr{T}(A)$.

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

1. if $c \in C$, then $c \in \Lambda(\Sigma)$;
2. if $x \in X$, then $x \in \Lambda(\Sigma)$;
3. if $x \in X$, $t \in \Lambda(\Sigma)$, and x occurs free in t exactly once, then $(\lambda x.\, t) \in \Lambda(\Sigma)$;
4. if $t, u \in \Lambda(\Sigma)$, and the sets of free variables of t and u are disjoint, then $(t\, u) \in \Lambda(\Sigma)$.

$\Lambda(\Sigma)$ is provided with the usual notion of capture avoiding substitution, and the relations of α-conversion, β-reduction, β-conversion, and $\beta\eta$-conversion (Barendregt 1984), this latter relation being used as the notion of equality between λ-terms. We use the usual conventions when writing λ-terms: $t\, u_1\, u_2 \cdots u_n$ will stand for $(\cdots((t\, u_1)\, u_2) \cdots u_n)$, and $\lambda x_1 \ldots x_n.\, t$ for $\lambda x_1 \ldots .\, \lambda x_n.\, t$. Moreover, when $\mathbf{x}$ denotes a sequence of λ-variables $x_1, \ldots, x_n$, we write $\lambda \mathbf{x}.\, t$ for $\lambda x_1 \ldots x_n.\, t$.

Given a higher-order linear signature $\Sigma = \langle A, C, \tau \rangle$, each linear λ-term in $\Lambda(\Sigma)$ may be assigned a linear implicative type in $\mathscr{T}(A)$. This type assignment obeys an inference system whose judgements are sequents of the following form:

$$\Gamma \vdash_\Sigma t : \alpha$$

where:

1. Γ is a finite set of λ-variable typing declarations of the form '$x : \beta$' (with $x \in X$ and $\beta \in \mathscr{T}(A)$), such that any λ-variable is declared at most once;
2. $t \in \Lambda(\Sigma)$;
3. $\alpha \in \mathscr{T}(A)$.

The axioms and inference rules are the following:

$$\vdash_\Sigma c : \tau(c) \quad \text{(cons)}$$

$$x : \alpha \vdash_\Sigma x : \alpha \quad \text{(var)}$$

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

$$\frac{\Gamma \vdash_\Sigma t : (\alpha \multimap \beta) \quad \Delta \vdash_\Sigma u : \alpha}{\Gamma, \Delta \vdash_\Sigma (t\, u) : \beta} \quad \text{(app)}$$

Given two higher-order linear signatures Σ_1 and Σ_2, we define a *lexicon* $\mathscr{L} : \Sigma_1 \to \Sigma_2$ to be a realization of Σ_1 into Σ_2, i.e., an interpretation of the atomic types of Σ_1 as types built upon Σ_2 together with an interpretation of the constants of Σ_1 as linear λ-terms built upon Σ_2. These two interpretations must be such that their homomorphic extensions commute with the typing relations. This is spelled out in the next definition.

Definition 4.1. *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 linear signatures. A lexicon $\mathscr{L}$ from Σ_1 to Σ_2 is defined to be a pair $\mathscr{L} = \langle F, G \rangle$ such that:*

1. *$F : A_1 \to \mathscr{T}(A_2)$ is a function that interprets the atomic types of Σ_1 as linear implicative types built upon A_2;*
2. *$G : C_1 \to \Lambda(\Sigma_2)$ is a function that interprets the constants of Σ_1 as linear λ-terms built upon Σ_2;*
3. *the interpretation functions are compatible with the typing relation, i.e., for any $c \in C_1$, the following typing judgement is derivable:*
$$\vdash_{\Sigma_2} G(c) : \hat{F}(\tau_1(c)),$$
where $\hat{F}$ is the unique homomorphic extension of F.

In the sequel, given such a lexicon $\mathscr{L} = \langle F, G \rangle$, $\mathscr{L}(a)$ will stand for either $\hat{F}(a)$ or $\hat{G}(a)$, according to the context.

We are now in a position of defining the notion of Abstract Categorial Grammar.

Definition 4.2. *An Abstract Categorial Grammar is a quadruple $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ where:*

1. *Σ_1 and Σ_2 are two higher-order linear signatures; they are called the abstract vocabulary and the object vocabulary, respectively;*
2. *$\mathscr{L} : \Sigma_1 \to \Sigma_2$ is a lexicon from the abstract vocabulary to the object vocabulary;*
3. *s is an atomic type of the abstract vocabulary; it is called the distinguished type of the grammar.*

Every ACG $\mathscr{G}$ generates two languages: an *abstract language*, $\mathcal{A}(\mathscr{G})$, and an object language $\mathcal{O}(\mathscr{G})$.

The abstract language, which may be seen as a set of abstract parse structures, is the set of closed linear λ-terms built upon the abstract vocabulary and whose type is the distinguished type of the grammar.

Definition 4.3. *Let $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ be an Abstract Categorial Grammar. The abstract language $\mathcal{A}(\mathscr{G})$, generated by $\mathscr{G}$ is defined as follows:*

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

On the other hand, the *object language*, which may be seen as the set of concrete forms generated by the grammar, is defined to be the image of the abstract language by the term homomorphism induced by the lexicon.

Definition 4.4. *Let $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ be an Abstract Categorial Grammar. The object language $\mathcal{O}(\mathscr{G})$, generated by $\mathscr{G}$ is defined as follows:*

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

4.3 Strings as linear λ-terms

We are concerned, in this paper, with the representation of grammatical formalisms that generate strings. We must, therefore, specify a higher-order linear signature that allows strings to be defined and manipulated. This signature will serve as the object vocabulary of the several ACGs we will define.

There is, in fact, a canonical way of representing strings as linear λ-terms. It consists of encoding a string of symbols as a composition of functions. Consider, for instance, a string such as '$abbac$'. It may be represented by the linear λ-term:

$$\lambda x.\, a\,(b\,(b\,(a\,(c\,x)))),$$

where the atomic strings 'a', 'b', and 'c' are declared to be constants of functional type.

More formally, the higher-order linear signature corresponding to an alphabet obeys the following definition.

Definition 4.5. *let $T = \{a_1, \ldots, a_n\}$ be an alphabet. The higher-order linear signature, $\Sigma_T = \langle A, C, \tau\rangle$, is defined as follows:*

1. $A = \{\sigma\}$*;*
2. $C = \{a_1, \ldots, a_n\}$*;*
3. $\tau(a_i) = (\sigma \multimap \sigma)$*, for all* $1 \leq i \leq n$*.*

Given such a signature, the empty word (ϵ) is represented by the identity function ($\lambda x.\, x$), and concatenation is defined to be functional composition ($\lambda f.\, \lambda g.\, \lambda x.\, f\,(g\,x)$), which is indeed an associative operator that admits the identity function as a unit.

We define *string* to be the type $(\sigma \multimap \sigma)$, and λ-terms of type *string*, such as $\lambda x.\, a\,(b\,(b\,(a\,(c\,x))))$, will be written $/abbac/$. Finally, the infix operator $+$ will denote the composition (i.e., the concatenation) of such λ-terms.

4.4 Three context-free formalisms

In this section, we remind the reader of the definitions of the grammatical formalisms we intend to encode as ACGs.

4.4.1 Context-free string grammars

A context-free string grammar is a quadruple $G = \langle N, T, P, s\rangle$ where:

1. N is a finite set of symbols called the alphabet of non-terminal symbols;
2. T is a finite set of symbols, disjoined from N, called the alphabet of terminal symbols;
3. P is a finite set of production rules of the form $a \rightarrow \alpha$, where $a \in N$, and $\alpha \in (N \cup T)^*$;
4. $s \in N$ is called the start symbol of the grammar.

Given two words $\alpha, \beta \in (N \cup T)^*$, one says that β is directly derivable from α if and only if there exist $\beta_1, \beta_2, \beta_3 \in (N \cup T)^*$ and $a \in N$ such that:

1. $a \rightarrow \beta_2$ is a production rule of P;
2. $\alpha = \beta_1 a \beta_3$;
3. $\beta = \beta_1 \beta_2 \beta_3$.

This relation of direct derivability is written $\alpha \Rightarrow \beta$ and, as usual, $\Rightarrow^*$ denotes the reflexive, transitive closure of $\Rightarrow$. Finally, the language generated by G is defined to be the set of terminal words $\alpha \in T^*$ such that $s \Rightarrow^* \alpha$.

4.4.2 Linear context-free tree grammars

A ranked alphabet is defined to be a pair $\Sigma = \langle F_\Sigma, r_\Sigma \rangle$ such that F_Σ is a finite set of symbols, and $r_\Sigma : F_\Sigma \to \mathbb{N}$ is a function that assigns to each symbol a natural number called its rank. By a slight abuse of notation, we will write $a \in \Sigma$ for $a \in F_\Sigma$

Given such a ranked alphabet Σ, and a possibly inifinite countable set of variables X, the set of trees $T_\Sigma(X)$ is inductively defined as follows:

1. $X \subset T_\Sigma(X)$;
2. if $f \in \Sigma$ and $r_\Sigma(f) = 0$ then $f \in T_\Sigma(X)$;
3. if $f \in \Sigma$, $r_\Sigma(f) = n$, and $t_1, \ldots, t_n \in T_\Sigma(X)$ then $f(t_1, \ldots, t_n) \in T_\Sigma(X)$.

In case X is the empty set, the set of trees $T_\Sigma(\varnothing)$ is simply written T_Σ. Let $X_n = \{x_1, \ldots, x_n\}$ be a finite set of variables. A tree $t \in T_\Sigma(X_n)$ that contains exactly one occurrence of each variable x_i ($1 \leq i \leq n$) is called a n-context. Let t be such a n-context, and let $u_1, \ldots, u_n \in T_\Sigma$. We write $t[u_1, \ldots, u_n]$ to denote the tree obtained from t by replacing $x_1, \ldots, x_n$ by $u_1, \ldots, u_n$, respectively. The set of n-contexts built upon a given ranked alphabet Σ, will be written $C_\Sigma(n)$. Strictly speaking, the notion of n-context should not depend on the choice of the set X_n. Nevertheless, in the sequel, we will use the following convention: if t is a n-context then t and $t[x_1, \ldots, x_n]$ denote the same tree.

Let X be a set of variables, let Σ be a ranked alphabet, and let Σ_0 be the set of symbols $a \in \Sigma$ such that $r_\Sigma(a) = 0$. To each tree $t \in T_\Sigma(X)$, one associates its yield $\overline{t}$, which is a string over Σ_0, inductively defined as follows:

1. $\overline{x} = x$, for $x \in X$;
2. $\overline{a} = a$, for $a \in \Sigma_0$;
3. $\overline{f(t_1, \ldots, t_n)} = \overline{t_1} \ldots \overline{t_n}$.

A linear context-free tree grammar is a quadruple $G = \langle N, T, P, s \rangle$ where:

1. N is a ranked alphabet of non-terminal symbols;
2. T is a ranked alphabet of terminal symbols, disjoined from N;
3. P is a finite set of production rules of the form $a(x_1, \ldots, x_n) \to t[x_1, \ldots, x_n]$, where $a \in N$, $r_N(a) = n$, the variable $x_1, \ldots, x_n$ are all distinct, and $t \in C_{N \cup T}(n)$.
4. the start symbol $s \in N$ is such that $r_N(s) = 0$.

Let $u, v \in T_{N \cup T}$. v is directly derivable from u ($u \Rightarrow v$) if and only if there exist $c \in C_{N \cup T}(1)$, $a \in N$ with $r_N(a) = n$, $t \in C_{N \cup T}(n)$, and $u_1, \ldots, u_n \in T_{N \cup T}$ such that:

1. $a(x_1, \ldots, x_n) \to t[x_1, \ldots, x_n]$ is a production rule of P;
2. $u = c[a(u_1, \ldots, u_n)]$;
3. $v = c[t[u_1, \ldots, u_n]]$.

The tree language generated by G is then defined to be the set of terminal trees $t \in T_T$ such that $s \Rightarrow^* t$, where $\Rightarrow^*$ stands for the reflexive, transitive closure of $\Rightarrow$.

Note that the tree language generated by a linear context-free tree grammar is not sensitive to the derivation mode. This is due to the linearity condition which derives from the fact that the right-hand side of a production rule is restricted to be a context rather than an arbitrary tree. Consequently, the usual distinction between *outside-in* and *inside-out* tree languages does not apply in the present case.

In this paper, we are interested in string languages rather than in tree languages. Consequently, we will focus on the yield language generated by a linear context-free tree grammar, i.e., the set of strings α such that $\alpha = \overline{t}$ for some tree $t \in T_T$ such that $s \Rightarrow^* t$. In the general case, the class of yield languages generated by the context-free tree grammars corresponds to the class of indexed languages. In our case, because of the linearity constraint, the class of yield languages we consider is much more restrictive. To the best of our knowledge, whether this class corresponds to a class of languages definable by some other well-established formalism is an open question. Nevertheless, it is worth noting that it contains Joshi's Tree Adjoining Languages (Joshi and Schabes 1997) as a proper subclass (Mönnich 1997).

4.4.3 Linear context-free rewriting systems

Linear Context-free rewriting systems (Vijay-Shanker, Weir and Joshi 1987, Weir 1988) may be defined as a proper subclass of multiple context-free grammars (Seki, Matsumura, Fujii and Kasami 1991), which are themeselves a particular case of generalized context-free grammars (Pollard 1984). We do not follow this general approach here, but give a direct tailor-made definition, which is indeed equivalent to Weir's.

Let T be an alphabet, and consider a function $f : (T^*)^m \to (T^*)^n$ that acts on tuples of strings. Such a function is called a linear transform if and only if there exist

$$\alpha_{10}, \alpha_{11}, \ldots, \alpha_{1p_1}, \ldots, \alpha_{n0}, \alpha_{n1}, \ldots, \alpha_{np_n} \in T^*$$

such that:

$$f\langle x_1, \ldots, x_m\rangle = \langle \alpha_{10}x_{11}\alpha_{11} \ldots x_{1p_1}\alpha_{1p_1}, \ldots, \alpha_{n0}x_{n1}\alpha_{n1} \ldots x_{np_n}\alpha_{np_n}\rangle$$

where $\bigcup_{i=1}^{m}\{x_i\} = \bigcup_{i=1}^{n}\bigcup_{j=1}^{p_i}\{x_{ij}\}$, and $x_{ij} \neq x_{kl}$, whenever $i \neq k$ or $j \neq l$.

In the sequel, we work modulo the associativity of the cartesian product, i.e., we identify $(T^*)^n \times (T^*)^m$ with $(T^*)^{n+m}$ and, consequently, $\langle\langle\alpha_1, \ldots, \alpha_n\rangle, \langle\beta_1, \ldots, \beta_m\rangle\rangle$ with $\langle\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m\rangle$.

A linear context-free rewriting system is defined to be a quadruple $G = \langle N, T, P, s\rangle$ where:

1. N is a ranked alphabet of non-terminal symbols;
2. T is an alphabet of terminal symbols, disjoined from N;
3. P is a finite set of production rules of the form $\langle f, a \to \alpha\rangle$, where:
   (a) $a \in N$,
   (b) $\alpha = a_1 \ldots a_n \in N^*$,
   (c) f is a linear transform from $(T^*)^{\sum_{i=1}^{n} r_N(a_i)}$ into $(T^*)^{r_N(a)}$;
4. the start symbol $s \in N$ is such that $r_N(s) = 1$.

In Clause 3, the non-terminal word α is possibly empty, in which case the linear transform f degenerates into a constant tuple $f\langle\rangle$.

To each non-terminal symbol $a \in N$, one associates a set $L(a) \subset (T^*)^{r_N(a)}$, inductively defined as follows:

1. for each production rule $\langle f, a \to \epsilon\rangle$, where ϵ stands for the empty word, one has $f\langle\rangle \in L(a)$;
2. If $t_1 \in L(a_1), \ldots, t_n \in L(a_n)$, and $\langle f, a \to a_1 \ldots a_n\rangle$ is a production rule of P, then $f\langle t_1, \ldots, t_n\rangle \in L(a)$.

The language generated by G is then defined to be the set $L(s)$. Observe that this set is indeed a set of strings because $r_N(s) = 1$.

4.5 Specifying context-free derivations

In order to encode a formalism as an ACG, we have to give an abstract vocabulary, an object vocabulary, and a lexicon. The three formalisms of Section 4 generate string languages. Consequently, their object vocabulary will obey the construct of Definition 4.5. They will also share the same kind of abstract vocabulary, whose construction is explained in the present section.

Let $a \to \alpha$ be a production rule of a context-free string grammar. We define the skeleton of this rule to be the pair $\langle a, \lceil\alpha\rceil\rangle$, where $\lceil\alpha\rceil$ is a word of non-terminal symbols inductively defined as follows:

1. $\lceil b\rceil = \epsilon$, if b is a terminal symbol;
2. $\lceil b\rceil = b$, if b is a non-terminal symbol;
3. $\lceil b\beta\rceil = \lceil\beta\rceil$, if b is a terminal symbol;
4. $\lceil b\beta\rceil = b\lceil\beta\rceil$, if b is a non-terminal symbol.

Similarly, let $a(x_1, \ldots, x_n) \to t$ be a production rule of a context-free tree grammar. Its skeleton is defined to be the pair $\langle a, \lceil t\rceil\rangle$, where $\lceil t\rceil$ is inductively defined as follows:

1. $\lceil x_i \rceil = \epsilon$, for x_i a variable;
2. $\lceil f \rceil = \epsilon$, if f is a terminal symbol of rank 0;
3. $\lceil f \rceil = f$, if f is a non-terminal symbol of rank 0;
4. $\lceil f(t_1, \dots, t_n) \rceil = \lceil t_1 \rceil \dots \lceil t_n \rceil$, if f is a terminal symbol;
5. $\lceil f(t_1, \dots, t_n) \rceil = f \lceil t_1 \rceil \dots \lceil t_n \rceil$, if f is a non-terminal symbol.

finally, let $\langle f, a \to \alpha \rangle$ be a production rule of a linear context-free rewriting system. Its skeleton is defined to be the pair $\langle a, \alpha \rangle$.

To summarize, in the three cases, the skeleton of a production rule is a pair $\langle a, \alpha \rangle$, where a is the non-terminal symbol occurring in the left-hand side of the rule, and α is a word consisting of the non-terminal symbols occurring in its right-hand side.

This notion of skeleton of a production rule allows us to define the higher-order linear signature associated to a given context-free string grammar, linear context-free tree grammar, or linear context-free rewriting system.

Definition 4.6. *Let $G = \langle N, T, P, s \rangle$ be a context-free string grammar, a linear context-free tree grammar, or a linear context-free rewriting system. The higher-order linear signature $\Sigma_G = \langle A, C, \tau \rangle$, associated to G, is defined as follows:*

1. *$A = N$;*
2. *to each $p \in P$, one associates a constant c_p, and $C = \bigcup_{p \in P} \{c_p\}$;*
3. *$\tau(c_p) = a_1 \multimap \cdots a_n \multimap a$, where $\langle a, a_1 \dots a_n \rangle$ is the skeleton of rule p.*

It is not difficult to see that the closed λ-terms of atomic type built upon the above signature are regular trees that correspond to context-free parse trees.

4.6 Composition as first-order susbtitution

In order to define ACGs representing the formalisms of Section 4, it remains to specify appropriate lexicons. This section explains the construction of such lexicons in the case of context-free string grammars.

Let $G = \langle N, T, P, s \rangle$ be a context-free string grammar, and let $p \in P$ be the following production rule:

$$a \to \alpha_0 a_1 \alpha_1 \dots a_n \alpha_n$$

where $a, a_1, \dots, a_n \in N$ and $\alpha_0, \alpha_1, \dots, \alpha_n \in T^*$. The linear λ-term $[\![p]\!]$ is defined to be:

$$\lambda y_1 \dots y_n . /\alpha_0/ + y_1 + /\alpha_1/ + \cdots + y_n + /\alpha_n/$$

We now define the ACG corresponding to a given context-free string grammar.

Definition 4.7. *Let $G = \langle N, T, P, s \rangle$ be a context-free string grammar. The Abstract Categorial Grammar $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s \rangle$ is defined as follows:*

1. *the abstract vocabulary Σ_G is constructed according to Definition 4.6;*
2. *the object vocabulary Σ_T is constructed according to Definition 4.5;*
3. *the lexicon $\mathscr{L}_G : \Sigma_G \to \Sigma_T$ is such that:*
   (a) *$\mathscr{L}_G(a) = string$, for all $a \in N$*
   (b) *$\mathscr{L}_G(c_p) = [\![p]\!]$, for all $p \in P$;*
4. *the distinguished type s is identical to the start symbol of G.*

It remains to prove that the ACG constructed according to the above definition is indeed a correct representation of the corresponding context-free string grammar. This is established by the next two propositions.

Proposition 4.1. *Let $G = \langle N, T, P, s\rangle$ be a context-free string grammar, and let $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s\rangle$ be the Abstract Categorial Grammar constructed from G according to Definition 4.7.*

For all $a \in N$ and all $\alpha \in T^*$, if $a \Rightarrow^* \alpha$ then there exists a closed λ-term $t \in \Lambda(\Sigma_G)$ such that $\vdash_{\Sigma_G} t : a$ and $\mathscr{L}_G(t) = /\alpha/$.

Proof. We proceed by induction on the length of the derivation $a \Rightarrow^* \alpha$.

If $a \Rightarrow^* \alpha$ because of a production rule $a \rightarrow \alpha$, there must exist an abstract constant c corresponding to this production rule, which is of type a and such that $\mathscr{L}_G(c) = /\alpha/$.

Now, suppose that the first rule of the derivation is

$$a \rightarrow \alpha_0 a_1 \alpha_1 \dots a_n \alpha_n \tag{4.1}$$

Consequently, there exists $\beta_1, \dots, \beta_n \in T^*$ such that $a_i \Rightarrow^* \beta_i$ and $\alpha = \alpha_0 \beta_1 \alpha_1 \dots \beta_n \alpha_n$. Then, by induction hypothesis, there must exist closed λ-terms $t_1, \dots, t_n$ of type $a_1, \dots, a_n$, respectively, such that $\mathscr{L}_G(t_i) = /\beta_i/$. On the other hand, there exists an abstract constant c corresponding to (1), whose type is $a_1 \multimap \cdots a_n \multimap a$ and such that

$$\mathscr{L}_G(c) = \lambda y_1 \dots y_n. /\alpha_0/ + y_1 + /\alpha_1/ + \cdots + y_n + /\alpha_n/.$$

Consequently, we have that

$$\mathscr{L}_G(c\, t_1 \cdots t_n) = /\alpha_0/ + /\beta_1/ + /\alpha_1/ + \cdots + /\beta_n/ + /\alpha_n/ = /\alpha/.$$

□

Proposition 4.2. *Let $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s\rangle$ be the Abstract Categorial Grammar constructed from a given context-free string grammar $G = \langle N, T, P, s\rangle$, according to Definition 4.7.*

For all $a \in N$, and all closed λ-term $t \in \Lambda(\Sigma_G)$ such that $\vdash_{\Sigma_G} t : a$, there exists $\alpha \in T^*$ such that $\mathscr{L}_G(t) = /\alpha/$, and $a \Rightarrow^* \alpha$.

Proof. We proceed by induction on the structure of t. Note that, t being a closed term of atomic type, it is either a constant or an application.

If t is a constant then $t = c_p$ for some $p \in P$ whose skeleton is $\langle a, \epsilon\rangle$. Then, by definition of $\mathscr{G}_G$, p must be of the form $a \rightarrow \alpha$ with $\mathscr{L}_G(c_p) = /\alpha/$.

If t is an application then $t = c_p\, t_1 \cdots t_n$ for some $p \in P$ whose skeleton is $\langle a, a_1 \dots a_n\rangle$. In this case, each λ-term t_i must be a closed λ-term of type a_i, and p must be of the form

$$a \rightarrow \alpha_0 a_1 \alpha_1 \dots a_n \alpha_n$$

where $\alpha_0, \alpha_1, \dots, \alpha_n \in T^*$, and

$$\mathscr{L}_G(c_p) = \lambda y_1 \dots y_n. /\alpha_0/ + y_1 + /\alpha_1/ + \cdots + y_n + /\alpha_n/.$$

Then, by induction hypothesis, there exist $\beta_1, \dots, \beta_n \in T^*$ such that $\mathscr{L}_G(t_i) = /\beta_i/$ and $a_i \Rightarrow^* \beta_i$. This implies that

$$\mathscr{L}_G(t) = /\alpha_0/ + /\beta_1/ + /\alpha_1/ + \cdots + /\beta_n/ + /\alpha_n/,$$

and that $a \Rightarrow^* \alpha_0 \beta_1 \alpha_1 \dots \beta_n \alpha_n$. □

4.7 Composition as second-order susbtitution

In order to adapt the construction of the previous section to the case of linear context-free tree grammars, we will interpret the atomic types of the abstract vocabulary as second-order types over strings.

Let $G = \langle N, T, P, s\rangle$ be a linear context-free tree grammar, and let $p \in P$ be a production rule

$$a(x_1, \dots, x_n) \rightarrow t$$

whose skeleton is $\langle a, a_1 \dots a_m\rangle$. The linear λ-term $[\![p]\!]$ is defined to be:

$$\lambda y_1 \dots y_m. \lambda x_1 \dots x_n. |t|$$

where $|t|$ is inductively defined as follows:

1. $|x_i| = x_i$;
2. $|f| = /f/$, if f is a terminal symbol of rank 0;
3. $|a_i| = y_i$, if the non-terminal a_i is of rank 0;
4. $|f(t_1, \ldots, t_k)| = |t_1| + \cdots + |t_k|$, if f is a terminal symbol;
5. $|a_i(t_1, \ldots, t_k)| = y_i\, |t_1| \cdots |t_n|$.

Adapting Definition 4.7 to the case of linear context-free tree grammars is then straightforward.

Definition 4.8. *Let $G = \langle N, T, P, s\rangle$ be a linear context-free tree grammar. The Abstract Categorial Grammar $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s\rangle$ is defined as follows:*

1. *the abstract vocabulary Σ_G is constructed according to Definition 4.6;*
2. *the object vocabulary Σ_T is constructed according to Definition 4.5;*
3. *the lexicon $\mathscr{L}_G : \Sigma_G \to \Sigma_T$ is such that:*
   (a) *$\mathscr{L}_G(a) = string^{r_N(a)} \multimap string$, for all $a \in N$*
   (b) *$\mathscr{L}_G(c_p) = [\![p]\!]$, for all $p \in P$;*
4. *the distinguished type s is identical to the start symbol of G.*

In order to establish the correctness of the above construction, we first state two technical lemmas concerning the operator $|\cdot|$ used in the definition of $[\![p]\!]$. Their proofs, which consist of simple inductions, are left to the reader.

Lemma 4.1. *Let $G = \langle N, T, P, s\rangle$ be a linear context-free tree grammar. For all terminal tree $t \in T_T$, $|t| = /\bar{t}/$.*

Lemma 4.2. *Let $G = \langle N, T, P, s\rangle$ be a linear context-free tree grammar. Let $u, u_1, \ldots, u_n \in T_{N\cup T}$, $c \in C_{N\cup T}(1)$, $a_1, \ldots, a_m \in N$, and $t \in C_T(n)$ be such that:*

1. *$a_1, \ldots, a_m$ is the sequence of occurrences of non-terminal symbols in u;*
2. *$r_N(a_1) = n$;*
3. *$u = c[a_1(u_1, \ldots, u_n)]$;*

Then, $(\lambda y_1 \ldots y_m.\, |u|)\, (\lambda x_1 \ldots x_n.\, |t|) = \lambda y_2 \ldots y_m.\, |c[t[u_1, \ldots, u_n]]|$.

Proposition 4.3. *Let $G = \langle N, T, P, s\rangle$ be a linear context-free tree grammar, and let $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s\rangle$ be the Abstract Categorial Grammar constructed from G according to Definition 4.8.*

For all $a \in N$ such that $r_N(a) = n$, all $v \in C_T(n)$, and all $u_1, \ldots, u_n \in T_T$, if $a(u_1, \ldots, u_n) \Rightarrow^* v[u_1, \ldots, u_n]$ then there exists a closed λ-term $t \in \Lambda(\Sigma_G)$ such that $\vdash_{\Sigma_G} t : a$ and $\mathscr{L}_G(t)\, /\overline{u_1}/ \cdots /\overline{u_n}/ = /\overline{v[u_1, \ldots, u_n]}/$.

Proof. We proceed by induction on the length of the derivation $a(u_1, \ldots, u_n) \Rightarrow^* v[u_1, \ldots, u_n]$.

If $a(u_1, \ldots, u_n) \Rightarrow^* v[u_1, \ldots, u_n]$ because of a production rule $a(x_1, \ldots, x_n) \to v[x_1, \ldots, x_n]$, there exists an abstract constant corresponding to this rule, and we are done by taking t to be this abstract constant.

Now suppose that the first rule of the derivation is the production rule p, $a(x_1, \ldots, x_n) \to w[x_1, \ldots, x_n]$, whose skeleton is $\langle a, a_1 \ldots a_m\rangle$. Then, for all a_i there exist $c \in C_{N\cup T}(1)$, $w_{i1}, \ldots, w_{ir_N(a_i)} \in T_{N\cup T}$, $c' \in C_T(1)$, $w'_{i1}, \ldots, w'_{ir_N(a_i)} \in T_T$, and $v_i \in C_T(r_N(a_i))$ such that

1. $w(u_1, \ldots, u_n) = c[a_i(w_{i1}, \ldots, w_{ir_N(a_i)})]$;
2. $v(u_1, \ldots, u_n) = c'[v_i(w'_{i1}, \ldots, w'_{ir_N(a_i)})]$;
3. $c[s] \Rightarrow^* c'[s]$, for all $s \in T_{N\cup T}$;
4. $a_i(s_1, \ldots, s_{r_N(a_i)}) \Rightarrow^* v_i(s_1, \ldots, s_{r_N(a_i)})$, for all $s_1, \ldots, s_{r_N(a_i)} \in T_{N\cup T}$;

5. $w_{ij} \Rightarrow^* w'_{ij}$.

Therefore, by induction hypothesis, there exist closed λ-terms $t_1, \ldots, t_m \in \Lambda(\Sigma_G)$ such that $\vdash_{\Sigma_G} t_i : a_i$ and

$$\mathscr{L}_G(t_i)\,/\overline{s_1}/\,\cdots/\overline{s_{r_N(a_i)}}/ = /\overline{v_i[s_1, \ldots, s_{r_N(a_i)}]}/$$

for all $s_1, \ldots, s_{r_N(a_i)} \in T_{N \cup T}$. This implies that

$$\mathscr{L}_G(t_i) = \lambda x_1 \ldots x_{r_N(a_i)}.\,/\overline{v_i}/.$$

Then we take

$$t = c_p\, t_1\, \cdots t_m,$$

and the result follows by iterating Lemma 4.2. □

Proposition 4.4. *Let $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s\rangle$ be the Abstract Categorial Grammar constructed from a given context-free tree grammar $G = \langle N, T, P, s\rangle$, according to Definition 4.8.*

For all $a \in N$ such that $r_N(a) = n$, and all closed λ-term $t \in \Lambda(\Sigma_G)$ such that $\vdash_{\Sigma_G} t : a$, there exists a context $v \in C_T(n)$ such that, for all $u_1, \ldots, u_n \in T_T$, $\mathscr{L}_G(t)\,/\overline{u_1}/\,\cdots/\overline{u_n}/ = /\overline{v[u_1, \ldots, u_n]}/$, and $a(u_1, \ldots, u_n) \Rightarrow^* v[u_1, \ldots, u_n]$.

Proof. We proceed by induction on the structure of t.

If t is a constant then $t = c_p$ for some $p \in P$ whose skeleton is $\langle a, \epsilon\rangle$. Consequently, p must be of the form $a(x_1, \ldots, x_n) \to w[x_1, \ldots, x_n]$ with $w \in C_T(n)$. On the one hand, we have that

$$a(u_1, \ldots, u_n) \Rightarrow w[u_1, \ldots, u_n].$$

On the other hand, by Lemma 4.1,

$$\mathscr{L}_G(c_p) = \lambda x_1 \ldots x_n.\,/\overline{w}/,$$

which implies that

$$\mathscr{L}_G(c_p)\,/\overline{u_1}/\,\cdots/\overline{u_n}/ = /\overline{w[u_1, \ldots, u_n]}/.$$

If t is an application then $t = c_p\, t_1\, \cdots t_m$ for some $p \in P$ whose skeleton is $\langle a, a_1 \ldots a_m\rangle$, and each λ-term t_i must be a closed λ-term of type a_i. Consequently, by induction hypothesis, there exist contexts $v_1, \ldots, v_m$ such that $v_i \in C_T(r_N(a_i))$ and, for all all $u_1, \ldots, u_{r_N(a_i)} \in T_T$,

$$\mathscr{L}_G(t_i)\,/\overline{u_1}/\,\cdots/\overline{u_{r_N(a_i)}}/ = /\overline{v_i[u_1, \ldots, u_{r_N(a_i)}]}/,$$

and

$$a(u_1, \ldots, u_{r_N(a_i)}) \Rightarrow^* v_i[u_1, \ldots, u_{r_N(a_i)}].$$

This implies that

$$\mathscr{L}_G(t_i) = \lambda x_1 \ldots x_{r_N(a_i)}.\, v_i[x_1, \ldots, x_{r_N(a_i)}],$$

and the result follows by iterating Lemma 4.2 and applying Lemma 4.1. □

4.8 Composition as third-order substitution

Finally, in this section, we define the ACG corresponding to a linear context-free rewriting system. To this end, we interpret the atomic types of the abstract vocabulary as third-order types over strings.

Let $G = \langle N, T, P, s\rangle$ be a linear context-free rewriting system, and let $p \in P$ be a production rule $\langle f, a \to a_1 a_2 \ldots a_l\rangle$, whose linear transform obeys the following equation:

$$f\langle x_1, \ldots, x_m\rangle = \langle \alpha_{10} x_{11} \alpha_{11} \ldots x_{1p_1} \alpha_{1p_1}, \ldots, \alpha_{n0} x_{n1} \alpha_{n1} \ldots x_{np_n} \alpha_{np_n}\rangle.$$

We define the λ-terms $u_1, \ldots, u_n$ as follows:

$$u_i = /\alpha_{i0}/ + x_{i1} + /\alpha_{i1}/ + \cdots + x_{ip_i} + /\alpha_{ip_i}/$$

The linear λ-term $[\![p]\!]$ is then defined to be:

$$\lambda y_1\, y_2 \ldots y_l.\, \lambda z.\, y_1\, (\lambda \mathbf{x}_1.\, y_2\, (\lambda \mathbf{x}_2.\, \cdots y_l\, (\lambda \mathbf{x}_l.\, z\, u_1\, \cdots u_n)))$$

where:

- $\mathbf{x}_1$ is the sequence of λ-variables $x_1, \ldots, x_{r_N(a_1)}$,
- $\mathbf{x}_2$ is the sequence of λ-variables $x_{r_N(a_1)+1}, \ldots, x_{r_N(a_1)+r_N(a_2)}$,
- etc.

Then, the ACG corresponding to a given linear context-free rewriting system is defined as follows.

Definition 4.9. *Let $G = \langle N, T, P, s\rangle$ be a linear context-free rewriting system. The Abstract Categorial Grammar $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s\rangle$ is defined as follows:*

1. *the abstract vocabulary Σ_G is constructed according to Definition 4.6;*
2. *the object vocabulary Σ_T is constructed according to Definition 4.5;*
3. *the lexicon $\mathscr{L}_G : \Sigma_G \to \Sigma_T$ is such that:*
   (a) *$\mathscr{L}_G(a) = (string^{r_N(a)} \multimap string) \multimap string$, for all $a \in N$*
   (b) *$\mathscr{L}_G(c_p) = [\![p]\!]$, for all $p \in P$;*
4. *the distinguished type s is identical to the start symbol of G.*

In order to prove the correctness of the above construction, we start by stating a technichal lemma, whose proof is left to the reader.

Lemma 4.3. *Let $G = \langle N, T, P, s\rangle$ be a linear context-free rewriting system, let $p \in P$ be the production rule $\langle f, a \to a_1 \ldots a_n\rangle$, and let $\alpha_{i1}, \ldots, \alpha_{ir_N(a_i)} \in T^*$, for $1 \le i \le n$. Then, there exists $\alpha_1, \ldots, \alpha_{r_N(a)} \in T^*$ such that*

1. $f\langle \alpha_{11}, \ldots, \alpha_{1r_N(a_1)}, \ldots, \alpha_{n1}, \ldots, \alpha_{nr_N(a_n)}\rangle = \langle \alpha_1, \ldots, \alpha_{r_N(a)}\rangle$
2. $[\![p]\!]\,(\lambda z.\, z\,/\alpha_{11}/\, \cdots /\alpha_{1r_N(a_1)}/) \cdots (\lambda z.\, z\,/\alpha_{n1}/\, \cdots /\alpha_{nr_N(a_n)}/)$
   $= \lambda z.\, z\,/\alpha_1/\, \cdots /\alpha_{r_N(a)}/$

Proposition 4.5. *Let $G = \langle N, T, P, s\rangle$ be a linear context-free rewriting system, and let $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s\rangle$ be the Abstract Categorial Grammar constructed from G according to Definition 4.9.*

For all $a \in N$ such that $r_N(a) = n$, and all $\alpha_1, \ldots, \alpha_n \in T^*$, if $\langle \alpha_1, \ldots, \alpha_n\rangle \in L(a)$ then there exists a closed λ-term $t \in \Lambda(\Sigma_G)$ such that $\vdash_{\Sigma_G} t : a$ and $\mathscr{L}_G(t) = \lambda z.\, z\,/\alpha_1/\, \cdots /\alpha_n/$.

Proof. We proceed by induction on the definition of $L(a)$.

If $f\langle\rangle = \langle \alpha_1, \ldots, \alpha_n\rangle \in L(a)$ because there exists a production rule p of the form $\langle f, a \to \epsilon\rangle$, there exists an abstract constant c_p of type a such that $\mathscr{L}_G(c_p) = \lambda z.\, z\,/\alpha_1/\, \cdots /\alpha_n/$.

Now suppose that $\langle \alpha_1, \ldots, \alpha_n\rangle \in L(a)$ because there exists a production rule $\langle f, a \to a_1 \ldots a_m\rangle$, together with tuples $\langle \alpha_{i1}, \ldots, \alpha_{ir_N(a_i)}\rangle \in L(a_i)$, such that

$$f\langle \alpha_{11}, \ldots, \alpha_{1r_N(a_1)}, \ldots, \alpha_{m1}, \ldots, \alpha_{mr_N(a_m)}\rangle = \langle \alpha_1, \ldots, \alpha_n\rangle.$$

Hence, by induction hypothesis, there exist closed λ-terms $t_1, \ldots, t_m \in \Lambda(\Sigma_G)$ such that $\vdash_{\Sigma_G} t_i : a_i$ and $\mathscr{L}_G(t_i) = \lambda z.\, z\,/\alpha_{i1}/\, \cdots /\alpha_{ir_N(a_i)}/$. Then, the result follows by Lemma 4.3. □

Proposition 4.6. *Let $\mathscr{G}_G = \langle \Sigma_G, \Sigma_T, \mathscr{L}_G, s\rangle$ be the Abstract Categorial Grammar constructed from a given linear context-free rewriting system $G = \langle N, T, P, s\rangle$, according to Definition 4.9.*

For all $a \in N$ such that $r_N(a) = n$, and all closed λ-term $t \in \Lambda(\Sigma_G)$ such that $\vdash_{\Sigma_G} t : a$, there exist $\alpha_1, \ldots, \alpha_n \in T^*$ such that $\mathscr{L}_G(t) = \lambda z.\, z\,/\alpha_1/\, \cdots /\alpha_n/$, and $\langle \alpha_1, \ldots, \alpha_n\rangle \in L(a)$.

Proof. We proceed by induction on the structure of t.

If t is a constant then $t = c_p$ for some $p \in P$ whose skeleton is $\langle a, \epsilon\rangle$. Consequently, p must be of the form $\langle f, a \to \epsilon\rangle$. Then, there exist $\alpha_1, \ldots, \alpha_n \in T^*$ such that $f\langle\rangle = \langle \alpha_1, \ldots, \alpha_n\rangle$. Hence, by definition, $\mathscr{L}_G(c_p) = \lambda z.\, z\,/\alpha_1/\, \cdots /\alpha_n/$, and $\langle \alpha_1, \ldots, \alpha_n\rangle \in L(a)$.

If t is an application then $t = c_p\, t_1 \cdots t_m$ for some $p \in P$ whose form is $\langle f, a \to a_1 \ldots a_m\rangle$, and each λ-term t_i must be a closed λ-term of type a_i. Therefore, by induction hypothesis, there exist $\alpha_{i1}, \ldots, \alpha_{ir_N(a_i)} \in T^*$ such that $\mathscr{L}_G(t_i) = \lambda z.\, z\,/\alpha_{i1}/\, \cdots /\alpha_{ir_N(a_i)}/$, and $\langle \alpha_{i1}, \ldots, \alpha_{ir_N(a_i)}\rangle \in L(a_i)$. Then, the result follows by Lemma 4.3. □

Observe that we do not have that $\alpha \in L(s)$ if and only if $/\alpha/ \in \mathcal{O}(\mathscr{G}_G)$. We have instead that $\alpha \in L(s)$ if and only if $\lambda z.\, z\,/\alpha/ \in \mathcal{O}(\mathscr{G}_G)$. This possible defect can be easily fixed by changing the distinguished type of the grammar to be a new abstract atomic type s', and by adding a new abstract constant c of type $s \multimap s'$. The lexicon is then extended in such a way that $\mathscr{L}_G(s') = \mathit{string}$ and $\mathscr{L}_G(c) = \lambda y.\, y\,(\lambda x.\, x)$.

4.9 Conclusions

The embeding of context-free string grammars, linear context-free tree grammars, and linear context-free rewriting systems in Abstract Categorial Grammars exemplifies some of the features of the ACG framework.

The fact that an ACG generates two languages offer an explicit control of the parse structure of the grammar. Consequently, the three encodings we have given are in fact strong equivalences.[1].

The fact that the basic objects manipulated by an ACG are linear λ-terms allows higher-order operations to be defined. Typically, tree-adjunction is such a higher-order operation (Abrusci et al. 1999, Joshi and Kulick 1997, Mönnich 1997), and we have seen that the possibility of defining such higher-order operations is the keystone in encoding linear context-free tree grammars and linear context-free rewriting systems.

Finally, the fact that the embeddings of the three context-free formalisms are based, respectively, on first-order, second-order, and third-order interpretations suggests the existence of an Abstract Categorial Hierarchy that would allow the expressive power of the ACGs to be controlled.

Bibliography

Abrusci, M., Fouqueré, C. and Vauzeilles, J.: 1999, Tree-adjoining grammars in a fragment of the Lambek calculus, *Computational Linguistics* **25**(2), 209–236.

Barendregt, H. P.: 1984, *The lambda calculus, its syntax and semantics*, North-Holland. Revised edition.

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

de Groote, P.: 2001, Towards abstract categorial grammars, *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155.

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

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

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

Joshi, A. K. and Kulick, S.: 1997, Partial proof trees as building blocks for a categorial grammar, *Linguistics and Philosophy* **20**(6), 637–667.

Joshi, A. K. and Schabes, Y.: 1997, *Tree-adjoining grammars*, Vol. 3 of *G. Rozenberg and A. Salomaa, editors, Handbook of formal languages*, Springer, chapter 2.

Mönnich, U.: 1997, Adjunction as substitution, *in* G.-J. Kruijff, G. Morrill and R. Oehrle (eds), *Proceedings of Formal Grammar*, pp. 169–178.

Moortgat, M.: 1996, Categorial type logics, *in* J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, Elsevier Science Publishers, Amsterdam, pp. 93–177.

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

Oehrle, R. T.: 1994, Term-labeled categorial type systems, *Linguistic and Philosophy* **17**.

[1]In the case of linear context-free tree grammars, this claim might be discussed because our encoding does not give access to the tree-language generated by the linear context-free tree grammar. This possible problem may be fixed by defining the embedding in two stages as it is done in (de Groote 2002)

Pollard, C.: 1984, *Generalized Phrase Structure Grammars, Head Grammars, and Natural Language*, PhD thesis, Stanford University, CA.

Ranta, A.: 2004, Grammatical Framework, *Journal of Functional Programming* **14**(2), 145–189.

Seki, H., Matsumura, T., Fujii, M. and Kasami, T.: 1991, On multiple context-free grammars, *Theoretical Computer Science* **223**, 87–120.

Vijay-Shanker, K., Weir, D. J. and Joshi, A. K.: 1987, Characterizing structural descriptions produced by various grammatical formalisms, *Proceedings of the 25th ACL*, Stanford, CA, pp. 104–111.

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

Chapter 5

Computing Semantic Representation: Towards ACG Abstract Terms as Derivation Trees

S. Pogodalla, 2004a

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

Introduction

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

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

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

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

5.1 ACG principles

An ACG $\mathscr{G}$ defines:

1. two sets of typed λ-terms: Λ_1 (based on the typed constant set C_1) and Λ_2 (based on the typed constant set C_2);
2. a morphism $\mathscr{L} : \Lambda_1 \to \Lambda_2$;
3. a distinguished type **S**.

(de Groote 2001) defines both Λ_1 and Λ_2 as sets of *linear* λ-terms. In this paper, we use simply typed λ-terms for Λ_2, using the translation of intuitionnistic logic into linear logic $A \to B = (!A) \multimap B$ (Girard 1987, Danos and Cosmo 1992). We don't elaborate on that subject in this paper, but it does not change the main properties of ACGs[1]. Then the abstract language $\mathcal{A}(\mathscr{G})$ and the object language $\mathcal{O}(\mathscr{G})$ are defined as follows:

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

Note that $\mathscr{L}$ binds the parse structures in $\mathcal{A}(\mathscr{G})$ to the concrete expressions of $\mathcal{O}(\mathscr{G})$. Depending on the choice of Λ_1, Λ_2 and $\mathscr{L}$, it can map for instance derivation trees and derived trees for TAGs (de Groote 2002), derivation trees of context-free grammars and strings of the generated language (de Groote 2001), derivation trees of m-linear context-free rewriting systems and strings of the generated language (de Groote and Pogodalla 2003). Of course, this link between an abstract and a concrete structure can apply not only to syntactical formalisms, but also to semantic formalisms.

The main point here is that ACGs can be mixed in different ways: in a transversal way, were two ACGs use the same abstract language, or in a compositional way, were the abstract language of an ACG is the object language of an other one. In this paper, as described in figure 5.1, we use different ACGs and some composition with $\mathscr{G}$: Λ_2 is the tree language of TAGs, Λ_1 the tree language of our *derivation trees*. For $\mathscr{G}'$, we have the same abstract language and Λ'_2 is the underspecified representation language. In dotted lines is a composition presented in (de Groote 2001) between strings and derivation trees we do not use here.

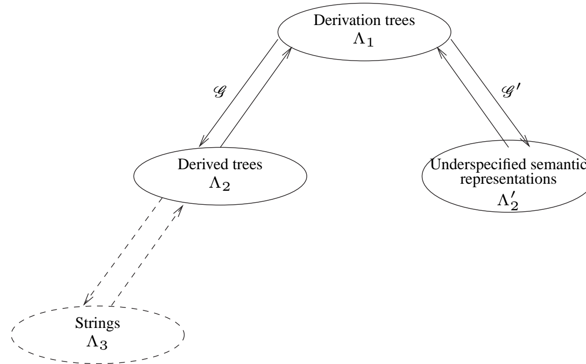


Figure 5.1: Moving from an object language to another

5.2 TAGs as ACGs

This section refers to (de Groote 2002), which proposes to encode TAGs into ACGs. Given a TAG G, Λ_1 is build as follows:

- for every non-termninal symbol X, there are two types X_S and X_A standing for places where substitution and adjunction can occur respectively;

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

- for every elementary tree γ, there is a constant $c_\gamma \in C_1$. Moreover, for every non-terminal symbol X, there is a constant $I_X : X_A$.

For instance, given the trees of table 5.1, we have the constants and their types (for concision, we suppress parameters that are not used in the next examples of this paper, namely nodes where no adjunction occur[2]):

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

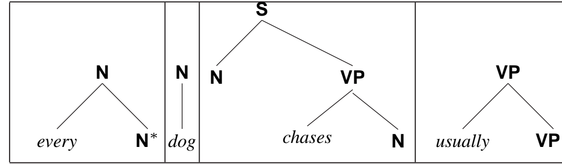


Table 5.1: Examples of elementary trees

To completely define the ACG $\mathscr{G}$, we need to define Λ_2 and $\mathscr{L}$. The types of Λ_2 are made of the single type τ, representing the type of trees. For any non-terminal symbol X, there are constants $X_0, \ldots, X_i$ where i is the maximal number of children of the X nodes in the elementary trees. For any terminal symbol X in G, there is a constant $X : \tau \in C_2$. Then $\mathscr{L}$ is defined by sending any X_S type to the type τ, and any X_A types to the type $\tau \multimap \tau$. Corresponding to the trees of table 5.1, we have for instance:

$$\begin{array}{rl} \mathscr{L}(I_X) & = \lambda x.x : \tau \multimap \tau \\ \mathscr{L}(c_{every}) & = \lambda x.\mathbf{N}_2(\mathbf{every}\ x) : \tau \multimap \tau \\ \mathscr{L}(c_{dog}) & = \lambda N.N(\mathbf{N}_1 \mathbf{dog}) : (\tau \multimap \tau) \multimap \tau \\ \mathscr{L}(c_{chases}) & = \lambda SV.\lambda x.\lambda y.S(\mathbf{S}_2 x(V \\ & \quad (\mathbf{VP}_2 \mathbf{chases}\ y))) \\ & \quad : (\tau \multimap \tau) \multimap (\tau \multimap \tau) \multimap \tau \multimap \tau \multimap \tau \end{array}$$

Note that in the adjunction operation, the auxiliary tree is a parameter. But it also has a higher-order type, that is a function from trees to trees. We let the reader check that $\mathscr{L}(c_{chases} I_{\mathbf{S}} I_{\mathbf{VP}}(c_{dog} c_{every})(c_{cat} c_{some}))$ correspond the derived tree associated to *every dog chases some cat* of figure 5.2.

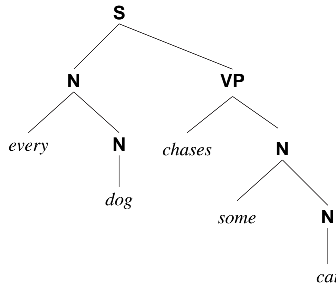


Figure 5.2: $\mathscr{L}(c_{chases} I_{VP}(c_{dog} c_{every})(c_{cat} c_{some})) = \mathbf{S}_2(\mathbf{N}_2\ every\ (\mathbf{N}_1\ dog\ ))(\mathbf{VP}_2\ chases\ (\mathbf{N}_2\ some\ (\mathbf{N}_1\ cat\ )))$

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

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

5.3 Semantic representation for TAGs as ACGs

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

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

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

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

Both these formulas have the property that:

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

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

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

illustrated in figure 5.3. The syntax of URL is basically the same that first-order logic, except that if atomic formulas remain the same, formulas are built from *holes* and *labels*, the latter being used as place holder for logical formulas in the underspecified representation language. We use the usual logical symbols ($\exists$, $\wedge$), an infix predicate $\geq$ to specify the constraints and an infix operateur : for URL. The symbol $h \geq l$ imposes the constraint for a formula that is associated to l to be a subformula of the one associated to h. $l : p$ indicates that a predicate p of SRL is labelled in URL by l.

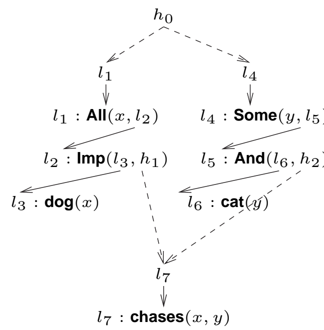


Figure 5.3: URL formula for *every dog chases a cat*

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

Coming back to the figure 5.1, we established in the previous section the $\mathscr{G}$ ACG to encode TAGs. We know want to rely on the common abstract language, Λ_1, the one of derivation trees, to build the $\mathscr{G}'$ ACG that model the semantic behaviour, with URL as Λ'_2. So let us now define $\mathscr{G}'$.

First is Λ'_2:

- the types we use are e, h, l, p, t where e stands for entities, h for holes, l for labels, p for predicate of the logical language and t for truth values;
- the constants are $\geq$, $:$, $\exists_l$, $\exists_e$, $\exists_h$, $\wedge$, **Imp**, **And**, **Some**, **All** and the set of the predicate symbols of the logical language (**dog**, **chases**, etc. in the examples). Their types are described in table 5.2.

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

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

Table 5.2: Typing of constants of Λ'_2

Finally, to define the ACG $\mathscr{G}'$, we need the lexicon $\mathscr{L}'$. It transforms the types from Λ_1 as follows:

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

Contrary to Λ_2, that model derived trees and generates linear terms, we use in Λ'_2 non-linear terms, as the intuitionnistic $\to$ shows. The definition of $\mathscr{L}'$ on the terms justifies it. We shall introduce this definition in the next sections, illustrating different linguistic phenomena.

5.3.1 Quantification

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

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

$$\begin{aligned}
\mathscr{L}'(c_{dog}) &= \lambda q.q(\lambda xhl.h \geq l \wedge l : \mathbf{dog}(x)) \\
\mathscr{L}'(c_{cat}) &= \lambda q.q(\lambda xhl.h \geq l \wedge l : \mathbf{cat}(x)) \\
\mathscr{L}'(c_{chases}) &= \lambda baso.s(b(\lambda x.a(o(\lambda yh'l'.h' \geq l' \\
&\quad \wedge l' : \mathbf{chases}(x, y))))) \\
\mathscr{L}'(c_{every}) &= \lambda rp.\lambda hl.\exists h_1 l_1 l_2 l_3 v_1 (h \geq l_2 \\
&\quad \wedge l_2 : \mathbf{All}(v_1, l_3) \wedge l_3 : \mathbf{Imp}(l_1, h_1) \\
&\quad \wedge h_1 \geq l \wedge r\, v_1\, h\, l_1 \wedge p\, v_1\, h\, l) \\
\mathscr{L}'(c_{some}) &= \lambda rp.\lambda h'l'.\exists h'_1 l'_1 l'_2 l'_3 v'_1 (h' \geq l'_2 \\
&\quad \wedge l'_2 : \mathbf{Ex}(v'_1, l'_3) \wedge l'_3 : \mathbf{And}(l'_1, h'_1) \\
&\quad \wedge h'_1 \geq l' \wedge r\, v'_1\, h'\, l'_1 \wedge p\, v'_1\, h'\, l')
\end{aligned}$$

It's easy to check that the translation from the abstract term, or the derivation tree in our sense:

$$t = c_{chases} I_{\mathsf{S}} I_{\mathsf{VP}}(c_{dog} c_{every})(c_{cat} c_{some})$$

by $\mathscr{L}'$ has the expected form:

$$\begin{aligned}
\mathscr{L}'(c_{dog} c_{every}) &= \lambda p.\lambda hl.\exists h_1 l_1 l_2 l_3 v_1 (h \geq l_2 \\
&\quad \wedge l_2 : \textbf{All}(v_1, l_3) \wedge l_3 : \textbf{Imp}(l_1, h_1) \\
&\quad \wedge h_1 \geq l \wedge h \geq l_1 \wedge l_1 : \textbf{dog}(v_1) \\
&\quad \wedge p\, v_1\, h\, l) \\
\mathscr{L}'(c_{cat} c_{some}) &= \lambda p.\lambda h' l'.\exists h'_1 l'_1 l'_2 l'_3 v'_1 (h' \geq l'_2 \\
&\quad \wedge l'_2 : \textbf{Ex}(v'_1, l'_3) \wedge l'_3 : \textbf{And}(l'_1, h'_1) \\
&\quad \wedge h'_1 \geq l' \wedge h' \geq l'_1 \wedge l'_1 : \textbf{cat}(v'_1) \\
&\quad \wedge p\, v'_1\, h'\, l') \\
\mathscr{L}'(c_{chases} I_{\mathsf{S}} I_{\mathsf{VP}}) &= \lambda so.s(\lambda x.o(\lambda y h' l'.h' \geq l' \\
&\quad \wedge l' : \textbf{chases}(x, y)))
\end{aligned}$$

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

$$\begin{aligned}
&\lambda hl.\exists h_1 l_1 l_2 l_3 v_1 (h \geq l_2 \wedge l_2 : \textbf{All}(v_1, l_3) \\
&\wedge l_3 : \textbf{Imp}(l_1, h_1) \wedge h_1 \geq l \wedge h \geq l_1 \wedge l_1 : \textbf{dog}(v_1) \\
&\wedge \exists h'_1 l'_1 l'_2 l'_3 v'_1 (h \geq l'_2 \wedge l'_2 : \textbf{Ex}(v'_1, l'_3) \\
&\wedge l'_3 : \textbf{And}(l'_1, h'_1) \wedge h'_1 \geq l \wedge h \geq l'_1 \wedge l'_1 : \textbf{cat}(v'_1) \\
&\quad \wedge h \geq l \wedge l : \textbf{chases}(v_1, v'_1)))
\end{aligned}$$

recovering the one from the figure 5.3 (modulo variable renaming). To deal with quantification in this example, we don't add any extra-link to the derivation tree (or abstract term) ones, contrary to (Kallmeyer 2002). Both the subject (the s variable in $\mathscr{L}'(c_{chases})$) and the object parameter (the o variable) are considered as the real functors, applyed to the relation **chases** as in $s(\cdots(o(\cdots \textbf{chases}(x, y) \cdots)))$. This implies that **N**s and **NP**s have higher-order types (see also the semantic term associated to entities in section 5.3.4). This is reminiscent to Montague's approach (Montague 1974).

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

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

5.3.2 Adverbs

In the semantic representation we associate to c_{chases} in the previous section, we see, between the subject s and the "**VP** relation", an argument a. Its type ($\mathscr{L}'(\mathsf{VP}_A) = (h \to l \to t) \multimap (h \to l \to t)$) shows it is a verb modifier. So let us introduce a new consant $c_{usually} : \mathsf{VP}_A \multimap \mathsf{VP}_A \in C_1$. We can associate it, with $\mathscr{L}'$, to the term:

$$\begin{aligned}
&\lambda a.\lambda r.\lambda hl.\exists h_1 l_1 (r\, h\, l \wedge h \geq l_1 \wedge l_1 : \textbf{U}(h_1) \\
&\wedge h_1 \geq l \wedge a(\lambda h' l'.h' \geq l') h\, l_1)
\end{aligned}$$

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

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

$$\begin{aligned}
&\lambda r.\lambda hl.\exists h_1 l_1 (r\, h\, l \wedge h \geq l_1 \wedge l_1 : \textbf{U}(h_1) \\
&\wedge h_1 \geq l \wedge \exists h'_1 l'_1 (h \geq l_1 \wedge h \geq l'_1 \wedge l'_1 : \textbf{A}(h'_1) \\
&\wedge h'_1 \geq l_1))
\end{aligned}$$

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

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

5.3.3 Raising verbs

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

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

5.3.4 Verbs with phrasal arguments

Going upward in the syntactic tree, we can now try to model expressions that act on $\mathbf{S}$ nodes like *claims* (see table 5.3). Coming back to our modelling of *chases*, we had a b argument of type $\mathscr{L}'(\mathbf{S}_A) = (e \to h \to l \to t) \multimap (e \to h \to l \to t)$. So we can associate to a term $c_{claims} : \mathbf{N}_S \multimap \mathbf{S}_A \multimap \mathbf{S}_A \in \Lambda_1$ a term in Λ'_2:

$$\lambda spr.\lambda y.p(s(\lambda xhl.\exists l_1 h_1(h \geq l_1 \wedge \\ l_1 : \mathbf{claims}(x, h_1) \wedge ryh_1 l)))$$

which specifies that x claims something, the latter being dominated by h_1 (hence $\mathbf{claims}$).

So for instance, an expression *Paul claims John loves Mary* would give the abstract term:

$$c_{loves}(c_{claims} c_{Paul} I_{\mathbf{S}}) I_{\mathbf{VP}} c_{John} c_{Mary}$$

and its underspecified representation ($\mathscr{L}'(c_{Paul}) = \lambda P.P\mathbf{p}$):

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

because

$$\begin{aligned} \mathscr{L}'(c_{claims} c_{Paul}) &= \lambda r.\lambda y.\lambda hl.\exists l_1 h_1 (h \geq l_1 \\ &\quad \wedge l_1 : \mathbf{claims}(\mathbf{p}, h_1) \wedge ryh_1 l) \\ \mathscr{L}'(c_{loves} t I_{\mathbf{VP}} \\ c_{John} c_{Mary}) &= (\lambda P.P\mathbf{j})(t(\lambda x.(\lambda Q.Q\mathbf{m}) \\ &\quad (\lambda yh'l'.h' \geq l' \wedge \\ &\quad l' : \mathbf{loves}(x, y)))) \\ &= (\lambda P.P\mathbf{j})(t(\lambda x.\lambda h'l'.h' \geq l' \\ &\quad \wedge l' : \mathbf{loves}(x, \mathbf{m}))) \end{aligned}$$

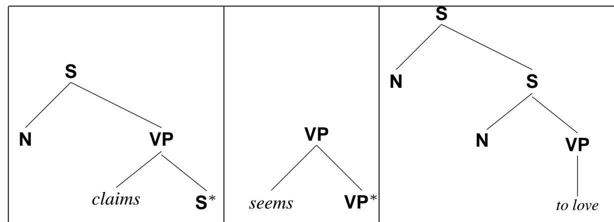


Table 5.3: Few more trees

Let us now illustrate the long distance dependancy behaviour, together with phrasal arguments. We can see that if the syntactic properties of the infinitive *to love* (see table 5.3) really differs from the ones of *loves*, their semantic counterpart only differs in the order of argument (and an extra $\mathscr{L}'(\mathbf{S}_A)$ whose role should be precised). We can naturally associate to $\mathscr{L}'(c_{to\ love})$ the term:

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

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

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

which is the expected result.

5.3.5 Wh-questions

This section provides an example of an adjunction occurring on the root node of an auxiliary tree which is itself adjoined to a third tree. The expression *who does Paul think John said Bill liked*, can be analyzed with the constants $c_{who} : \mathbf{WH}_S \in \Lambda_1$ and $c_{liked} : \mathbf{S}_A \multimap \mathbf{VP}_A \multimap \mathbf{WH}_S \multimap \mathbf{N}_S \multimap \mathbf{S}_S \in \Lambda_1$, that correspond to the trees of figure 5.4. The two other constants $c_{does\ think}$ and c_{said}, corresponds to the auxiliary trees of the same figure and the derivation tree is $c_{liked}(c_{said}c_{John}(c_{does\ think}c_{Paul}I_{\mathbf{S}}))I_{\mathbf{VP}}c_{who}c_{Bill}$.

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

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

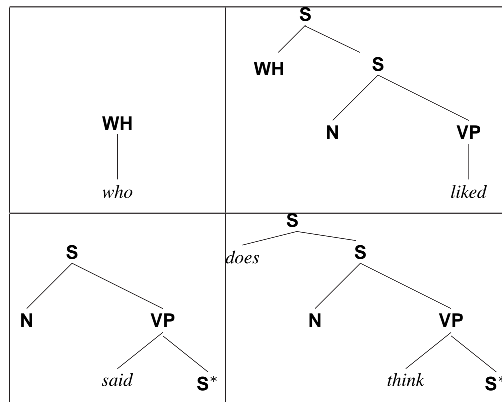


Figure 5.4: Wh-question example

Then, we have :

$$\begin{aligned}
\mathscr{L}'(c_{\textit{does think}} c_{\textit{Paul}} I_{\mathbf{S}}) &= \mathscr{L}'(t_0) \\
&= \lambda r' \lambda yhl.\exists l'_1 h'_1 (h \geq l'_1 \\
&\quad \wedge l'_1 : \mathbf{T}(\mathbf{p}, h'_1) \wedge h'_1 \geq l \\
&\quad \wedge r' y h'_1 l) \\
\mathscr{L}'(c_{\textit{said}} c_{\textit{John}} t_0 I_{\mathbf{S}}) &= \mathscr{L}'(t_1) \\
&= \lambda r.\lambda yhl.\exists l'_1 h'_1 (h \geq l'_1 \\
&\quad \wedge l'_1 : \mathbf{T}(\mathbf{p}, h'_1) \wedge h'_1 \geq l \\
&\quad \wedge \exists h_1 l_1 (h'_1 \geq l_1 \wedge l_1 : \mathbf{S}(\mathbf{j}, h_1) \\
&\quad \wedge h_1 \geq l \wedge r y h_1 l))
\end{aligned}$$

This yields the following result:

$$\begin{aligned}
\mathscr{L}'(c_{\textit{liked}} t_1 I_{\mathbf{VP}} c_{\textit{who}} c_{\textit{Bill}}) &= (\lambda o.o(\lambda yhl.\exists l'_1 h'_1 (h \geq l'_1 \\
&\quad \wedge l'_1 : \mathbf{T}(\mathbf{p}, h'_1) \wedge h'_1 \geq l \\
&\quad \wedge \exists h_1 l_1 (h'_1 \geq l_1 \wedge l_1 : \mathbf{S}(\mathbf{j}, h_1) \\
&\quad \wedge h_1 \geq l \wedge h_1 \geq l \\
&\quad \wedge l : \mathbf{liked}(\mathbf{b}, y))))) \mathscr{L}'(c_{\textit{who}}) \\
&= \lambda hl.\exists v_1 h''_1 l''_1 (h \geq l''_1 \\
&\quad \wedge l''_1 : \mathbf{W}(v_1, h''_1) \wedge h''_1 \geq l \\
&\quad \wedge \exists l'_1 h'_1 (h''_1 \geq l'_1 \\
&\quad \wedge l'_1 : \mathbf{T}(\mathbf{p}, h'_1) \wedge h'_1 \geq l \\
&\quad \wedge \exists h_1 l_1 (h'_1 \geq l_1 \wedge l_1 : \mathbf{S}(\mathbf{j}, h_1) \\
&\quad \wedge h_1 \geq l \wedge h_1 \geq l \\
&\quad \wedge l : \mathbf{liked}(\mathbf{b}, v_1))))
\end{aligned}$$

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

5.3.6 Control verbs

Control verbs, as presented in (Gardent and Kallmeyer 2003) or (Frank and van Genabith 2001), with adjunction on a **S** node (see table 5.4) to produce an expression like *John tries to sleep*, with the adjunction of *tries to* on *sleep*, is a problem for our approach.

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

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

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

Conclusion

We propose to reconsider semantic representation computation for TAG from the derivation trees. But derivation trees here are understood as abstract terms of ACGs, even if the informations born by each of the formalism are

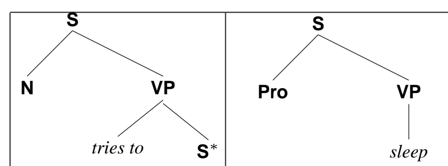


Table 5.4: Derived trees for control verbs

essentially the same. Whereas they hold the specification of how trees should combine, the locality of computing the meanings held by the different nodes is described in the object vocabulary. It obviates the addition of extra links to manage scoping and shows that derivation trees, by themselves, are enough, even if further investigation are required to handle control verbs.

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

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

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

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

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

Bibliography

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

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

Bos, J.: 1995, Predicate logic unplugged, *Proceedings of the Tenth Amsterdam Colloquium*.

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

Danos, V. and Cosmo, R. D.: 1992, The linear logic primer, `http://www.pps.jussieu.fr/~dicosmo/CourseNotes/LinLog/`. An introductory course on Linear Logic.

de Groote, P.: 2001, Towards abstract categorial grammars, *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155.

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

de Groote, P. and Pogodalla, S.: 2003, m-linear context-free rewriting systems as abstract categorial grammars, *in* R. Oehrle and J. Rogers (eds), *MOL 8, proceedings of the eighth Mathematics of Language Conference*.

Frank, A. and van Genabith, J.: 2001, Glue tag: Linear logic based semantics construction for ltag - and what it teaches us about the relation between lfg and ltag, *in* M. Butt and T. H. King (eds), *Proceedings of the LFG '01 Conference*, Online Proceedings, CSLI Publications. `http://cslipublications.stanford.edu/LFG/6/lfg01.html`.

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

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

Joshi, A. K., Kallmeyer, L. and Romero, M.: 2003, Flexible composition in ltag: Quantifier scope and inverse linking, *in* H. Bunt, I. van der Sluis and R. Morante (eds), *Proceedings of the Fifth International Workshop on Computational Semantics IWCS-5*.

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

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

Pogodalla, S.: 2004a, Computing semantic representation: Towards ACG abstract terms as derivation trees, *Proceedings of the Seventh International Workshop on Tree Adjoining Grammar and Related Formalisms (TAG+7)*, pp. 64–71.

Pogodalla, S.: 2004b, Using and extending the ACG technology: Endowing categorial grammars with an underspecified semantic representation, *Proceedings of Categorial Grammars 2004, Montpellier*, pp. 197–209.

Portner, P. and Partee, B. H. (eds): 2002, *Formal Semantics: The Essential Readings*, Blackwell Publishers.

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

Chapter 6

Further Readings

The ACG web page is located at `http://www.loria.fr/equipes/calligramme/acg/` and links to many papers related to ACGs.

Bibliography

de Groote, P. and Salvati, S.: 2004, Higher-order matching in the linear lambda-calculus with pairing, *in* A. T. Jerzy Marcinkowski (ed.), *18th International Workshop on Computer Science Logic - CSL'2004, Karpacz, Poland*, Vol. 3210 of *Lecture notes in Computer Science*, Springer, pp. 220–234.

Muskens, R.: 2001, Lambda Grammars and the Syntax-Semantics Interface, *in* R. van Rooy and M. Stokhof (eds), *Proceedings of the Thirteenth Amsterdam Colloquium*, Amsterdam, pp. 150–155.

Muskens, R.: 2003, Lambdas, Language, and Logic, *in* G.-J. Kruijff and R. Oehrle (eds), *Resource Sensitivity in Binding and Anaphora*, Studies in Linguistics and Philosophy, Kluwer, pp. 23–54.

Pogodalla, S.: 2004, Using and extending the ACG technology: Endowing categorial grammars with an underspecified semantic representation, *Proceedings of Categorial Grammars 2004, Montpellier*, pp. 197–209.

Pollard, C.: 2004, High-order categorical grammars, *Proceedings of Categorial Grammars 2004, Montpellier*, pp. 340–361.

Yoshinaka, R. and Kanazawa, M.: 2005, The complexity and generative capacity of lexicalized abstract categorial grammars, *in* P. Blache, E. Stabler, J. Busquets and R. Moot (eds), *Proceedings of Logical Aspects of Computational Linguistics: 5th International Conference, LACL 2005*, Vol. 3492 of *LNCS*, Springer-Verlag GmbH, pp. 330–348.

Bibliography

Chapter 1

Abrusci, M., Fouqueré, C. and Vauzeilles, J.: 1999, Tree-adjoining grammars in a fragment of the Lambek calculus, *Computational Linguistics* **25**(2), 209–236.

Barendregt, H. P.: 1984, *The lambda calculus, its syntax and semantics*, North-Holland. Revised edition.

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

Dalrymple, M., Lamping, J., Pereira, F. and Saraswat, V.: 1995, Linear logic for meaning assembly, *in* G. Morrill and R. Oehrle (eds), *Proceedings of Formal Gramma*, pp. 75–93.

de Groote, P.: 2000, Linear higher-order matching is np-complete, *in* L. Bachmair (ed.), *Rewriting Techniques and Applications, RTA'00*, Vol. 1833 of *LNCS*, Springer, pp. 127–140.

de Groote, P.: 2001, Towards abstract categorial grammars, *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155.

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

Joshi, A. K. and Schabes, Y.: 1997, *Tree-adjoining grammars*, Vol. 3 of *G. Rozenberg and A. Salomaa, editors, Handbook of formal languages*, Springer, chapter 2.

Lambek, J.: 1958, The mathematics of sentence structure, *American Mathematical Monthly* **65**(3), 154–170.

Merenciano, J. M. and Morrill, G. V.: 1997, Generation as deduction on labelled proof nets, *in* C. Retoré (ed.), *Proceedings of LACL-96*, Vol. 1328 of *LNAI*, Springer.

Montague, R.: 1970a, English as a formal language, *Linguaggi nella Società e nella Tecnica*, Edizioni di Comunità, Milan, pp. 189–224.

Montague, R.: 1970b, Universal grammar, *Theoria* **36**, 373–398.

Montague, R.: 1973, The proper treatment of quantification in ordinary english, *in* J. Hintikka (ed.), *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*, D. Reidel Publishing Co., Dordrecht, Holland, pp. 221–242. Reprinted in *Formal Philosophy*, by Richard Montague, Yale University Press, New Haven, CT, 1974, pp. 247–270.

Montague, R.: 1974, *Formal Philosophy: Selected Papers of Richard Montague*, Yale University Press, New Haven, CT. edited and with an introduction by Richmond Thomason.

Moortgat, M.: 1996, Categorial type logics, *in* J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, Elsevier Science Publishers, Amsterdam, pp. 93–177.

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

Nivat, M.: 1965, Transduction des langages de chomsky, *Annales de l'Institut Fourier* **18**, 339–455.

Oehrle, R. T.: 1994, Term-labeled categorial type systems, *Linguistic and Philosophy* **17**.

Pentus, M.: 1993, Lambek grammars are context free, *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Montreal, Canada, pp. 429–433.

Pogodalla, S.: 2000, Generation, lambek calculus, montague's semantics and semantic proof nets, *proceedings of the International Conference on Computational Linguistics*. *http://www.xrce.xerox.com/Publications/Attachments/2000-006/pogodalla-coling2000.pdf

Prawitz, D.: 1965, *Natural Deduction, A Proof-Theoretical Study*, Almqvist & Wiksell, Stockholm.

Ranta, A.: 1994, *Type Theoretical Grammar*, Oxford University Press.

van Benthem, J.: 1986, *Essays in Logical Semantics*, Reidel, Dordrecht.

Chapter 2

Abrusci, M., Fouqueré, C. and Vauzeilles, J.: 1999, Tree-adjoining grammars in a fragment of the Lambek calculus, *Computational Linguistics* **25**(2), 209–236.

Barendregt, H. P.: 1984, *The lambda calculus, its syntax and semantics*, North-Holland. Revised edition.

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

de Groote, P.: 2001, Towards abstract categorial grammars, *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155.

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

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

Joshi, A. K. and Kulick, S.: 1997, Partial proof trees as building blocks for a categorial grammar, *Linguistics and Philosophy* **20**(6), 637–667.

Joshi, A. K. and Schabes, Y.: 1997, *Tree-adjoining grammars*, Vol. 3 of *G. Rozenberg and A. Salomaa, editors, Handbook of formal languages*, Springer, chapter 2.

Mönnich, U.: 1997, Adjunction as substitution, *in* G.-J. Kruijff, G. Morrill and R. Oehrle (eds), *Proceedings of Formal Grammar*, pp. 169–178.

Moortgat, M.: 1996, Categorial type logics, *in* J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, Elsevier Science Publishers, Amsterdam, pp. 93–177.

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

Oehrle, R. T.: 1994, Term-labeled categorial type systems, *Linguistic and Philosophy* **17**.

Ranta, A.: 2002, Grammatical Framework: A type-theoretical grammar formalism, *Journal of Functional Programming* . To appear. Manuscript available at `http://www.cs.chalmers.se/articles/gf-jfp.ps.gz`.

Chapter 3

Comon, H.: 1998a, Completion of rewrite systems with membership constraints. Part I: Deduction rules, *Journal of Symbolic Computation* **25**(4), 397–420.

Comon, H.: 1998b, Completion of rewrite systems with membership constraints. Part II: Constraint solving, *Journal of Symbolic Computation* **25**(4), 421–454.

Cook, S. A.: 1971, The complexity of theorem-proving procedures, *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, ACM Press, New York, NY, USA, pp. 151–158.

de Groote, P.: 2000, Linear higher-order matching is np-complete, *in* L. Bachmair (ed.), *Rewriting Techniques and Applications, RTA'00*, Vol. 1833 of *LNCS*, Springer, pp. 127–140.

Dougherty, D. J. and Wierzbicki, T.: 2002, A decidable variant of higher order matching, *RTA '02: Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, Springer-Verlag, London, UK, pp. 340–351.

Dowek, G.: 1994, Third order matching is decidable, *Annals of Pure and Applied Logic* **69**(2–3), 135–155.

Dowek, G.: 2001, Higher-order unification and matching, *in* A. Robinson and A. Voronkov (eds), *Handbook of Automated Reasoning*, Vol. 2, Elsevier Science, chapter 16.

Goldfarb, W. D.: 1981, The undecidability of the second-order unification problem, *Theoretical Computer Science* **13**(2), 225–230.

Huet, G. P.: 1973, The undecidability of unification in third order logic, *Information and Control* **22**(3), 257–267.

Huet, G. P.: 1976, *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω*, PhD thesis, Université Paris, 7.

Kilpeläinen, P. and Mannila, H.: 1995, Ordered and unordered tree inclusion, *SIAM J. Comput.* **24**(2), 340–356.

Levy, J.: 1996, Linear second-order unification, *in* H. Ganzinger (ed.), *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA-96)*, Vol. 1103 of *LNCS*, Springer-Verlag, Berlin, pp. 332–346.

Loader, R.: 2003, Higher order β matching is undecidable, *Logic Journal of the IGPL* **11**(1), 51–68.

Padovani, V.: 1996, *Filtrage d'ordre supérieur*, PhD thesis, Université de Paris 7.

Salvati, S. and De Groote, P.: 2003, On the complexity of higher-order matching in the linear λ-calculus, *in* R. Nieuwenhuis (ed.), *International Conference on Rewriting Techniques and Applications - RTA'2003, Valencia, Spain*, Vol. 2706 of *Lecture notes in Computer Science*, pp. 234–245.

Schmidt-Schauß, M. and Stuber, J.: 2001, On the complexity of linear and stratified context matching problems, *Technical Report A01-R-411*, LORIA.

Chapter 4

Abrusci, M., Fouqueré, C. and Vauzeilles, J.: 1999, Tree-adjoining grammars in a fragment of the Lambek calculus, *Computational Linguistics* **25**(2), 209–236.

Barendregt, H. P.: 1984, *The lambda calculus, its syntax and semantics*, North-Holland. Revised edition.

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

de Groote, P.: 2001, Towards abstract categorial grammars, *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155.

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

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

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

Joshi, A. K. and Kulick, S.: 1997, Partial proof trees as building blocks for a categorial grammar, *Linguistics and Philosophy* **20**(6), 637–667.

Joshi, A. K. and Schabes, Y.: 1997, *Tree-adjoining grammars*, Vol. 3 of *G. Rozenberg and A. Salomaa, editors, Handbook of formal languages*, Springer, chapter 2.

Mönnich, U.: 1997, Adjunction as substitution, *in* G.-J. Kruijff, G. Morrill and R. Oehrle (eds), *Proceedings of Formal Grammar*, pp. 169–178.

Moortgat, M.: 1996, Categorial type logics, *in* J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, Elsevier Science Publishers, Amsterdam, pp. 93–177.

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

Oehrle, R. T.: 1994, Term-labeled categorial type systems, *Linguistic and Philosophy* **17**.

Pollard, C.: 1984, *Generalized Phrase Structure Grammars, Head Grammars, and Natural Language*, PhD thesis, Stanford University, CA.

Ranta, A.: 2004, Grammatical Framework, *Journal of Functional Programming* **14**(2), 145–189.

Seki, H., Matsumura, T., Fujii, M. and Kasami, T.: 1991, On multiple context-free grammars, *Theoretical Computer Science* **223**, 87–120.

Vijay-Shanker, K., Weir, D. J. and Joshi, A. K.: 1987, Characterizing structural descriptions produced by various grammatical formalisms, *Proceedings of the 25th ACL*, Stanford, CA, pp. 104–111.

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

Chapter 5

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

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

Bos, J.: 1995, Predicate logic unplugged, *Proceedings of the Tenth Amsterdam Colloquium*.

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

Danos, V. and Cosmo, R. D.: 1992, The linear logic primer, `http://www.pps.jussieu.fr/~dicosmo/CourseNotes/LinLog/`. An introductory course on Linear Logic.

de Groote, P.: 2001, Towards abstract categorial grammars, *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155.

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

de Groote, P. and Pogodalla, S.: 2003, m-linear context-free rewriting systems as abstract categorial grammars, *in* R. Oehrle and J. Rogers (eds), *MOL 8, proceedings of the eighth Mathematics of Language Conference*.

Frank, A. and van Genabith, J.: 2001, Glue tag: Linear logic based semantics construction for ltag - and what it teaches us about the relation between lfg and ltag, *in* M. Butt and T. H. King (eds), *Proceedings of the LFG '01 Conference*, Online Proceedings, CSLI Publications. `http://cslipublications.stanford.edu/LFG/6/lfg01.html`.

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

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

Joshi, A. K., Kallmeyer, L. and Romero, M.: 2003, Flexible composition in ltag: Quantifier scope and inverse linking, *in* H. Bunt, I. van der Sluis and R. Morante (eds), *Proceedings of the Fifth International Workshop on Computational Semantics IWCS-5*.

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

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

Pogodalla, S.: 2004a, Computing semantic representation: Towards ACG abstract terms as derivation trees, *Proceedings of the Seventh International Workshop on Tree Adjoining Grammar and Related Formalisms (TAG+7)*, pp. 64–71.

Pogodalla, S.: 2004b, Using and extending the ACG technology: Endowing categorial grammars with an underspecified semantic representation, *Proceedings of Categorial Grammars 2004, Montpellier*, pp. 197–209.

Portner, P. and Partee, B. H. (eds): 2002, *Formal Semantics: The Essential Readings*, Blackwell Publishers.

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

Chapter 6

de Groote, P. and Salvati, S.: 2004, Higher-order matching in the linear lambda-calculus with pairing, *in* A. T. Jerzy Marcinkowski (ed.), *18th International Workshop on Computer Science Logic - CSL'2004, Karpacz, Poland*, Vol. 3210 of *Lecture notes in Computer Science*, Springer, pp. 220–234.

Muskens, R.: 2001, Lambda Grammars and the Syntax-Semantics Interface, *in* R. van Rooy and M. Stokhof (eds), *Proceedings of the Thirteenth Amsterdam Colloquium*, Amsterdam, pp. 150–155.

Muskens, R.: 2003, Lambdas, Language, and Logic, *in* G.-J. Kruijff and R. Oehrle (eds), *Resource Sensitivity in Binding and Anaphora*, Studies in Linguistics and Philosophy, Kluwer, pp. 23–54.

Pogodalla, S.: 2004, Using and extending the ACG technology: Endowing categorial grammars with an underspecified semantic representation, *Proceedings of Categorial Grammars 2004, Montpellier*, pp. 197–209.

Pollard, C.: 2004, High-order categorical grammars, *Proceedings of Categorial Grammars 2004, Montpellier*, pp. 340–361.

Yoshinaka, R. and Kanazawa, M.: 2005, The complexity and generative capacity of lexicalized abstract categorial grammars, *in* P. Blache, E. Stabler, J. Busquets and R. Moot (eds), *Proceedings of Logical Aspects of Computational Linguistics: 5th International Conference, LACL 2005*, Vol. 3492 of *LNCS*, Springer-Verlag GmbH, pp. 330–348.