Master Sciences Cognitives et Applications

Spécialité Traitement Automatique du Langage

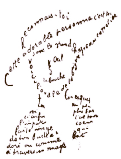
Master Thesis

Towards a Wide-Coverage Grammar: Graphical Abstract Categorial Grammars

June 25, 2013

Author:
Jiří Maršík

Supervisor:
Maxime Amblard



Abstract

We present work whose ultimate goal is the creation of a wide-coverage abstract categorial grammar (ACG) that could be used to automatically build discourse-level representations. In our work, we advance towards that goal by laying down the foundations necessary for building wide-coverage ACGs.

We first examine existing language resources, in particular the Frigram interaction grammar and its lexicon Frilex, and assess their utility to building a wide-coverage ACG. We then present our implementation of the ACG machinery which allows us to experiment with grammars lexicalized by Frilex. Finally, we consider the challenge of integrating the treatment of disparate linguistic constraints in a single ACG and propose a generalization of the formalism: graphical abstract categorial grammars. The report concludes with an exploration of some of the formal properties of graphical ACGs.

Acknowledgments

I would like to thank my friends in Nancy for making my year at Université de Lorraine a wonderful one.

I would like to thank my supervisor for his enlightening guidance and his dedication and attention to our work.

I would like to thank my parents for always supporting me in the things I have chosen to do.

I would like to thank Wes Anderson for *Moonrise Kingdom*.

Contents

Chapter 1

Preliminaries

1.1 Motivation

To develop applications capable of understanding natural language, we have to analyze the discourse as a complex structure. The structure of the discourse and the rhetorical relations between its constituent propositions have shown to have an important effect on anaphora and on the semantic content of the discourse [4].

Let us consider these examples from [4].

(1) a. Max fell.
    b. John helped him up.

(2) a. Max fell.
    b. John pushed him.

In (1), we intuitively recognize that the second proposition (1b) serves as a narrative continuation of (1a), whereas in (2), the proposition (2b) serves as an explanation to the event described in (2a). This distinction has interesting consequences as to what we can infer from these two discourse excerpts. In the case of (1), we can infer that the event described in (1b) occurred after the event described in (1a). In (2), we can conversely infer that the events happened in the opposite order. We feel that a complete understanding of the two examples above presupposes correctly inferring the temporal order of the events described and we would thus welcome a principled way to handle these distinctions.

(3) a. Max had a lovely evening last night.
    b. He had a great meal.
    c. He ate salmon.
    d. He devoured lots of cheese.
    e. He then won a dancing competition.

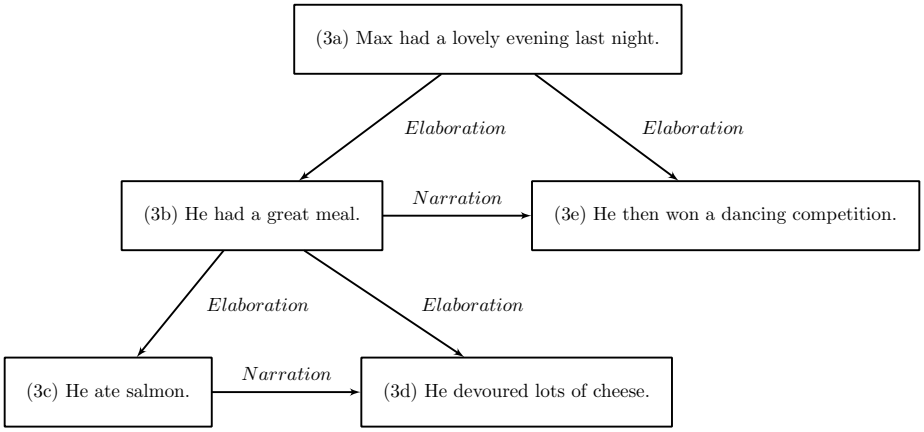


Figure 1.1: The SDRT structure of the discourse in 3.

The discourse in (3) and the representation of its structure on Figure 1.1 demonstrate another feature of discourse structures. First off, seeing the hierarchical structure of the discourse gives us important information about the granularity of the description employed in the individual propositions, information that could be useful for performing tasks such as text summarization.

Furthermore, the discourse structure has grammatical consequences. It is thanks to our knowledge of the discourse structure that we can predict that a proposition like "It was a beautiful pink." could not coherently follow our excerpt. SDRT, [4], the theory of discourse structure that we will adhere to, would not license a discourse structure in which the new proposition connects to (3c) as its *Elaboration*. As a consequence, it states that the pronoun "it" cannot have the salmon as its antecedent.

(4) a. A: Smith doesn't seem to have a girlfriend.
b. B: He's been paying lots of visits to New York lately.

In the example dialogue (4), based on the prosody of proposition (4b) the discourse structure would link the two propositions with an *Evidence* or *Counter-evidence* relation. Knowing this structure would allow us to either infer that B believes that Smith has a girlfriend in New York or that Smith doesn't have a girlfriend because he is too busy in New York.

Based on the findings in [4], we surmise that a clear picture of the discourse structure is essential to capturing the intended meaning of a discourse.

To support our approach, we will stand on the shoulders of many giants, the first of them being Richard Montague. As in his approach [17], we will assign functions and values as denotations of wordforms and use function application to compose them together to yield the denotations of phrases, propositions and discourses. To account for the discourse-level phenomena, we will defer to the theories of DRT [14] and SDRT [4].

The grammatical formalism of our choice for this task will be the abstract categorial grammars (ACGs) [6]. This framework lets use lambda calculus to express the syntax-semantics interface in a fashion similar to Montague's. Furthermore, elegant techniques for using ACGs to implement DRT and SDRT using continuations have been discovered (see [7], [3], [2] and [25]).

However, abstract categorial grammars are quite young and no significant grammar has been developed under this framework. To facilitate the creation of such a grammar, we will borrow heavily from the Frigram Interaction Grammar [22] and its linguistic resources. Namely, we will rely on the fact that Frigram is defined separately from its lexicon, Frilex, which we will use in its entirety. Frigram, the grammar itself, can also serve us as a guideline when designing our own grammar thanks to the close ties between the formalisms of interaction grammars and abstract categorial grammars [20].

Finally, our work will be motivated by the existence of a corpus annotated with the rhetorical relations of SDRT, the Annodis corpus [1], which would allow evaluation of the final grammar.

1.2 Outline

In the rest of this chapter, we proceed to introduce two grammatical formalisms that are of interest to our work, the formalism of abstract categorial grammars and the formalism of interaction grammars. We talk about a formal connection between the two and its value with respect to constructing our grammar.

In Chapter 2, we present our software implementation of the ACG machinery which will enable us to write and test grammars lexicalized by Frilex.

In Chapter 3, we take the first steps into designing a wide-coverage grammar by showing the treatment of several disparate linguistic constraints in the framework of ACGs and the challenges inherent in trying to combine them all in a single grammar.

Chapter 4 builds on the problems highlighted in Chapter 3 and introduces a generalization of abstract categorial grammars that is geared towards solving these problems.

We conclude our report with Chapter 5, in which we summarize our findings and point out potential directions for future work.

1.3 Abstract Categorial Grammars

We present the grammatical framework in which we will develop our system. Abstract categorial grammars are built upon two mathematical structures, *(higher-order) signatures* and *lexicons*.

1.3.1 Higher-Order Signatures

A **higher-order signature** is a set of elements that we call *constants*, each of which is associated with a type. Formally, it is defined as a triple $\Sigma = <A, C, \tau>$,

$$\emptyset; \Gamma_i \vdash_\Sigma c : \tau(c) \text{ (cons)}$$

$$(x : \alpha); \Gamma_i \vdash_\Sigma x : \alpha \text{ (l-var)}$$

$$\emptyset; (\Gamma_i, x : \alpha) \vdash_\Sigma x : \alpha \text{ (i-var)}$$

$$\frac{(\Gamma_l, x : \alpha); \Gamma_i \vdash_\Sigma t : \beta}{\Gamma_l; \Gamma_i \vdash_\Sigma \lambda^\circ x.t : \alpha \multimap \beta} \text{ (l-abs)}$$

$$\frac{\Gamma_l; (\Gamma_i, x : \alpha) \vdash_\Sigma t : \beta}{\Gamma_l; \Gamma_i \vdash_\Sigma \lambda x.t : \alpha \to \beta} \text{ (i-abs)}$$

$$\frac{\Gamma_l; \Gamma_i \vdash_\Sigma t : \alpha \multimap \beta \qquad \Delta_l; \Delta_i \vdash_\Sigma u : \alpha}{(\Gamma_l, \Delta_l); (\Gamma_i, \Delta_i) \vdash_\Sigma (t\ u) : \beta} \text{ (l-app)}$$

$$\frac{\Gamma_l; \Gamma_i \vdash_\Sigma t : \alpha \to \beta \qquad \emptyset; \Delta_i \vdash_\Sigma u : \alpha}{\Gamma_l; (\Gamma_i, \Delta_i) \vdash_\Sigma (t\ u) : \beta} \text{ (i-app)}$$

Figure 1.2: Type judgment schemas of the well-typed lambda terms $\Lambda(\Sigma)$ built on a signature $\Sigma = <A, C, \tau>$.

where:

- C is the (finite) set of constants
- A is a (finite) set of atomic types
- τ is the type-associating mapping from C to $\mathcal{T}(A)$, the set of types built over A

In our case, $\mathcal{T}(A)$ is the implicative fragment of linear and intuitionistic logic with A being the atomic propositions. This means that $\mathcal{T}(A)$ contains all the $a \in A$ and all the $\alpha \multimap \beta$ and $\alpha \to \beta$ for $\alpha, \beta \in \mathcal{T}(A)$.

A signature $\Sigma = <A, C, \tau>$, by itself, already lets us define an interesting set of structures, that is the set $\Lambda(\Sigma)$ of *well-typed lambda terms* built upon the signature Σ. The set of well-typed terms and their types are established through the judgment schemas in Figure 1.2.

The definition of a well-typed lambda term already gives us an interesting combinatorial structure. To make this structure even more useful, we often focus ourselves only on terms that have a specific *distinguished type*. Using this notion of a signature of typed constants and some distinguished type, we can describe languages of, e.g. tree-like (and by extension string-like), lambda terms.

1.3.2 Example Signature

Let us take a look at an example from [23]. We will start with a very simple signature of string expressions with the concatenation operator, $\Sigma_{String} = <A_{String}, C_{String}, \tau_{String}>$. Our signature will need only one atomic type, $A_{String} = \{\text{STRING}\}$. Our constants will be the wordforms of our example fragment (*every*, *some*, *man*, *woman*, *loves*), the empty string ϵ and the string concatenation operator, $+$. τ_{String} assigns the type STRING to all the wordforms and to ϵ, and the type STRING $\multimap$ STRING $\multimap$ STRING[1] to the concatenation operator $+$.

Now we can look at some well-typed terms from the string domain (i.e. elements of $\Lambda(\Sigma_{String})$). The term $t_1 = \mathit{some} \ + \ \mathit{woman}$[2] denotes the concatenation of *some* and *woman*, which is the phrase *some woman*; its type is STRING. A term such as $t_2 = \lambda^{\circ} x.x \ + \ \mathit{man}$ denotes a function that appends the string *man* to its argument; its type is STRING $\multimap$ STRING. If we restrict the set $\Lambda(\Sigma_{String})$ only to terms having the distinguished type STRING, we get the set of expressions which denote strings (a set that contains terms like t_1 but not t_2).

1.3.3 Lexicons

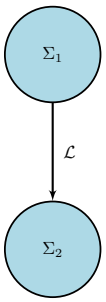
The idea of a signature is coupled with the one of **lexicon**, which is a mapping between two different signatures (mapping the constants of one into well-typed terms of the other). Formally speaking, a lexicon $\mathcal{L}$ from a signature $\Sigma_1 = <A_1, C_1, \tau_1>$ (which we call the abstract signature) to a signature $\Sigma_2 = <A_2, C_2, \tau_2>$ (which we call the object signature) is a pair $<F, G>$ such that:

- G is a mapping from C_1 to $\Lambda(\Sigma_2)$ assigning to every constant of the abstract signature a term in the object signature, which can be understood as its interpretation/implementation/realization.
- F is a mapping from A_1 to $\mathcal{T}(A_2)$ which links the abstract-level types with the object-level types that they are realized as.
- F and G are compatible, meaning that for any $c \in C_1$, we have $\vdash_{\Sigma_2} G(c) : \hat{F}(\tau_1(c))$ (we will be using $\hat{F}$ and $\hat{G}$ to refer to the homomorphic extensions of F and G to $\mathcal{T}(A_1)$ and $\Lambda(\Sigma_1)$ respectively). This property will be referred to as the *homomorphism property* of a lexicon.

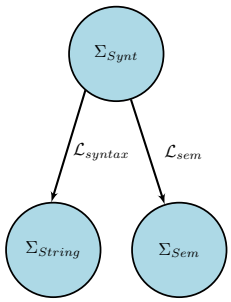
See Figure 1.3(a) for the kind of diagram we will be using to represent signatures and lexicons that will form our grammars.

[1] As a convention, whenever we omit parentheses in a type, we presume the $\multimap$ and $\rightarrow$ type constructors always bind from the right, meaning that $a \rightarrow b \rightarrow c$ should be read as $a \rightarrow (b \rightarrow c)$.

[2] Here we introduce another notational convention. The unadorned way of writing this term would be $t_1 = (+ \ \mathit{some}) \ \mathit{woman}$. First, we will allow ourselves to drop the parentheses, meaning that any string of expressions $f \ x \ y$ should be read as $(f \ x) \ y$. Second, we will admit a specific infix notation for some constants which denote functions of two arguments. This will then let us write $x \ + \ y$ instead of $+ \ x \ y$.



(a) A lexicon with its abstract and object signatures..



(b) Two lexicons sharing the same abstract signature.

Figure 1.3: Diagrams of abstract categorial grammars.

An abstract categorial grammar for us will then be just a collection of signatures with their distinguished types and lexicons connecting these signatures. A common pattern will have us using two object signatures for the surface forms of utterances (strings) and their logical forms (logical propositions) and an abstract signature which is connected to both of the object signatures via lexicons (as we can see on Figure 1.3(b)). Parsing is then just a matter of inverting the surface lexicon to get the abstract term and then applying to it the logical lexicon. Generation is symmetric, we simply invert the logical lexicon and apply the surface lexicon.

1.3.4 Example Lexicon

To illustrate the ideas of a lexicon and an abstract categorial grammar, we will expand our example from 1.3.2. We will consider another signature, Σ_{Synt}, which will describe the syntax of quantified noun phrases in the style of Montague [17]. The atomic types A_{Synt} will consist of the types NP, N and S. Our constants, $C_{Synt} = \{C_{\text{every}}, C_{\text{some}}, C_{\text{love}}, C_{\text{man}}, C_{\text{woman}}\}$, will have the following types, as predicted by τ_{Synt}:

$$\begin{aligned}
\tau_{Synt}(C_{\text{every}}) &= N \multimap ((NP \multimap S) \multimap S) \\
\tau_{Synt}(C_{\text{some}}) &= N \multimap ((NP \multimap S) \multimap S) \\
\tau_{Synt}(C_{\text{love}}) &= NP \multimap NP \multimap S \\
\tau_{Synt}(C_{\text{man}}) &= N \\
\tau_{Synt}(C_{\text{woman}}) &= N
\end{aligned}$$

Let us explore some terms from $\Lambda(\Sigma_{Synt})$. Let $t_3 = C_{\text{some}}\ C_{\text{woman}}$, t_3 has type $(NP \multimap S) \multimap S$, which is the type that serves to describe quantified noun phrases in our example. Terms of this type expect a verb phrase (or some other predicate) of type $NP \multimap S$, and can yield a sentence of type S.

Now we will see how this approach can handle transitive verbs. Let us consider the term $t_4 = \lambda^\circ x.\ (C_{\text{some}}\ C_{\text{woman}})\ (\lambda^\circ y.\ C_{\text{love}}\ x\ y)$, which has type $NP \multimap S$ and represents the verb phrase *loves some woman*. To build it, we first take the expression $C_{\text{love}}\ x\ y$ (x *loves* y, type S) and we abstract over y to get the predicate $\lambda^\circ y.\ C_{\text{love}}\ x\ y$ (*is loved by* x, type $NP \multimap S$). By applying the quantified noun phrase t_3 to this term, we get $(C_{\text{some}}\ C_{\text{woman}})\ (\lambda^\circ y.\ C_{\text{love}}\ x\ y)$ (x *loves some woman*, type S). Finally, by abstracting over x, we get the verb phrase t_4.

$t_5 = (C_{\text{every}}\ C_{\text{man}})\ (\lambda^\circ x.\ (C_{\text{some}}\ C_{\text{woman}})\ (\lambda^\circ y.\ C_{\text{love}}\ x\ y))$ is the result of applying the quantified noun phrase $C_{\text{every}}\ C_{\text{man}}$ (of type $(NP \multimap S) \multimap S$) to t_4. What we get is a term of type S which describes the sentence *every man loves some woman*. Note that as before, we can restrict the set $\Lambda(\Sigma_{Synt})$ to terms having the distinguished type S and we will arrive at the set of terms which describe sentences (this set will include t_5, but not t_3 or t_4).

In the example above, I have been presuming some kind of implied connection between the terms of $\Lambda(\Sigma_{Synt})$ and English phrases. Indeed, while we have defined a system of structures which describe the quantification and predicate structures of a microscopic fragment of English and a system for talking about strings in concatenation, we have not given any explicit link between the two. It is at this moment that we will introduce a lexicon to link these two levels of description.

Our lexicon $\mathcal{L}_{syntax}$ will map the constants of Σ_{Synt}, our abstract signature, into terms from $\Lambda(\Sigma_{String})$, our object signature. If we view terms of $\Lambda(\Sigma_{Synt})$ as abstract computations, the lexicon will instantiate this abstraction by providing an implementation for the constants of Σ_{Synt}. This way we can map an abstract computation representing a phrase into a more specific object computation which calculates the string representation.

So, how does our lexicon $\mathcal{L}_{syntax}$ look like? It will map all the atomic types of Σ_{Synt} into String.[3]

$$\begin{aligned}\mathcal{L}_{syntax}(N) &= \text{String}\\ \mathcal{L}_{syntax}(NP) &= \text{String}\\ \mathcal{L}_{syntax}(S) &= \text{String}\end{aligned}$$

We can construct the homomorphic extension of the above type mapping which can give us the object-level interpretations of complex abstract types, such as the type of predicates, $NP \multimap S$, which becomes a unary string function, String $\multimap$ String, or the type of quantified noun phrases, $(NP \multimap S) \multimap S$, which becomes a higher-order string function, (String $\multimap$ String) $\multimap$ String.

With the types out of the way, we can give the interpretation of the individual constants of Σ_{Synt}.

[3]Whenever we talk about some lexicon, we will use the lexicon itself, which is formally a pair of mappings $<F, G>$, to mean either F, G or their homomorphic extensions $\hat{F}$ and $\hat{G}$ depending on whether the argument is an atomic type, a constant, a complex type or a term respectively.

$$\begin{aligned}
\mathcal{L}_{syntax}(C_{\text{every}}) &= \lambda^{\circ} xR.\ R\ (every\ +\ x) \\
\mathcal{L}_{syntax}(C_{\text{some}}) &= \lambda^{\circ} xR.\ R\ (some\ +\ x) \\
\mathcal{L}_{syntax}(C_{\text{love}}) &= \lambda^{\circ} xy.\ x\ +\ loves\ +\ y \\
\mathcal{L}_{syntax}(C_{\text{man}}) &= man \\
\mathcal{L}_{syntax}(C_{\text{woman}}) &= woman
\end{aligned}$$

We can now go back to our examples from $\Lambda(\Sigma_{Synt})$ and look at what they look like after applying the lexicon to them.

- $t_3 = C_{\text{some}}\ C_{\text{woman}}$
  Mapping this term using the lexicon and β-reducing gives us $\mathcal{L}_{syntax}(t_3) =_{\beta} \lambda^{\circ} R.\ R\ (some\ +\ woman)$, which is a type-raised version of the string *some woman*.

- $t_4 = \lambda^{\circ} x.\ (C_{\text{some}}\ C_{\text{woman}})\ (\lambda^{\circ} y.\ C_{\text{love}}\ x\ y)$
  Our verb phrase ends up being mapped onto a function which appends the phrase *loves some woman* to its argument, $\mathcal{L}_{syntax}(t_4) =_{\beta} \lambda^{\circ} x.\ x\ +\ loves\ +\ some\ +\ woman$.

- $t_5 = (C_{\text{every}}\ C_{\text{man}})\ (\lambda^{\circ} x.\ (C_{\text{some}}\ C_{\text{woman}})\ (\lambda^{\circ} y.\ C_{\text{love}}\ x\ y))$
  Finally, the sentence (type S) is mapped into a simple String, $\mathcal{L}_{syntax}(t_5) =_{\beta} every\ +\ man\ +\ loves\ +\ some\ +\ woman$.

Now we have enough machinery in play to define the set of strings which form our fragment of English. We will consider the *abstract language* generated by our signature Σ_{Synt} and the distinguished type S, that is, elements of $\Lambda(\Sigma_{Synt})$ having type S. Then we can define the *object language*, which is the image of the abstract language when transformed using the lexicon. The object language contains strings, i.e. terms from $\Lambda(\Sigma_{String})$ having the type String. However, the object language does not contain all such terms, but only those object terms, for which there exists a term in the abstract language such that its image given by the lexicon yields the same object term, i.e. it contains only terms which denote strings that spell out sentences of our fragment of English.

1.3.5 Example Semantic Lexicon

One perk of working with abstract categorial grammars is that one abstract term can be interpreted in multiple ways, using different lexicons. In the previous chapter, we considered the abstract terms of the Σ_{Synt} signature and interpreted them as computations on strings. Now we will take the same abstract terms, but try to interpret them as computations on entities and truth values. A diagram of this setup can be seen on the Figure 1.3(b).

We will introduce a new signature, Σ_{Sem}, for our semantic computations. The atomic types A_{Sem} will consist of a type for entities, e, and a type for truth

values, t. The constants C_{Sem} will contain the atomic predicates from our fragment, **man**, **woman** and **love**, logical connectives, $\Rightarrow$ and $\wedge$, and quantifiers, $\forall$ and $\exists$. Their types as given by τ_{Sem} are as follows:[4]

$$\begin{aligned}
\tau_{Sem}(\mathbf{man}) &= e \multimap t \\
\tau_{Sem}(\mathbf{woman}) &= e \multimap t \\
\tau_{Sem}(\mathbf{love}) &= e \multimap e \multimap t \\
\tau_{Sem}(\Rightarrow) &= t \multimap t \multimap t \\
\tau_{Sem}(\wedge) &= t \multimap t \multimap t \\
\tau_{Sem}(\forall) &= (e \to t) \multimap t \\
\tau_{Sem}(\exists) &= (e \to t) \multimap t
\end{aligned}$$

Now to see how this signature connects to our abstract signature Σ_{Synt}, we will give a lexicon from the latter to the former. First, we give the type mappings. $\mathcal{L}_{sem}(N) = e \multimap t$ meaning that nouns will yield boolean functions of one argument (unary *predicates* in functional programming terminology). $\mathcal{L}_{sem}(NP) = e$, simple noun phrases will be mapped into terms which yield some entity. Finally, $\mathcal{L}_{sem}(S) = t$, sentences will be assigned terms which compute their truthiness. Let us see how this plan is achieved in the $\mathcal{L}_{sem}$ lexicon.[5]

$$\begin{aligned}
\mathcal{L}_{sem}(C_{\text{every}}) &= \lambda^\circ PQ.\ \forall x.\ ((P\ x) \Rightarrow (Q\ x)) \\
\mathcal{L}_{sem}(C_{\text{some}}) &= \lambda^\circ PQ.\ \exists x.\ ((P\ x) \wedge (Q\ x)) \\
\mathcal{L}_{sem}(C_{\text{love}}) &= \lambda^\circ so.\ (\mathbf{love}\ s\ o) \\
\mathcal{L}_{sem}(C_{\text{man}}) &= \lambda^\circ x.\ (\mathbf{man}\ x) \\
\mathcal{L}_{sem}(C_{\text{woman}}) &= \lambda^\circ x.\ (\mathbf{woman}\ x)
\end{aligned}$$

Let us now consider the example terms of the Σ_{Synt} signature from 1.3.4. $t_3 = C_{\text{some}}\ C_{\text{woman}}$ will be interpreted as $\mathcal{L}_{sem}(t_3) =_\beta \lambda^\circ Q.\ \exists x.\ ((\mathbf{woman}\ x) \wedge (Q\ x))$ having type $(e \multimap t) \multimap t$. It is a function that expects some predicate and tests whether that predicate holds for at least some woman (some entity for which the **woman** predicate holds).

When we consider $t_4 = \lambda^\circ x.\ (C_{\text{some}}\ C_{\text{woman}})\ (\lambda^\circ y.\ C_{\text{love}}\ x\ y)$, we get $\mathcal{L}_{sem}(t_4) =_\beta \lambda^\circ x.\ \exists y.\ ((\mathbf{woman}\ y) \wedge (\mathbf{love}\ x\ y))$ of type $e \multimap t$. What we have here is a predicate function testing whether the argument entity loves some woman.

Finally, in $t_5 = (C_{\text{every}}\ C_{\text{man}})\ (\lambda^\circ x.\ (C_{\text{some}}\ C_{\text{woman}})\ (\lambda^\circ y.\ C_{\text{love}}\ x\ y))$, we have $\mathcal{L}_{sem}(t_5) =_\beta \forall x.\ ((\mathbf{man}\ x) \Rightarrow (\exists y.\ ((\mathbf{woman}\ y) \wedge (\mathbf{love}\ x\ y))))$ of type t. This computation yields a truth value, which tells us whether the proposition that

[4]The non-linear implication ($\to$) in the types of $\forall$ and $\exists$, $(e \to t) \multimap t$, means that the variable "introduced" by the quantifier can be used multiple times in the predicate, which has type $e \to t$. We will rely on this property later when defining the meanings of determiners.

[5]We will be using $\forall x.\ M$ as a shortcut for $\forall\ (\lambda x.\ M)$ and $\exists x.\ M$ for $\exists\ (\lambda x.\ M)$.

every man loves some woman is true given some universe that the quantifiers range over and some values of the **man**, **woman** and **love** predicates. The term itself, in its β-normal form, can serve as our semantic representation for the proposition *every man loves some woman*.

We will finish our exposition of abstract categorial grammars by highlighting a prominent feature of the example grammar we have just described and that is its treatment of scope ambiguity. You might have noticed that $t_5 = (C_{\text{every}}\ C_{\text{man}})\ (\lambda^\circ x.\ (C_{\text{some}}\ C_{\text{woman}})\ (\lambda^\circ y.\ C_{\text{love}}\ x\ y))$ is not the only term in $\Lambda(\Sigma_{Synt})$ for which $\mathcal{L}_{syntax}(t_5) =_\beta every + man + loves + some + woman$. We could just as well take $t_6 = (C_{\text{some}}\ C_{\text{woman}})\ (\lambda^\circ y.\ (C_{\text{every}}\ C_{\text{man}})\ (\lambda^\circ x.\ C_{\text{love}}\ x\ y))$ which is not β-equivalent to t_5. When we apply our syntactic lexicon to t_6, we see that we end up with an expression denoting the same string, $\mathcal{L}_{syntax}(t_6) =_\beta every\ +\ man\ +\ loves\ +\ some\ +\ woman$.

However, when we consider a different interpretation for the abstract constants in Σ_{Synt}, for example our semantic lexicon $\mathcal{L}_{sem}$, we can see the differences rise to surface.

$$\mathcal{L}_{sem}(t_5) =_\beta \forall x.\ ((\textbf{man}\ x) \Rightarrow (\exists y.\ ((\textbf{woman}\ y) \wedge (\textbf{love}\ x\ y))))$$
$$\mathcal{L}_{sem}(t_6) =_\beta \exists y.\ ((\textbf{woman}\ y) \wedge (\forall x.\ ((\textbf{man}\ x) \Rightarrow (\textbf{love}\ x\ y))))$$

When we try to parse some sentence using abstract categorial grammars, we are trying to find the abstract terms which upon transformation by the lexicon and β-reduction yield the sentence encoded in some object term. This is basically trying to invert the lexicon function, modulo β-equivalence. It is therefore not surprising then that we can find multiple abstract terms which all map to the same string term but which can be mapped to distinct terms using the semantic lexicon. It is this mechanism that enables abstract categorial grammars to handle ambiguity. When we reverse the scenario and go from semantic representations to strings, the same situation can occur (more than one abstract term having the same semantics but different surface strings) and this is what enables paraphrases.

1.4 Interaction Grammars

We will briefly discuss one more grammar formalism and that is the formalism of interaction grammars [13]. This formalism is of particular interest to us since it is based on the same logical basis as ACGs and there is already a detailed wide-coverage grammar written in it.

Interaction grammars are a grammatical formalism centered around the concept of *polarity*. An interaction grammar is a set of (under-specified) tree descriptions which define a set of trees that can be constructed by superposing some of these tree descriptions while respecting the polarities described in the nodes of the individual tree descriptions.

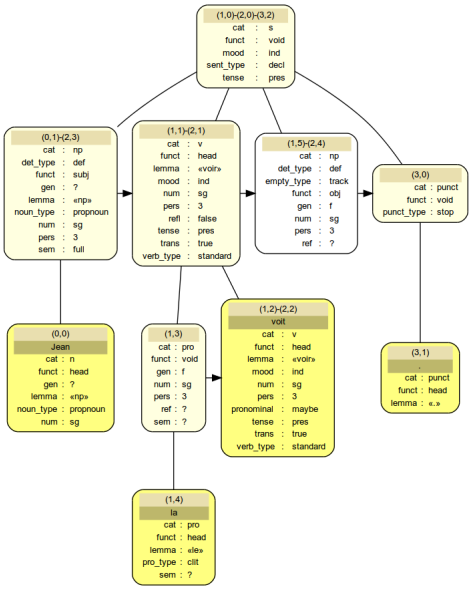


Figure 1.4: The syntactic tree of sentence (5). The numbers at the top of the nodes are labels of nodes from the elementary polarized tree descriptions that generated the syntactic tree (see Figure 1.5).

1.4.1 The Mechanisms of Interaction Grammars

We will illustrate the objects and mechanisms of interaction grammars on a simple example sentence. Since the largest and most-developed interaction grammar is the Frigram grammar of French, we will be using French example sentences and their analyses according to Frigram. Frigram will be discussed in more detail in subsection 1.4.2.

(5) Jean la voit.

Figure 1.4 shows the syntactic tree assigned by Frigram to sentence (5). It is a rooted ordered tree with the topmost node being the root. Solid lines indicate immediate dominance with the higher node being the parent and the lower node being the child. The ordering of children is given by the arrows, which signify the immediate precedence relation between sister nodes.

The nodes in the tree are of three different kinds with respect to the phonological form of their subtree. We recognize the *anchor nodes*, which are displayed in vivid yellow and which are leaf nodes containing some non-empty string as their phonological content (written in a gray rectangle at the top of the node). Then we have the *empty* nodes, which are drawn in white and whose phonological form is empty. Finally, there are the pale yellow *non-empty* nodes, internal nodes whose phonological content (the yield of the subtree) is not empty.

The nodes of the syntactic tree are also decorated with features.

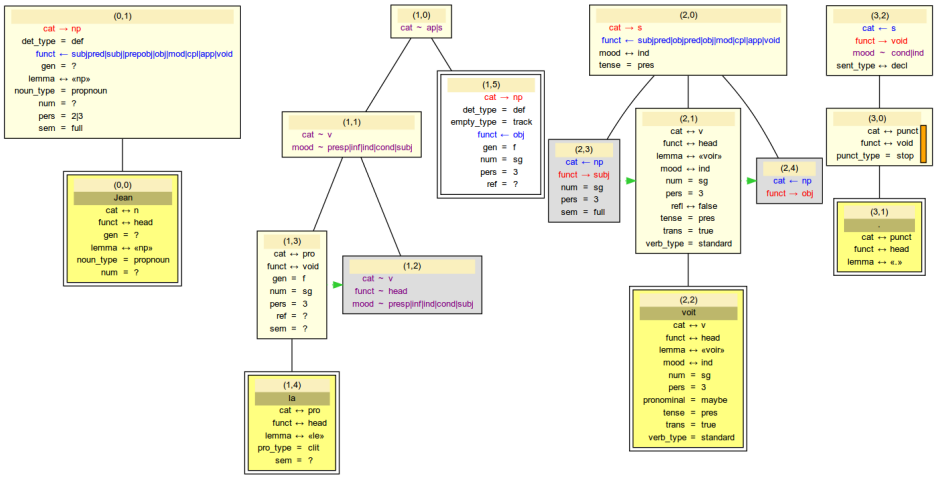


Figure 1.5: The elementary polarized tree descriptions used to generate the sentence (5).

Interaction grammars are a formalism enabling us to define sets of the structures described above, the *syntactic trees*. Similar to tree adjoining grammars, interaction grammars are formulated in terms of a set of elementary structures which can combine to produce the final output structures. However, unlike in tree adjoining grammars, the output structure is not constructed from the elementary structures via some set of algebraic operations (substitution and adjunction in case of TAGs). Instead, *polarized tree descriptions* (PTDs), the elementary structures of interaction grammars, impose constraints on the final structure and a structure is said to be generated by some PTDs if it is a minimal structure satisfying those constraints (we say it is a *minimal model*). This distinction separates TAGs as a formalism in the generative-enumerative framework of syntax from IGs as a formalism in the model-theoretic framework.

In Figure 1.5, we can see the elementary polarized tree descriptions (EPTDs) that generated the parse tree in Figure 1.4.

As we said before, a PTD is a set of constraints on some syntactic tree. Let us expound on what constraints the structures of Figure 1.5 impose.

A node in a PTD can be read as a statement that there must exist a node in the final syntactic tree that has compatible values for all the features and that carries the same phonological string, if any. Such a node of the syntactic tree is then called the *interpretation* of the PTD node (in Figure 1.4, every node of the syntactic tree bears a list of the PTD nodes that it interprets, so you can see exactly how the interpretation function works in our example).

A solid line between two nodes in a PTD means that the interpretations of these two nodes must be in a parent-child relationship (*immediate dominance*). A dashed green arrow tells us that the interpretations have to be sister nodes with the former preceding the latter in the ordered tree (*precedence*). Note that

not all sister nodes in a PTD have to be linked with this precedence relation, the tree can be under-specified. See for example the nodes (1,1) and (1,5) in the EPTD of the clitic pronoun *la*.

The formalism also allows us to specify *immediate precedence* and (large) *dominance*. The latter is useful for modelling unbounded dependencies such as those between a relative pronoun and its trace in some embedded clause of the relative clause, but it is not used in our present example.

Finally, one kind of constraint that is used in our example is the orange rectangle in node (3,0), which requires that the interpretation of (3,0) is the rightmost daughter amongst its siblings.

Now that we have covered the structural constraints imposed by the tree descriptions, we will turn our attention to the defining characteristic of interaction grammars, the polarities. As you have noticed on Figure 1.5, some of the features in our PTDs are annotated with special symbols and colors. These denote the different polarities and are used by the formalism for two distinct purposes: the positive ($\rightarrow$) and negative ($\leftarrow$) polarities are used to model the resource sensitivity of languages, while the virtual polarities ($\sim$) are used for pattern matching against the context.

The way the polarities are handled is that every model (output syntactic tree) is required to have only saturated polarities on its features (i.e. no positive, negative or virtual polarities). Whenever more than one node of the PTD is interpreted by the same node of the syntactic tree, the polarities of each feature are combined. The combination mechanism allows us to combine a positively polarized feature with a negatively polarized one to yield a saturated one. Virtual polarities can only be eliminated by combining them with a saturated polarity (they are analogous to the $=_c$ constraints of LFG [16]).

If we look at the PTDs of Figure 1.5, we can see this polarity mechanism in action. The EPTD of *Jean* has a positive *cat* feature in its root node saying that it provides one *np*, and a negative $funct$ feature saying that it expects some function. In the EPTD of *voit*, we have two nodes for the subject and the object, both of them expecting an *np* and providing them the *subj* and *obj* functions, respectively. The root of the EPTD then provides a complete sentence of category *s* and expects some function that this sentence will play. The full stop EPTD finalizes the sentence by accepting a node with category *s* and giving it a *void* function.

The clitic pronoun *la* participates in the positive/negative resource management system as well, since using the clitic fills up the object slot in the valency of a verb. The EPTD of *la* also uses virtual features heavily to select the right place where to hang the pronoun in the resulting tree.

Now that we understand the constraints imposed by PTDs, we can start to see that the syntactic tree in Figure 1.4 is truly a model of the PTDs given in Figure 1.5 (furthermore, it is the only minimal model). To illustrate more clearly how the polarities and constraints of the individual PTDs end up generating the syntactic tree of Figure 1.4, we can look at the result of superposing the EPTDs of *la* and *voit* by merging the nodes (1,2) and (1,5) with (2,2) and (2,4) respectively on Figure 1.6.

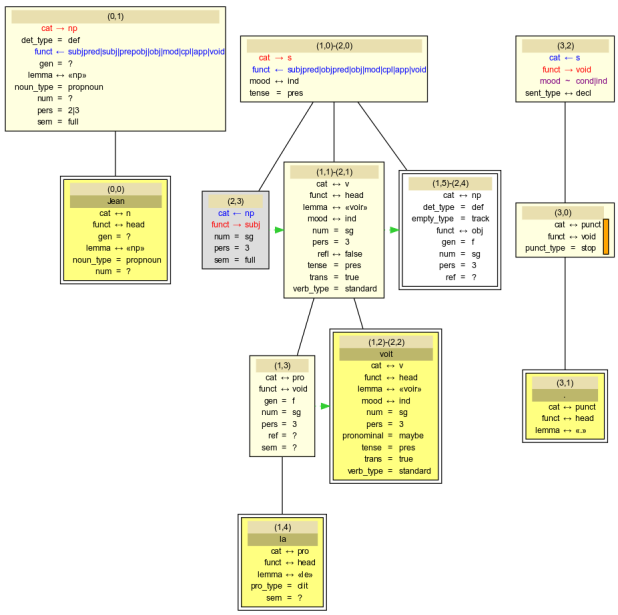


Figure 1.6: The result of merging the EPTDs of *la* and *voit* by merging the nodes (1,2) and (1,5) with (2,2) and (2,4), respectively.

1.4.2 Frigram, an Interaction Grammar

The reason we are interested in interaction grammars is because of the existence of Frigram, a large scale grammar of French which is lexicalized by Frilex, an independent lexical resource that we will be building our grammar on as well. Furthermore, as we will see in 1.4.3, interaction grammars are closely linked to abstract categorial grammars, which makes Frigram a suitable grammar to use as a guide when developing our grammar.

In this subsection, we will introduce the metagrammatical structure of Frigram and talk about how it uses the formalism of interaction grammars to solve several tricky linguistic phenomena.

Frigram as a Metagrammar

Frigram is a wide-coverage interaction grammar of French. As we mentioned before, an interaction grammar is a set of elementary polarized tree descriptions. In a lexicalized interaction grammar, each of these EPTDs is connected to a specific wordform whose particular use it describes. Given the amount of wordforms that a usable grammar of French would need to cover, the number of EPTDs in our grammar could easily reach hundreds of thousands, if not millions.

The first step in fighting this explosion is to factor out irrelevant differences between similar wordforms. For this purpose, Frilex was created. Frilex is

a morphosyntactic lexicon of French compiled from various other preexisting lexicons of French. It is in effect a large relation linking the wordforms of French to *hypertags*, feature structures describing the morphological properties of the wordforms and their syntactic valencies.

With Frilex in place, Frigram can be defined as a set of unanchored EPTDs which are paired with feature structures that delimit the subset of the Frilex items to which the EPTDs apply. This kind of simple "metagrammar" already saves us a lot of effort, the number of unanchored EPTDs defined by Frigram is somewhere around 4000[6].

However, defining some 4000 EPTDs manually still seems like a very tedious and error prone process that is likely to lead to a grammar that is hard to maintain. Many of the EPTDs have to repeatedly describe common phenomena such as subject-predicate agreement or predicate-argument saturation. Similarly as in other software endeavors, it would be preferable to define these common patterns in some reusable module and compose EPTDs from these building blocks. This is where XMG steps in [11].

XMG is a metagrammar compiler adding yet another level of indirection between what the grammar author writes and what ends up in the bottom-level (interaction) grammar. XMG provides a language that lets the grammar author define the lexical items of his grammar and to combine these definitions to yield new and more elaborate lexical items.

In the case of Frigram, the lexical items (termed *classes* in XMG) take the shape of PTDs coupled with feature structures which circumscribe the interface to Frilex. Both tree descriptions and feature structures have very natural ways of composing together, by conjunction and unification, respectively. This, alongside with disjunction, are the chief tools that the grammar authors can use (and in the case of Frigram, have used) to succinctly define their grammar.

With the added capability of composing classes, the definition of Frigram reduces to about 400 class definitions. Of these classes, 160 are terminal, meaning that they actually describe lexical items that are to be included in the final grammar. Thanks to the disjunction composition operator of XMG, these 160 classes end up generating the 4000 EPTDs.

Linguistic Description in Frigram

We will briefly discuss how interaction grammars, and Frigram in particular, handle a tricky linguistic phenomenon in an interesting way. In French, negation is signalled by the particle *ne*, which must be accompanied by one of several designated determiners, pronouns or adverbs. The particle *ne* assumes a position right before the inflected verb, but its partner, e.g. the determiner *aucun* can appear in a rich variety of positions:

(6) Aucun tatou ne court.

(7) Jean n'aime l'odeur d'aucun tatou.

[6] `http://wikilligramme.loria.fr/doku.php?id=frigram:frigram`

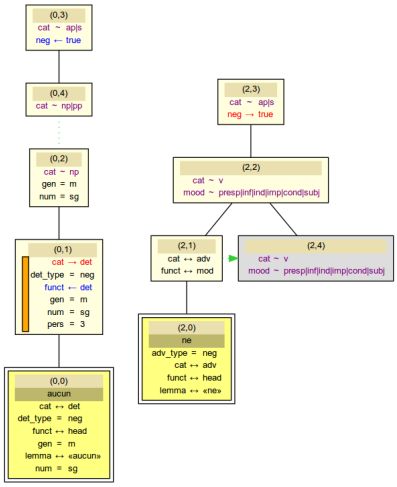


Figure 1.7: EPTDs of *aucun* and *ne* demonstrating the way polarities are used to model French negation.

(8) * Le tatou qu'aucun loup chasse ne court.

(9) Le tatou qu'aucun loup ne chasse court.

Sentences (6) and (7) demonstrate that the *aucun* determiner can be used both in the subject and object positions, as a determiner of either one of the arguments directly or of a noun phrase which complements one of them. However, it is not admissible to place the determiner in an embedded clause as it is in (8). If the determiner *aucun* is used in an embedded clause, it is the embedded clause itself that must be negated, as in sentence (9).

We can see the EPTDs of both *aucun* and *ne* on Figure 1.7. This example demonstrates two points of interest to us.

First, it uses a new polarized feature, *neg*, to model the fact that the two wordforms which enable negation must always co-occur. The particle *ne* provides a positive *neg* feature to the clause in which it occurs and this positive polarity must be saturated by a negative polarity contributed by one of the possible partner words, such as *aucun*. This shows that the usefulness of positive and negative polarities extends beyond the $cat/funct$ pair that we have already seen and which is used to model predicate/argument structure in a similar way as was already done in tree adjoining grammars or categorial grammars.

Second, the determiner *aucun* can occur deep, e.g. inside a complement of one of the arguments, but it cannot occur everywhere, as is demonstrated by (8), where we try to put the determiner inside another embedded clause. There is a way to express this kind of constraint in interaction grammars that we have not revealed yet. Whenever we use a (large) dominance (a top-down green dashed line), we usually want the range of nodes that this dominance relation can cross to be somehow restricted. In interaction grammars, we can associate a feature

structure with a dominance constraint which will force all of the nodes that span the distance between the two nodes linked by the dominance constraint to unify with that feature structure.

In the notation used in our illustrations (which were generated by the Leopar parser[7]), this is conveyed by putting all the restricting features as virtual features in a new node and splicing the new node in between the two nodes linked by the dominance constraint before (see the node (0,4) in Figure 1.7, which merely serves to ensure that all the nodes on the path from (0,3) to (0,2) have either *np* or *pp* as the value of their *cat* feature). When using this notation, the (large) dominance constraint should then be read as not only stating that the top node must be an ancestor to the bottom node, but also that any intermediate nodes between the ancestor and successor must match the ancestor's features.

1.4.3 The Link with Abstract Categorial Grammars

Both abstract categorial grammars and interaction grammars, having come out of the same research team, are both closely related to linear logic. Not only are these two formalisms based on the same logical framework, there is a surprisingly direct connection between the principles guiding the syntactic composition of constants having intuitionistic implicative linear types in abstract categorial grammars and of polarized tree descriptions in interaction grammars.

Perrier's Theorem

Perrier [20] proved an interesting result which bridges these two worlds. His theorem states that every IILL (intuitionistic implicative linear logic) sequent $F_1, ..., F_n \vdash G$ is provable if and only if the syntactic description $D((F_1 \multimap ... \multimap F_n \multimap G)^+)$ is valid. The second premise deserves some elaboration.

What we mean by a *syntactic description* is something similar to a polarized tree description, albeit more minimalistic. Syntactic descriptions only talk about immediate dominance and (large) dominance, there are no precedence constraints on sister nodes and (large) dominance cannot be restricted by a feature structure. Nodes themselves are no longer even represented by feature structures, but by simple atomic categories paired with polarities. Every node in a syntactic description is thus associated with a single atomic category and a polarity which is either positive or negative.

A syntactic description that is *valid* is one for which there exists a model. A model of a syntactic description is defined in the same manner as for polarized tree descriptions, but instead of unifying feature structures, we simply demand that all nodes of the description which map to a node in the model must have the same category and the number of positively polarized nodes must equal the number of negatively polarized nodes.

Finally, we explain the term $D((F_1 \multimap ... \multimap F_n \multimap G)^+)$. The + operation recursively assigns positive and negative polarities to all formula occurrences

[7] `http://leopar.loria.fr/`

within $F_1 \multimap ... \multimap F_n \multimap G$. These polarities then drive the recursive definition of D, which maps this polarized formula into a syntactic description. Since we will not cover the theorem deeply in our treatise, we will settle for knowing that the composition $D \circ +$ of the positive polarization operation $+$ and the syntactic description producing operation D maps an IILL formula into a syntactic description such that the above-stated theorem holds. Furthermore, this mapping can be easily constructed since Perrier gives straightforward ways of computing the results of both operations.

Perrier's theorem establishes a striking symmetry between the two formalisms. Let us consider the case of parsing a sentence using a lexicalized grammar in both formalisms. First, the sentence is tokenized into wordforms and for each wordform, a lexical item is selected, be it a typed constant in some abstract signature for ACGs or an EPTD for IGs.

In ACGs, we will try to take the typed constants and use them to produce a term having some distinguished type S. Thanks to the Curry-Howard correspondence, constructing a term having a given type is nothing more than proving that term's type as a formula in the logic of our type system. What this means is that our parsing problem boils down to proving the IILL sequent $\tau_1, ..., \tau_n \vdash S$ (we omit the inclusion of intuitionistic non-linear implications since they are not relevant for parsing sentences). The sentence thus becomes parsable by our grammar under the given lexical selection if and only if this sequent is provable and furthermore, the proof of the sequent is our desired syntactic structure in disguise.

Now in IGs, we will conjoin all the EPTDs into a single complex polarized tree description. Parsing is then just a matter of finding a model for this description. The sentence is therefore parsable by our grammar under the given lexical selection if and only if this description is valid and furthermore, the model which proves the description valid is our desired syntactic structure.

This symmetry turns Perrier's theorem into a strong claim that lets us transform the types of a signature into elementary syntactic descriptions such that some selection of descriptions will generate some tree if and only if the selection of the corresponding typed constants generated some properly typed term. Inverting this transformation would then let us take the elementary syntactic descriptions of an IG like Frigram and turn them into types for an abstract signature of our ACG.

Translating Tree Descriptions to Types

However, just inverting $D \circ +$ mapping to obtain types from tree descriptions is not sufficient. $D \circ +$, the transformation in Perrier's theorem, is far from a bijection.

Firstly, it is not injective and several types end up being represented by the same syntactic description. What this means is that the transformation ignores the insignificant differences between formulas such as $a \multimap (b \multimap c)$ and $b \multimap (a \multimap c)$. This would not pose a serious issue though, as when we would try to invert $D \circ +$, we could simply choose one of the possible formulas which

fit the syntactic description as our canonical representation since the differences between the types would be inconsequential.

Secondly, a bigger problem is posed by the fact that $D \circ +$ is not surjective. Perrier [21] characterized the set of syntactic descriptions that can be obtained as images of IILL sequents using our transformation $D \circ +$, the set of *IILL tree descriptions*, which is a proper subset of the set of all syntactic descriptions. Specifically, in an IILL tree description, all immediate dominance constraints go from negative nodes to positive nodes, all (large) dominance constraints lead from positive nodes to negative nodes and the root has a positive polarity.

While this property may not hold for the PTDs we find in Frigram, it still gives us a general plan for recovering IILL types. The strategy suggested by the description/formula connection given by the definition of $D \circ +$ yields a mapping F from the under-specified trees to IILL formulas.

$$F(N) = F(N_1) \multimap ... \multimap F(N_k) \multimap cat(N)$$

where $N_1, ..., N_k$ are the children of N and $cat(N)$ is the category of N. If N is a leaf, then $F(N)$ is simply $cat(N)$. In this definition, we treat the tree descriptions as trees with the union of the immediate dominance and (large) dominance constraints serving as the edges of the tree.

This strategy corresponds to our intuitions about how tree descriptions should be expressed using IILL types. We will consider two very simple cases to demonstrate it.

First, picture a simple tree description with a positive root A and two (necessarily) negative children B and C. Our strategy would give this tree the type $B \multimap C \multimap A$ (or $C \multimap B \multimap A$), which correctly expresses the fact that this tree needs to consume some node of category B and another one of category C and is able to provide a node of category A. This tree description can also be easily translated to TAGs by turning the negative leaf nodes into substitution nodes. The conventional translation of TAGs into categorial grammars then confirms the type proposed by our strategy.

The second example will demonstrate the translation of a Frigram EPTD. Before we do that, however, we will first discuss some of the ways the actual PTDs of interaction grammars and Frigram differ from the IILL tree descriptions we just talked about.

IG Features Outside the Scope of Perrier's Theorem

The PTDs of IGs include precedence constraints. We can see them as serving two purposes, to both provide an ordering of the constituents and to further constrain syntactic composition. In our translation to an ACG, we could imagine handling the first role, that of providing word order, by defining a lexicon from our abstract signature. Each constant would combine its arguments in the order in which their corresponding subtrees appeared in the ordered PTD. However, the second role of precedence constraints, to establish further restrictions on syntactic composition, would be lost.

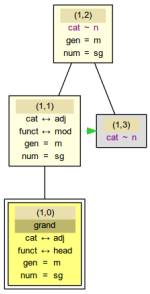


Figure 1.8: An EPTD for the adjective *grand* in Frigram.

Another large difference is the use of multiple independent polarized features per node. A possible solution here is to salvage the most salient polarized feature, *cat* (possibly with its complementary feature, $funct$), and ignore the others. The results and intuitions we have about interpreting tree descriptions as types do not offer an easy way of modeling polarities across different dimensions, features, at the same time. However, dropping the other polarized features would cost us the treatments of some phenomena, such as the paired grammatical words for negation in French, which were handled using the *neg* feature.

Next up are the virtual features. They serve multiple purposes as well. Most often they are used to select nodes in the context and to link them to the nodes contributed by the EPTD.

A common pattern using virtual features is Frigram's handling of modifiers. The way we would model a modifier in categorial grammars would be to have a function of type $X \multimap X$ where X is the type of the constituent being modified. In TAGs, we would have an adjunction tree with a root and a foot node of category X and which pins the modifier on the X phrase. This approach would work in interaction grammars as well but the formalism offers other solutions too. The designers of Frigram have opted for an alternate treatment in which modifying a constituent does not increase the depth of the parse tree. Modifiers are adjoined as sister nodes of the constituents they modify and it is through virtual features that IGs can ensure that a modifier is inserted only in the proper context.

See Figure 1.8 for an example of a modifier adjective in Frigram. Nodes (1,3) and (1,2) select the noun to be modified and its projection, respectively. Singling out this pair of nodes lets us hang the adjective as a child of the projection and state that it must precede the noun in the word order. We can see that this does not increase the depth of the noun in the parse tree, since the noun is only referred to using virtual features in this EPTD, meaning that some other EPTD must provide the nodes that will saturate these virtual features.

Understanding this pattern can give us a useful tool for reading EPTDs. Consider the EPTD of *la* we saw in Figure 1.5. The EPTD can now be seen as applying this pattern twice at the same time to both cliticize the pronoun to

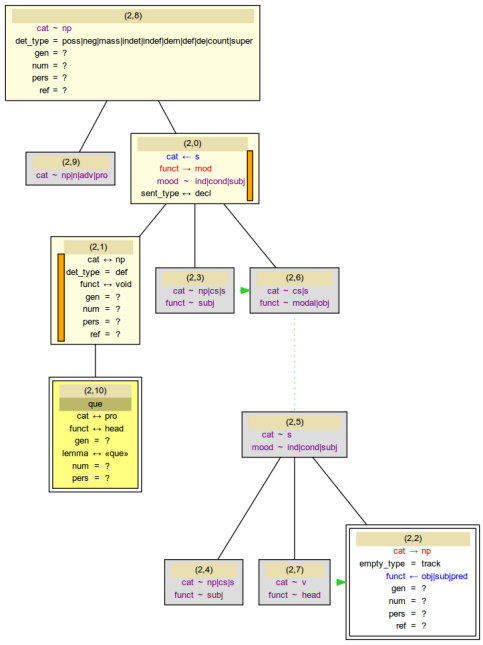


Figure 1.9: An EPTD of the relative pronoun *que*.

the verbal kernel and to fill the object valency slot of the verb.

Another use of virtual features in IGs is to restrict the range of (large) dominance relations. We have seen this before in the EPTD of the determiner *aucun* (Figure 1.7), which had to reach up to the nearest containing clause and hang a negative *neg* feature there. It is also present in the EPTD of the relative pronoun *que* (Figure 1.9) which will bring us back to the translation strategy proposed above.

Once again, at the root of the EPTD we see the modifier pattern in action. If we take the subtree formed by the nodes (2,8) and (2,9) to represent a modification of some constituent, say a noun, then we get a type like $n \multimap n$. However, the EPTD also contains positive and negative *cat* polarities. The node (2,0) has a negative *cat* polarity and we can consider it as an argument, meaning that the type we would like to derive would look like $F((2,0)) \multimap (n \multimap n)$. However, our strategy F has nothing to say about virtual features or (large) dominance constraints, so we will have to ignore those. If we look only at the positive and negative *cat* features, $F((2,0))$ becomes $np \multimap s$, which would give us $(np \multimap s) \multimap (n \multimap n)$ as the final type.

However, this translation is very lossy in terms of its discriminatory power. Many of the constraints which were enforced by the original EPTD are now ignored. We can imagine improving our strategy by using the feature structures as our types instead of atomic categories, which would not only ensure that e.g. the *sent_type* of the embedded clause must be *decl*, but would also capture the

information in the polarized $funct$ feature so that the relative pronoun *que* is not permitted to extract subjects. There are still constraints that we would miss out on though, such as the fact that the subject (2,3) of the clause (2,0) must precede any embedded clauses (2,6)...(2,5), the fact that any such embedded clauses must serve either the *modal* or the *obj* functions and the fact that the embedded clause (2,5) must have a subject (2,4).

Other constraints that we did not mention also do not have a direct equivalent in ACGs. One example would be a feature of IGs that we have not even mentioned which allows the grammar author to provide an *exhaustive* list of children of a node in a PTD.

Another kind of tricky constraint are the orange rectangles that we can see in Figure 1.9 which enforce that a constituent must be the leftmost or rightmost daughter of its parent. The problem with these is similar as with precedence relations. To model their impact on word order would be easy using an ACG lexicon, but to model their effect on restricting syntactic composition would be more tricky.

In the end, even though both formalisms (IGs and ACGs) are built on the framework of linear logic and their fundamental principles of syntactic composition have been linked together by Perrier's theorem, we conjecture that straightforwardly embedding Frigram into an ACG or constructing some automated technique for directly translating the Frigram EPTDs into ACG types and terms would not be feasible. We suspect that the incompatibility of these two formalisms is due to ACGs being a formalism based in the generative-enumerative framework of syntax while IGs are based in the model-theoretic approach.

In our grammar, we will still gladly use the lexicon Frilex though, which proves the usefulness of clearly defining the interface between the lexicon and the grammar in the architecture of Frigram. In our grammar, Frigram can still serve as a guide on how to interface with Frilex, on which phenomena need to be covered and in what detail, and what is a good way to structure the reusable components of a grammar of French.

Chapter 2

Implementing Wide-Coverage Abstract Categorial Grammars

Our goal is to develop large abstract categorial grammars which cover a variety of linguistic phenomena. All of these phenomena have been studied in detail but we would like to have a grammar which gracefully combines all these solutions. At this scale, working out the grammar on paper stops being sufficient to detect all the subtle interactions and it is desirable to have a formalized representation that can be reasoned about or tested by a computer.

To some extent, this need is already met by the **ACG development toolkit**[1]. The toolkit reads in signature and lexicon definitions and offers two modes of operation. In the first, it checks the validity of the defined ACG. That is to say, it checks whether the terms assigned by the lexicons are well-typed and whether their types are consistent with the object types dictated by the lexicon. The second mode of operation offers an interactive experience in which the user is free to run type inference on terms built on one of their signatures to find out whether they are well-typed and if so, what is their principal type. The interactive mode then lets the user apply a lexicon, or a composition of lexicons, to a term and get the β-normal form of the object term.

We consider the above capabilities vital for doing any involved grammar engineering in the framework of ACGs. However, there are some limitations which make it not sufficient for our needs. The foremost of these is that the signatures and lexicons are all explicitly realized in (primary) memory and the toolkit expects them to be given directly in a file. The grammars that we would like to develop will cover an exhaustive list of wordforms and the size of our grammar definitions would therefore grow proportionately with the size of our dictionary.

Given this setup, there are several approaches we might try, none of them

[1] `http://www.loria.fr/equipes/calligramme/acg/`

too appealing. We might write our metagrammar in a different format and use a tool which would combine the metagrammar and the lexical database[2] to produce the lexicalized[3] ACG in a direct representation, ready to be loaded by the toolkit. This workflow would make development on the grammar tedious as every time we would change the metagrammar, we would have to regenerate the direct representation and then load it in the ACG toolkit. With extra effort, the former cost could be alleviated by devising a system for regenerating only parts of the grammars which will have changed since last time. However, we would still have to pay the price of loading the entire grammar into the toolkit before being able to interact with it again.

Furthermore, we could pick a small representative fragment of the dictionary and test only on that fragment during the development of our grammar, to reduce the time it takes to both generate the direct representation of the grammar and the time it takes to load it into the toolkit. This would have made developing the grammar already tenable.

However, in our approach, we have opted to spend more time on tool development to ease subsequent work and we have developed a system which makes defining and experimenting with lexicalized ACGs easier. The metagrammars are defined as relational procedures which link the items of the lexical database to elements of signatures and lexicons. The grammars can be redefined without having to reload the lexical database or any file of similar scale. Type inference and lexicon application with β-normalization are provided as relations which can be used interactively or as part of larger programs. En example of such a larger program might be a routine to check the validity of an ACG or to test some of its properties.

In this chapter, we will spend some time explaining the design rationale underlying the system and the form it has now. The source code of the system in its current state can be perused at `https://github.com/jirkamarsik/acg-clj`.

2.1 The Tools and Techniques

The dominant decision which we made during the development of the system was to use *relational programming*. Relational programming is a discipline of logic programming where relations are written in a "pure" style [5]. This means that relations cannot be picky about which arguments are passed in as inputs and which as outputs, all modes must be permissible. Accepting such a discipline prohibits us from using some of the special operators of Prolog such as cut (`!`) and `copy_term/2`. Instead, we must resort to other techniques which do not break purity, like *constraint logic programming*, which allows us to express properties outside of the scope of unification by attaching constraints to terms and verifying them at the proper time.

[2] We will be using the term *lexical database* to refer to what is usually called a lexicon, in order not to cause confusion with the lexicons of ACGs.

[3] We will be using the term *lexicalized* in the context of ACGs to mean that the elements of the grammar (the typed constants and their images given by lexicons) are generated by a lexical database. This is not to be confused with the term used in [28].

By insisting on purity, I/O becomes more difficult as we would not allow the side-effectful pseudo-relations of Prolog. Therefore, we will not force ourselves to implement the entire program front-to-back using only relational programming. Instead, we will use a relational programming system, miniKanren, which is embedded inside a functional programming language. This lets us reap the benefits of relational programming inside the core logic of our program while deferring to the functional programming language to provide I/O and other bookkeeping operations.

2.1.1 The Case for Relational Programming

There are several reasons why we might want to choose a relational programming system for our implementation. First off, the problems which constitute the algorithmic core of our program are often mere textbook examples of how to use a relational programming system. This goes for type inference/checking/inhabitation, which are all implemented by a single relation which is often used as a kind of "Hello World" program in the world of relational programming. Similarly, implementing β-reduction cleanly (without having to handle variable clashes by generating new variables and α-converting) is a motivational example for introducing *nominal logic programming*. Finally, to reach feature parity with the ACG toolkit, one would only need to implement the mapping of a lexicon over a term, which is a simple functor.

The use of logic programming techniques is also very popular in our domain. If we look at the system demonstrations at the *Logical Aspects of Computational Linguistics 2012* conference, we see that 3 out of the 5 proposals have been implemented in a declarative/logic programming system (Oz, Prolog). 2 of the 3 systems are parsers for categorial grammars (automated theorem provers), which might turn out to be an interesting direction for our system to go into as well, as practical applications appear on the horizon.[4]

The third system at the conference was XMG, a language for writing metagrammars, which is another goal of our system, one which distinguishes it from the original ACG toolkit. We believe that using a declarative programming system such as miniKanren with the ability to extend it using a practical functional programming language presents an empowering way of defining metagrammars. This allows the grammar designer to compose her grammar in any way she sees fit and to use the metaprogramming facilities of the underlying programming language to transcribe the grammar in a way that makes the most sense in her case.

Finally, some of the structures that we will encounter in our domain are more directly modelled in mathematics as relations rather than functions. In a relational programming system, to implement a relation, one just writes a relation as it is, whereas if we were forced to encode everything in functions, we would have to simulate relational behavior manually. Frilex, our lexical database, is an example of a structure that has this relational shape since a

[4] Theorem provers are another problem area where miniKanren fits well, see αleanTAP [19].

single wordform can be mapped to multiple hypertags. We will see another example of a relation within our domain in 2.2.1 and we will also discuss the advantages and drawbacks of modelling functions using relations.

2.1.2 Choice of Implementation Language

There are two programming languages which play host to a substantial implementation of miniKanren (meaning one that has extensible unification and/or constraints and that includes support for nominal logic), Racket[5] and Clojure[6]. The two languages are very similar to each other and therefore the choice is of no great significance. An appealing feature of Clojure is that it uses abstract collection types pervasively instead of coupling the standard operations to specific data structures such as lists and cons cells and that it follows other similarly enlightened design decisions. Racket, on the other hand, has had more time to mature, has better metaprogramming facilities and is more recognized in academia. In developing our system, we have settled for Clojure. One of the advantages of doing so is that the Clojure implementation of miniKanren, dubbed core.logic, has inspired the software community and the project has attracted contributors that submit numerous issue reports and patches.[7]

2.2 Defining Signatures and Lexicons

In this section, we will go over the representation of signatures and lexicons in our implementation and the basic facilities for defining them.

2.2.1 Defining Signatures

In 1.3.1, we formally presented signatures as triples $<A, C, \tau>$. However, the set C can be always reconstructed as the domain of τ and the set A is just the set of all the atomic types occurring in the range of τ. Therefore, to define the signature $<A, C, \tau>$, it is enough to provide the function τ, which in its set-theoretical realization is just a set of pairs. In relational programming, a specific set can be easily encoded as a unary relation so a unary relation will serve as our representation of a signature and defining a signature will be a matter of constructing this relation.

The Relationship Between the Lexical Database and a Signature

We have our lexical database, a set of pairs of wordforms and hypertags, and we want to arrive at a set of pairs of constants and their types. This mapping can be decomposed into smaller parts by considering the contribution of every part of the lexical database individually. What will be the right model for the link between some part of the lexical database and some part of our signature?

[5] `https://github.com/calvis/cKanren`
[6] `https://github.com/clojure/core.logic`
[7] `http://dev.clojure.org/jira/browse/LOGIC`

Function of wordform We could imagine that every different wordform of the lexical database will end up being mapped to some constant in the signature. However, this is unsuitable since the lexical database is a relation that can assign more than one distinct hypertag to a single wordform (e.g. a wordform with different possible part of speech) and we would like to have constants of different types for each of these hypertags.

Function of hypertag Conversely, we might imagine that the hypertags of the lexical database are mapped via some function to the constants of the signature. Here we run into trouble when we consider variations in spelling or phonology. We might have two items in our lexical database that share the same hypertag but whose wordforms still differ. We would then like to have two distinct constants in our signature as well, so that when these two constants are mapped by the surface lexicon, they will yield two different strings.

Function of lexical entry Since neither the wordform nor the hypertag is enough, we could consider the entire lexical entry (an element of the lexical database, a pair of a wordform and one of its hypertags). This solves both of the above problems but it is still not perfect. Consider the case of a determiner and some semantic signature. Since our link between elements of the lexical database and of our signature is a function, there must be a constant in the signature for every item of our lexical database. If you recall in 1.3.5, the constants of the semantic signature Σ_{Sem} contained predicates for the common nouns *man* and *woman* and for the verb *loves* and some common logical furniture. However, it did not contain any constants which would correspond to the determiners *every* and *some* and therefore there would be no constant that the function could assign to these two wordforms.

Partial function of lexical entry We could fix the issue above by using a partial function. This accounts for cases when a lexical entry has no constants to represent it in a signature. Nevertheless, this only fixes half of the problem. We will consider the single lexical entry which represents e.g. the 3rd person singular present tense form *loves* of the transitive verb *love*. A legitimate account of subject/object scope ambiguity would be to consider two constants in our abstract syntactic signature that have distinct images in some semantic lexicon, one giving the outer scope to the subject and the other to the object. This means that in order to define such a signature, we would need to link a single lexical entry in our database to more than one constant in the signature.

Relation between lexical entry and constant Finally, considering all the above scenarios, the only mathematical model general enough to handle all of these cases is a relation. In our system, this will correspond exactly to a specific relation that will be provided by the grammar designer.

```
{:type [-> NP [-> NP S]]
 :id {:lex-entry {:wordform "loves"
                  :hypertag ...}
      :spec {:scope :subject-wide}}}
```

Figure 2.1: An example of a lexical constant from a hypothetical syntactic signature (the hypertag is elided for brevity).

Representing Constants and Defining Signatures

We have established that signatures will be defined by a relation, a relation between a lexical entry (a wordform and a hypertag) and a constant (an identity[8] and a type). The identity of a lexical constant also has a complex structure. We know that the lexical constants we define using the relation described above are always linked to a lexical entry. This lexical entry at least partially identifies the constant. For cases where we have more than one constant per lexical entry, such as in the example of the two different scope readings of *loves*, we need some extra piece of information to specify exactly which of the two constants we are referring to (we call this extra piece of information a *specifier*). This specifier can have any shape that the grammar designer desires. We can see an example of a lexical constant in Figure 2.1.

The system takes care of wiring up the relations and assembling the signature that yields these structures. The grammar designer's duty is to provide a 4-ary relation that links the four parts of a lexical constant: the wordform, the hypertag, the specifier and the type. This relation can be thought of as a multi-valued "function" that receives as input a wordform and a hypertag (that is, a single lexical entry) and outputs (possibly multiple) pairs of specifiers and types.

Since we are working in a functional programming language that supports manipulation of higher-order functions, we can provide helper functions which construct the signature relations for the grammar designer in some simple cases. Our library thus provides functions like `unitypedr`, which expects a type and returns a signature that assigns to every lexical item a single constant whose specifier is *nil* and whose type is the supplied type, or `ht->typer`, which expects a mapping from hypertag patterns to types and returns a signature which tries to map the hypertags of lexical items against the patterns and yields constants with the corresponding types and *nil* specifiers. When other patterns emerge, the grammar designer is completely free to implement her new abstractions using the full power of the functional programming language.

See Figure 2.2 for an example of implementing a lexicalized version of the Σ_{Synt} signature from 1.3.2 using the `ht->typer` abstraction.

In the above paragraphs, we have been referring to the constants that we were defining as *lexical constants*. That was to distinguish them from the *non-*

[8]The identity of a constant is simply the thing that distinguishes it from other constants of the same signature having the same type.

```
(ht->typer {{:head {:cat "n"}}      'N
            {:head {:cat "v"
                    :trans "true"}} (-> 'NP 'NP 'S)
            {:head {:cat "det"}}    (-> 'N (-> 'NP 'S) 'S)})
```

Figure 2.2: A lexicalized version of the syntactic signature from 1.3.2.

```
{:type [-> T [-> T T]]
 :id {:constant-name and?}}
```

Figure 2.3: An example of the representation of a non-lexicalized constant (in this case, it is the conjunction operator in a semantic signature).

lexical constants that we will introduce now. The reason for having non-lexical constants is that not every constant of every signature is linked to (generated by) some item of our lexical database. Consider, for example, the empty string ϵ and the string concatenation operator + of the Σ_{String} signature we gave in 1.3.2 or the quantifiers and logical connectives of the Σ_{Sem} signature in 1.3.5. These constants are not associated to any lexical entry and therefore their representations must be different. See Figure 2.3 for an example of the representation we use for non-lexicalized constants.

Signatures of non-lexical constants can be simply defined by a mapping from the constant names to their types as in Figure 2.4. Such signatures will generally not be sufficient by themselves and will need to be complemented by other signatures which contain lexical constants. Since we represent signatures as unary relations, combining them to produce their unions or intersections is just a matter of taking their disjunctions (`ors`) or conjunctions (`ands`), respectively.

With this, we have enough tools to fully define all the signatures of section 1.3 in a lexicalized manner, see Figure 2.5.

2.2.2 Defining Lexicons

We have explained how we can use elements of our lexical database to define signatures. Now, we will turn to lexicons. Similar to how we reduced signatures from the formal triples $<A, C, \tau>$ to just the type assignment function τ, we

```
(nonlex-sigr {'and?    (-> 'T 'T 'T)
              'imp?    (-> 'T 'T 'T)
              'forall? (-> (=> 'E 'T) 'T)
              'exists? (-> (=> 'E 'T) 'T)})
```

Figure 2.4: A signature of the non-lexical semantic constants belonging to the semantic signature of 1.3.5. The single arrow `->` corresponds to linear implication while the double arrow `=>` corresponds to intuitionistic implication.

```
(def synt-sig
  (ht->typer {{:head {:cat "n"}}      'N
              {:head {:cat "v"
                      :trans "true"}} (-> 'NP 'NP 'S)
              {:head {:cat "det"}}    (-> 'N (-> 'NP 'S) 'S)}))

(def string-sig
  (ors (nonlex-sigr {'++ (-> 'Str 'Str 'Str)
                     'empty-str 'Str})
       (unitypedr 'Str)))

(def sem-sig
  (ors (nonlex-sigr {'and?    (-> 'T 'T 'T)
                     'imp?    (-> 'T 'T 'T)
                     'forall? (-> (=> 'E 'T) 'T)
                     'exists? (-> (=> 'E 'T) 'T)})
       (ht->typer {{:head {:cat "n"}}      (-> 'E 'T)
                   {:head {:cat "v"
                           :trans "true"}} (-> 'E 'E 'T)})))
```

Figure 2.5: The definitions of the lexicalized versions of the example signatures of section 1.3.

can reduce lexicons from the pairs $<F, G>$ to just G, the part which operates on terms. F, or at least its relevant subset, can be easily retrieved from G. We take the pairs of abstract constants and object terms that G is composed of, we take their types and we arrive at pairs of abstract-level and object-level types. These pairs form a subset of $\hat{F}$. F (or at least its relevant subset), which only maps the atomic abstract types, can be inferred from this subset of $\hat{F}$.

In the relational framework, a mapping can be represented most directly as a binary relation and that is the representation of lexicons that we use in our system. In the previous subsection, we talked a lot about how individual entries in the lexical database generate the items of a signature. Solving the problem for signatures also solved it for lexicons, since a lexicon contains exactly one object term for every constant in its abstract signature. Therefore, a pair of an abstract constant and an object term belonging to some lexicon is generated by the same lexical entry that generated the abstract constant and its type.

Our representation of lexical constants was purposefully engineered so as to make the lexical entry directly accessible to the lexicon implementation. Since the lexicon is implemented by the grammar designer as a binary relation between constants and terms, the grammar designer can inspect the abstract constant and directly access the hypertag and wordform of the lexical entry that generated it, the specifier that was assigned to it and the type of the constant, and then express the object term assigned to the constant using any combination of these.

```
(def syntax-lexo
  (with-sig-consts string-sig
    (lexicalizer string-sig
                 (ht-lexiconr {{:head {:cat "n"}}
                               ,(rt (ll [_] _))
                               {:head {:cat "v"
                                       :trans "true"}}
                               ,(rt (ll [_ x y] (++ (++ x _) y)))
                               {:head {:cat "det"}}
                               ,(rt (ll [_ x R] (R (++ _ x))))}))))

(def sem-lexo
  (with-sig-consts sem-sig
    (orr (lexicalizer sem-sig
                      (ht-lexiconr {{:head {:cat "n"}}
                                    ,(rt (ll [_ x] (_ x)))
                                    {:head {:cat "v"
                                            :trans "true"}}
                                    ,(rt (ll [_ s o] (_ s o)))}))
         (ht-lexiconr {{:head {:cat "det"
                               :lemma "un"}}
                       ,(rt (ll [P Q] (exists? (il [x] (and? (P x)
                                                             (Q x))))))
                       {:head {:cat "det"
                               :lemma "chaque"}}
                       ,(rt (ll [P Q] (forall? (il [x] (imp? (P x)
                                                             (Q x))))))})))))
```

Figure 2.6: The definitions of the lexicalized versions of the lexicons of section 1.3.

Succinct Definitions of Lexicons

The above is already enough to define any lexicon. However, our toolkit presents convenience facilities that capture common forms of lexicons. With these, we will be able to define lexicalized versions of the lexicons of section 1.3 in a concise point-free style. Furthermore, the grammar designer is not limited to the convenience facilities we have provided and is completely free to use the same programming language we have used to build her own abstractions on top of ours right within her grammar definition.

We have the lexicon analogues to `nonlex-sigr` and `ht->typer`, they are called `nonlex-lexiconr` and `ht-lexiconr` and they work the same way as the signature versions but assign object terms instead of types.

When mapping lexical constants by a lexicon, we usually produce object terms which contain constants of the object signature that are generated by the

same lexical entry as the abstract constant (e.g. $\mathcal{L}_{syntax}(C_{\text{love}}) = \lambda^{\circ} xy.\ x + loves + y$). Managing this involves some tedious boilerplate that can be avoided by another convenience facility. First, we define a lexicon that produces the desired object term but with the object constant abstracted out (e.g. $\lambda^{\circ} cxy.\ x + c + y$). Then, we can apply the `lexicalizer` function to the object signature and to this lexicon to get the desired lexicon which fills in the correct object constants.

It is also useful to be able to easily insert the non-lexical constants of the object signature into the object terms we are giving when defining a lexicon. This service is provided by the anaphoric macro `with-sig-consts` which introduces all of the non-lexical constants of a signature into scope for easy use inside of lexicon definitions.

Finally, since lexicons are again just relations, we can take their unions and intersections using disjunction (`orr`) and conjunction (`andr`), respectively.

This gives us everything we need to define the lexicalized versions of the lexicons of section 1.3 in a concise and elegant manner. See Figure 2.6 for the definitions.

2.3 Checking Signatures and Lexicons

Since we let the grammar designer define signatures and lexicons using arbitrary relations, she can also end up defining relations which cannot be interpreted as well-formed signatures or lexicons. For this reason, our toolkit provides a set of testing facilities which help check that the signatures and lexicons are well-defined.

2.3.1 Checking Properties of Large Structures

As opposed to the existing **ACG development toolkit**, we cannot easily enumerate all of the elements in a given signature or lexicon and check that they are all consistent. The size of the lexicalized grammar is too large to make this approach practical.

We solve this problem by demanding that the grammar designer names some subset of the structure and testing is done only on that subset. In our case, the most practical solution has been to let the designer just list wordforms that generate a sufficient subset of the structure, i.e. the designer lists one wordform per every category she has handled in her grammar.

A more comfortable solution, which would also be more technically involved, would be to have the system test the grammar on an automatically deduced selection of constants such that they are guaranteed to cover all the cases and code paths in the grammar designer's definitions. Implementing such a system could still be manageable given the small number of primitives the relational programming system is built upon.

2.3.2 Checking Signatures

The crucial property that we need to check for in signatures is that the relation between the identities of constants and their types is a function. This simply reduces to querying the system for the number of typed constants belonging to the signature and having a specific identity. If this number is one for all the identities in our test set, we have verified that (the tested subset of) the signature assigns exactly one type to every one of its constants.

2.3.3 Checking Lexicons

For lexicons, we check the three following properties:

The lexicon is a function, assigning exactly one object term to each abstract constant. The verification proceeds analogously to the case of checking signatures (subsection 2.3.2).

The lexicon assigns only well-typed object terms. This is a consequence of the homomorphism property of lexicons. We verify it by checking whether we can infer some type for every object term assigned to a constant in the test set.

The lexicon has the homomorphism property. We use the process that we described in subsection 2.2.2 for inferring the type mapping of a lexicon from its term mapping.

We first enumerate the pairs of abstract constants belonging to the test set and the object terms assigned to them by the lexicon. We then take the types of these pairs. Finally, we state, by using unification, that this set of pairs is a subset of the homomorphic extension of some mapping of atomic abstract types and let the relational programming system find us that mapping. If it succeeds, we have verified the homomorphism property of the lexicon.

The above technique of verifying the signature and lexicon definitions by automated tests is not only useful for checking their well-formedness. The grammar designer can go further and implement new tests which demonstrate grammar-specific properties, an example of which would be the *cat-funct* principle in Frigram that imposes constraints on the relationship between the *cat* and *funct* features in the EPTDs of the grammar.

Chapter 3

Treatment of Multiple Linguistic Constraints

In this chapter, we will attempt to incorporate accounts of several linguistic phenomena in a single framework, highlight the modularity issues this brings up.

3.1 Negation

We will start with an account of the paired grammatical words mechanism for negation in French, specifically the interaction between negative noun phrases and the particle *ne*. Recall the EPTDs we have seen in 1.4.2, repeated here on Figure 3.1.

At a basic level, *aucun* functions like any other determiner and *ne* as any other verb modifier. However, we want to encode the constraint that whenever a verb is modified by *ne*, the modified verb will demand that one of its arguments contains a negative determiner. Since we need to make a distinction between phrases that contain negative determiners and those that do not, we will need to provide different types for both (two terms with the same type are indistinguishable w.r.t. syntactic composition in a type-logical grammar such as ACGs). Our type for *aucun* will thus end up being[1]

$$N\_NEG{=}F \multimap ((NP\_NEG{=}T \multimap S) \multimap S)$$

instead of the usual

$$N \multimap ((NP \multimap S) \multimap S)$$

The type for *le* will need to be able to handle cases where its argument noun either already contains a negative determiner or not.

[1] Strictly following the EPTD for *aucun* would lead us to give it a type $DET\_NEG{=}T$ and then give the two types $DET\_NEG{=}F \multimap ((NP\_NEG{=}F \multimap S) \multimap S)$ and $DET\_NEG{=}T \multimap ((NP\_NEG{=}T \multimap S) \multimap S)$ to nouns. In this demonstration, we prefer to align our examples with the treatment which is customary in ACG research.

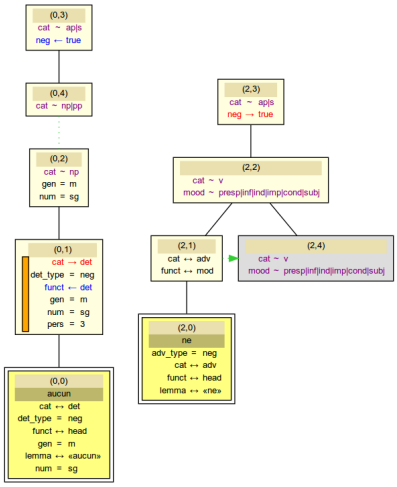


Figure 3.1: Frigram EPTDs for a negative determiner and the paired grammatical word *ne*.

$$N_{aucun} : N\_NEG{=}F \multimap ((NP\_NEG{=}T \multimap S) \multimap S)$$
$$N_{le_1} : N\_NEG{=}F \multimap ((NP\_NEG{=}F \multimap S) \multimap S)$$
$$N_{le_2} : N\_NEG{=}T \multimap ((NP\_NEG{=}T \multimap S) \multimap S)$$

The type of *ne* will be more verbose, as it will consume a verb and transform its valency so that it demands that exactly one of its NP arguments contains a negative determiner.

$$N_{ne_{tv_1}} : (NP\_NEG{=}F \multimap NP\_NEG{=}F \multimap S) \multimap (NP\_NEG{=}T \multimap NP\_NEG{=}F \multimap S)$$
$$N_{ne_{tv_2}} : (NP\_NEG{=}F \multimap NP\_NEG{=}F \multimap S) \multimap (NP\_NEG{=}F \multimap NP\_NEG{=}T \multimap S)$$

Furthermore, we could add the case when both the subject and the object contain negative determiners, such as in (10). This case is currently not covered by Frigram (though it could be), but is considered grammatical by French speakers. Finally, in a complete grammar, *ne* would have to provide types capable of transforming all the other syntactic valencies.

(10) Aucune fourmi n'aime aucun tatou.

$$N_{ne_{tv_3}} : (NP\_NEG{=}F \multimap NP\_NEG{=}F \multimap S) \multimap (NP\_NEG{=}T \multimap NP\_NEG{=}T \multimap S)$$

However, this is not the only change we have to effect on our grammar in order to properly handle the paired grammatical words for negation. Consider

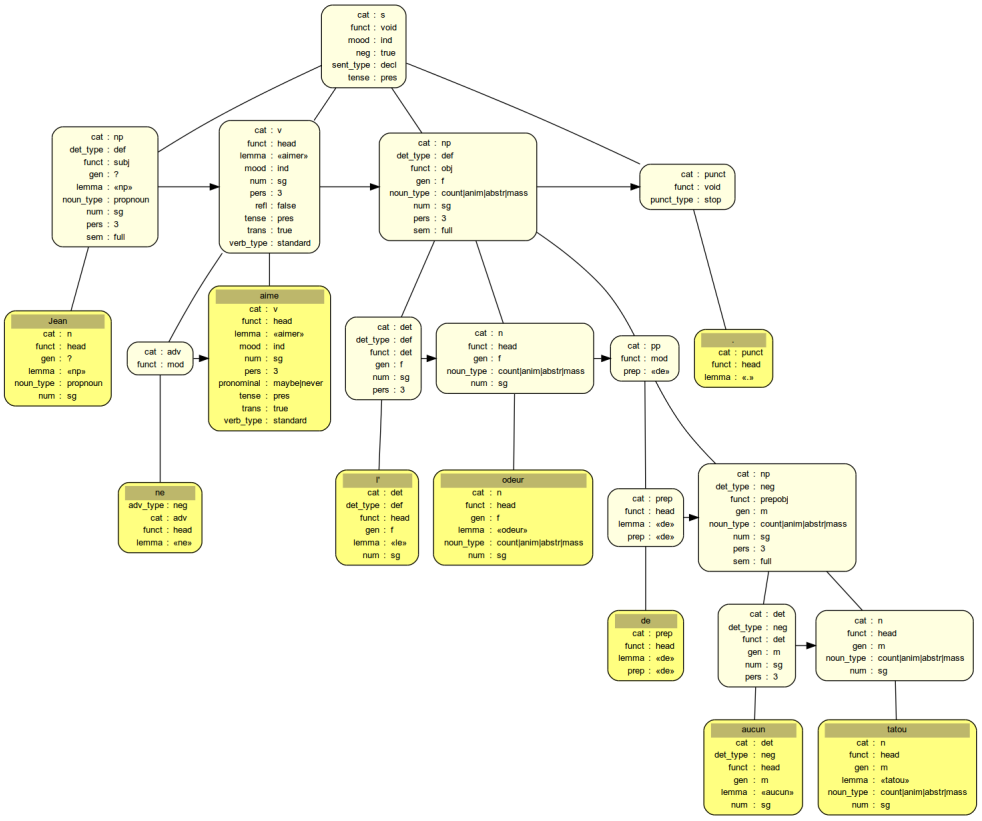


Figure 3.2: One of the parse trees assigned to sentence (7) by Frigram/Leopar.

the example sentence (7) and the parse tree assigned to it by Frigram on Figure 3.2. It is not enough that the embedded noun phrase *aucun tatou* has a type that tells us it contains a negative determiner. We need this bit of internal information also at the level of the enclosing noun phrase *l'odeur d'aucun tatou*, in which the former noun phrase is embedded, since this is a property of the enclosing noun phrase that is crucial to its syntactic combinatorics. This presupposes the existence of a mechanism for propagating this kind of information up the parse tree. In the case of our ACG, this means that all the functions which modify phrases must be aware whether or not their argument contains a negative determiner and to convey this information in its result type as well. Moreover, since the negative determiner can also come from a complement of a prepositional phrase, the types of prepositions will need to take into account the presence of negative determiners in both their complements and the phrases they modify.

$$
\begin{aligned}
N_{que_1} &: (NP\_NEG{=}F \multimap S) \multimap N\_NEG{=}F \multimap N\_NEG{=}F \\
N_{que_2} &: (NP\_NEG{=}F \multimap S) \multimap N\_NEG{=}T \multimap N\_NEG{=}T \\
N_{de_1} &: NP\_NEG{=}F \multimap N\_NEG{=}F \multimap N\_NEG{=}F \\
N_{de_2} &: NP\_NEG{=}F \multimap N\_NEG{=}T \multimap N\_NEG{=}T \\
N_{de_3} &: NP\_NEG{=}T \multimap N\_NEG{=}F \multimap N\_NEG{=}T
\end{aligned}
$$

In the above types, we not only propagate information about the presence of a negative determiner, we also introduce a constraint in the mechanism stating that it cannot be the case that there is a negative determiner both in the modified noun and in its noun phrase complement (i.e. phrases like *aucun tatou d'aucun homme* are not acceptable).

If we suppose the presence of the appropriate lexical items typed as in the examples of 1.3, we have a type system in which we can type the syntactic terms corresponding to the sentences (11), (12) and (13) while making it impossible to type the structures corresponding to sentences (14) and (15).

In sentence (14), we have no constant $N_{ne_{tv_?}}$ for *ne* which would yield a verb capable of consuming two noun phrases that contain no negative determiners.

In sentence (15), there are multiple problems with typing. First, N_{chasse} requires its first argument to be of type $NP\_NEG{=}F$, a noun phrase with no negative determiner. This means that $\lambda^{\circ}x.\ N_{chasse}\ x\ y$ has type $NP\_NEG{=}F \multimap S$ which makes it an invalid argument for the function $N_{aucun}\ N_{loup}$, which has type $(NP\_NEG{=}T \multimap S) \multimap S$. Even if we were able to assign it a type compatible with N_{que_1}, the final relative clause would map N_{tatou} to another term of type $N\_NEG{=}F$, which has no "free" negative determiners. N_{le_1} will map this to a term typed $(NP\_NEG{=}F \multimap S) \multimap S$. Finally, this term is not capable of taking $(\lambda^{\circ}x.\ N_{ne_{iv}}\ N_{court}\ x)$ as its argument, since its type is $NP\_NEG{=}T \multimap S$.

(11) Aucune fourmi n'aime aucun tatou.
$(N_{aucune}\ N_{fourmi})\ (\lambda^{\circ}x.\ (N_{aucun}\ N_{tatou})\ (\lambda^{\circ}y.\ N_{ne_{tv_3}}\ N_{aime}\ x\ y))$

(12) Jean n'aime l'odeur d'aucun tatou.
$(N_{le_2}\ (N_{de_3}\ (N_{aucun}\ N_{tatou})\ N_{odeur}))\ (\lambda^{\circ}y.\ N_{ne_{tv_2}}\ N_{aime}\ N_{Jean}\ y)$

(13) Le tatou qu'aucun loup ne chasse court.
$(N_{le_1}\ (N_{que_1}\ (\lambda^{\circ}y.\ (N_{aucun}\ N_{loup})\ (\lambda^{\circ}x.\ N_{ne_{tv_1}}\ N_{chasse}\ x\ y))\ N_{tatou}))\ (\lambda^{\circ}x.\ N_{court}\ x)$

(14) * Jean n'aime le tatou.
* $(N_{le_1}\ N_{tatou})\ (\lambda^{\circ}y.\ N_{ne_{tv_?}}\ N_{aime}\ N_{Jean}\ y)$

(15) * Le tatou qu'aucun loup chasse ne court.
* $(N_{le_1}\ (N_{que}\ (\lambda^{\circ}y.\ (N_{aucun}\ N_{loup})\ (\lambda^{\circ}x.\ N_{chasse}\ x\ y))\ N_{tatou}))\ (\lambda^{\circ}x.\ N_{ne_{iv}}\ N_{court}\ x)$

There are two things that we would like to highlight about this grammatical treatment. First off, we had to refine our types to make them convey more than one piece of information in a single type (e.g. something is a noun phrase and at the same time it is something that contains a negative determiner). Second, a property that we wish to express in the type of a term may be due to one of its subconstituents and it is therefore necessary to encode the propagation of this property up the parse tree in the types of all the intervening operators. In our case, it meant that handling negative determiners and the paired grammatical words for negation required us to not only define and type constants for these two categories, but to also change the types of noun modifiers (*que ...*) and prepositions (*de*).

3.2 Extraction

We will now turn to another phenomenon, extraction in relative clauses, focus on one of its properties, the fact that subjects can be extracted using *qui* only from the relative clause itself whereas objects can be extracted using *que* from embedded clauses contained therein, and present an implementation of this constraint by way of types in an ACG. The solution that we will show was published in [24]. Our presentation is slightly adapted to conform to the patterns seen in previous examples and it does not use dependent types as they are not essential to this specific constraint.

The key idea in the treatment presented below is to give a special type to extracted constituents. We will thus have two types for noun phrases: $NP\_VAR{=}T$, which stands for "empty" noun phrases introduced by extraction, and $NP\_VAR{=}F$, which stands for the other noun phrases, i.e. proper names and nouns with determiners. The grammar gives no constant capable of constructing a value of type $NP\_VAR{=}T$. The only way to obtain a term with this type is to abstract over a variable having the type. Then, the types of relative pronouns end up being $(NP\_VAR{=}T \multimap S) \multimap N \multimap N$, ensuring that the incomplete clause given as the first argument is a λ-abstraction whose argument has type $NP\_VAR{=}T$.

Once we have this kind of information in the type system, we can start distinguishing clauses containing rooted extraction, where a subject or object is extracted directly from the relative clause, embedded extraction, where an object is extracted from a clause embedded inside the relative clause, or no

extraction. Types of functions which produce values of type S will be sensitive to the presence of extracted variables in their noun phrase and clausal arguments.

$$
\begin{aligned}
E_{dort_1} &: NP\_VAR{=}F \multimap S\_EXT{=}NO \\
E_{dort_2} &: NP\_VAR{=}T \multimap S\_EXT{=}ROOT \\
E_{aime_1} &: NP\_VAR{=}F \multimap NP\_VAR{=}F \multimap S\_EXT{=}NO \\
E_{aime_2} &: NP\_VAR{=}T \multimap NP\_VAR{=}F \multimap S\_EXT{=}ROOT \\
E_{aime_3} &: NP\_VAR{=}F \multimap NP\_VAR{=}T \multimap S\_EXT{=}ROOT \\
E_{dit\ que_1} &: NP\_VAR{=}F \multimap S\_EXT{=}NO \multimap S\_EXT{=}NO \\
E_{dit\ que_2} &: NP\_VAR{=}T \multimap S\_EXT{=}NO \multimap S\_EXT{=}ROOT \\
E_{dit\ que_3} &: NP\_VAR{=}F \multimap S\_EXT{=}ROOT \multimap S\_EXT{=}EMB \\
E_{dit\ que_4} &: NP\_VAR{=}F \multimap S\_EXT{=}EMB \multimap S\_EXT{=}EMB
\end{aligned}
$$

Now all of the infrastructure is in place for us to express the type of the relative pronouns *qui* and *que*.

$$
\begin{aligned}
E_{qui} &: (NP\_VAR{=}T \multimap S\_EXT{=}ROOT) \multimap N \multimap N \\
E_{que_1} &: (NP\_VAR{=}T \multimap S\_EXT{=}ROOT) \multimap N \multimap N \\
E_{que_2} &: (NP\_VAR{=}T \multimap S\_EXT{=}EMB) \multimap N \multimap N
\end{aligned}
$$

The above types ensure that the abstracted variables corresponding to traces of extracted constituents will be given the type $NP\_VAR{=}T$, that *qui* can only be used to extract constituents from root positions while *que* allows for extraction from both root and embedded positions.[2]

We also note that the splitting of the clause type S into three finer types $S\_EXT{=}NO$, $S\_EXT{=}ROOT$ and $S\_EXT{=}EMB$ means that types that work with S regardless of its extraction status will have to provide alternatives for all the possible variants.

$$
\begin{aligned}
E_{le_1} &: N \multimap ((NP\_VAR{=}F \multimap S\_EXT{=}NO) \multimap S\_EXT{=}NO) \\
E_{le_2} &: N \multimap ((NP\_VAR{=}F \multimap S\_EXT{=}ROOT) \multimap S\_EXT{=}ROOT) \\
E_{le_3} &: N \multimap ((NP\_VAR{=}F \multimap S\_EXT{=}EMB) \multimap S\_EXT{=}EMB)
\end{aligned}
$$

While we have introduced the notion of an $NP\_VAR{=}T$ for describing traces, we can cover another element of our running fragment, the preposition *de*, and enforce the constraint that *qui*/*que* cannot extract prepositional complements by providing only a single type for *de* that accepts only unmoved noun phrases as complements.

[2]NB: The constraint that *qui* can only extract subjects and that *que* can only extract objects is not enforced here. Likewise, this fragment does not handle multiple extraction.

$$E_{de} : NP\_VAR{=}F \multimap N \multimap N$$

Given this type system, we can assign the type $S\_EXT{=}NO$ to the term expressing the syntactic structure of (17) but we cannot a find a typable term for sentence (16), where *qui* is used to extract a subject from a clause embedded in a relative clause. The problem with sentence (16) is that the type of the relative clause is $NP\_VAR{=}T \multimap S\_EXT{=}EMB$ which is not a valid argument for any constant representing the relative pronoun *qui*. Typing the variable t with $NP\_VAR{=}F$ instead of $NP\_VAR{=}T$ and using E_{aime_1} instead of E_{aime_2} would not help, since the resulting relative clause would then have type $NP\_VAR{=}F \multimap S\_EXT{=}NO$, which would not be admitted by any relative pronoun.

(16) * Le tatou qui Jean dit que _ aime Marie dort.
* $(E_{le_1}\ (E_{qui_?}\ (\lambda^\circ t.\ E_{dit\ que_3}\ E_{Jean}\ (E_{aime_2}\ t\ E_{Marie}))\ E_{tatou}))\ (\lambda^\circ x.\ E_{dort_1}\ x)$

(17) Le tatou que Jean dit que Marie aime _ dort.
$(E_{le_1}\ (E_{que_2}\ (\lambda^\circ t.\ E_{dit\ que_3}\ E_{Jean}\ (E_{aime_3}\ E_{Marie}\ t))\ E_{tatou}))\ (\lambda^\circ x.\ E_{dort_1}\ x)$

As was the case with our treatment of negation, we refined our familiar types by tacking on new kinds of information and defined rules for how these pieces of information percolate up the parse tree all the way to the topmost level still pertinent for ensuring grammaticality.

3.3 Agreement

Finally, we will look at another example of implementing a linguistic constraint in ACGs by covering the phenomenon of number agreement.

Number agreement motivates a straightforward refinement of our types, where all noun and noun phrase types carry number information.

$$\begin{aligned}
A_{Marie} &: NP_NUM{=}SG \\
A_{tatou} &: N_NUM{=}SG \\
A_{tatous} &: N_NUM{=}PL \\
A_{le} &: N_NUM{=}SG \multimap ((NP_NUM{=}SG \multimap S) \multimap S) \\
A_{les} &: N_NUM{=}PL \multimap ((NP_NUM{=}PL \multimap S) \multimap S) \\
A_{dort} &: NP_NUM{=}SG \multimap S \\
A_{dorment} &: NP_NUM{=}PL \multimap S \\
A_{aime_1} &: NP_NUM{=}SG \multimap NP_NUM{=}SG \multimap S \\
A_{aime_2} &: NP_NUM{=}SG \multimap NP_NUM{=}PL \multimap S
\end{aligned}$$

$$
\begin{aligned}
A_{aiment_1} &: NP\_NUM{=}PL \multimap NP\_NUM{=}SG \multimap S \\
A_{aiment_2} &: NP\_NUM{=}PL \multimap NP\_NUM{=}PL \multimap S \\
A_{qui_1} &: (NP\_NUM{=}SG \multimap S) \multimap N\_NUM{=}SG \multimap N\_NUM{=}SG \\
A_{qui_2} &: (NP\_NUM{=}PL \multimap S) \multimap N\_NUM{=}PL \multimap N\_NUM{=}PL \\
A_{de_1} &: NP\_NUM{=}SG \multimap N\_NUM{=}SG \multimap N\_NUM{=}SG \\
A_{de_2} &: NP\_NUM{=}PL \multimap N\_NUM{=}SG \multimap N\_NUM{=}SG \\
A_{de_3} &: NP\_NUM{=}SG \multimap N\_NUM{=}PL \multimap N\_NUM{=}PL \\
A_{de_4} &: NP\_NUM{=}PL \multimap N\_NUM{=}PL \multimap N\_NUM{=}PL
\end{aligned}
$$

Given the above signature, we can assign the type S to the syntactic term of sentence (18) but not to the one of sentence (19). In the latter sentence, the type of the variable t has to be $NP\_NUM{=}SG$ since it is used as an argument to A_{dort}. This means that the relative clause *dort* has type $NP\_NUM{=}SG \multimap S$. However, there is no type for *qui* which would allow us to modify the plural noun A_{tatous} of type $N\_NUM{=}PL$ by a relative clause with a missing singular noun phrase, which has the type $NP\_NUM{=}SG \multimap S$.

(18) Les tatous de Marie dorment.
$(A_{les}\ (A_{de}\ A_{Marie}\ A_{tatous}))\ (\lambda^{\circ}x.\ A_{dorment}\ x)$

(19) * Marie aime les tatous qui dort.
* $(A_{les}\ (A_{qui_?}\ (\lambda^{\circ}t.\ A_{dort}\ t)\ A_{tatous}))\ (\lambda^{\circ}y.\ A_{aime_2}\ A_{Marie}\ y)$

Once again, we have expressed a linguistic constraint by refining our types, augmenting them with a new bit of information, we constrained the arguments of functions using these new types and we have described how this new type information propagates from the given arguments to the produced values.

3.4 Putting It All Together

In this section, we will see what it takes to enforce all of the constraints introduced in the preceding sections (negation, extraction and agreement) at the same time, which will give us some idea on how to go about building a grammar which handles a wide variety of such phenomena.

In a grammar that handles negation, extraction and agreement, our types will need to carry all of the refinements we introduced before. To recapitulate, NP will be parameterized by whether or not it contains a negative determiner ($\_NEG{=}T$ and $\_NEG{=}F$ respectively), whether or not it is a trace from an extraction ($\_VAR{=}T$ and $\_VAR{=}F$) and its number ($\_NUM{=}SG$ or $\_NUM{=}PL$). For N, we have the same refinements except for VAR since extraction only moves NPs. Clauses, of type S, will be discriminated based on the level of extraction within them ($\_EXT{=}NO$, $\_EXT{=}ROOT$ and $\_EXT{=}EMB$).

The individual types will have to respect all of the constraints at the same time, each one being a complete specification of a possible situation (complete

w.r.t. the features listed above). To see and appreciate what this means in practice, we give the types for the preposition *de*:

$$C_{de_1} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}SG) \multimap (N\_NEG{=}F\_NUM{=}SG) \multimap (N\_NEG{=}F\_NUM{=}SG)$$
$$C_{de_2} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}SG) \multimap (N\_NEG{=}F\_NUM{=}PL) \multimap (N\_NEG{=}F\_NUM{=}PL)$$
$$C_{de_3} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}SG) \multimap (N\_NEG{=}T\_NUM{=}SG) \multimap (N\_NEG{=}T\_NUM{=}SG)$$
$$C_{de_4} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}SG) \multimap (N\_NEG{=}T\_NUM{=}PL) \multimap (N\_NEG{=}T\_NUM{=}PL)$$
$$C_{de_5} : (NP\_NEG{=}T\_VAR{=}F\_NUM{=}SG) \multimap (N\_NEG{=}F\_NUM{=}SG) \multimap (N\_NEG{=}T\_NUM{=}SG)$$
$$C_{de_6} : (NP\_NEG{=}T\_VAR{=}F\_NUM{=}SG) \multimap (N\_NEG{=}F\_NUM{=}PL) \multimap (N\_NEG{=}T\_NUM{=}PL)$$
$$C_{de_7} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}PL) \multimap (N\_NEG{=}F\_NUM{=}SG) \multimap (N\_NEG{=}F\_NUM{=}SG)$$
$$C_{de_8} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}PL) \multimap (N\_NEG{=}F\_NUM{=}PL) \multimap (N\_NEG{=}F\_NUM{=}PL)$$
$$C_{de_9} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}PL) \multimap (N\_NEG{=}T\_NUM{=}SG) \multimap (N\_NEG{=}T\_NUM{=}SG)$$
$$C_{de_{10}} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}PL) \multimap (N\_NEG{=}T\_NUM{=}PL) \multimap (N\_NEG{=}T\_NUM{=}PL)$$
$$C_{de_{11}} : (NP\_NEG{=}T\_VAR{=}F\_NUM{=}PL) \multimap (N\_NEG{=}F\_NUM{=}SG) \multimap (N\_NEG{=}T\_NUM{=}SG)$$
$$C_{de_{12}} : (NP\_NEG{=}T\_VAR{=}F\_NUM{=}PL) \multimap (N\_NEG{=}F\_NUM{=}PL) \multimap (N\_NEG{=}T\_NUM{=}PL)$$

This highlights two important problems with the current approach. One is the untenable complexity. Even though the three mechanisms are completely independent of each other, here we are forced to refer to all of them in every type. It is no longer easy to glance at the types assigned to *de* and see how it behaves with respect to, for example, the mechanism of negation (compare with the type assignments we gave to *de* in previous subsections). Another cause of unreadability of this signature is due to the sheer number of types on display, which leads us to our second issue.

The signature demonstrates an exponential growth in the number of typed constants included. In cases where the behaviors of a wordform across the different mechanisms are independent, the complete type signature will need to provide a type for every combination of situations in all of the mechanisms. Assuming that every mechanism will assign on average more than one type to every wordform, the number of the wordform's types in the final signature will be exponential w.r.t. the number of different mechanisms handled.

We can try making the enumeration of these types easier for the grammar author by introducing metavariables for the values of the features. This means that we could write the following 3 templates instead of the 12 types above:

$$\forall x, y \in \{SG, PL\}$$

$$C_{de_{1(x,y)}} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}x) \multimap (N\_NEG{=}F\_NUM{=}y) \multimap (N\_NEG{=}F\_NUM{=}y)$$
$$C_{de_{2(x,y)}} : (NP\_NEG{=}F\_VAR{=}F\_NUM{=}x) \multimap (N\_NEG{=}T\_NUM{=}y) \multimap (N\_NEG{=}T\_NUM{=}y)$$
$$C_{de_{3(x,y)}} : (NP\_NEG{=}T\_VAR{=}F\_NUM{=}x) \multimap (N\_NEG{=}F\_NUM{=}y) \multimap (N\_NEG{=}T\_NUM{=}y)$$

This is getting more concise and easier to comprehend. We can take this

approach further by not only quantifying over variables but also computing some of the values using functions:

$$\forall x, y \in \{SG, PL\}, \forall p, q \in \{F, T\}, p \wedge q \neq T$$

$$C_{de(x,y,p,q)} : (NP\_NEG{=}p\_VAR{=}F\_NUM{=}x) \multimap (N\_NEG{=}q\_NUM{=}y) \multimap (N\_NEG{=}(p \vee q)\_NUM{=}y)$$

The above is vastly more concise and manages to convey the intent more clearly (the intent being that it cannot be the case that there are negative determiners in both the complement noun phrase and the modified noun and that the value of the result's NEG feature is always the disjunction of the NEG features of the two arguments).

As of right now, we have introduced this kind of rule only as meta-notation. This means that the real underlying grammar that will end up being used by some algorithm still suffers from the same exponential cardinality of the signature. Fortunately, there exists a well-studied construction in type theory that lets us assign to terms generic types such as the one defined in the template above. This construction is the *dependent product* and its utility in ACGs has already been noted.[3]

While extending the type system by including dependent types is definitely useful and it helps to solve the problem of having our signatures grow exponentially in size, it does not solve our problem with the complexity of the new grammar. Granted, the new meta-type written in the dependent style is more readable and easier to reason about than the enumeration of 12 basic types we presented at the beginning of this section, however, the result is still complex in that it forces us to describe the behavior of three independent mechanisms in one place. This one place happens to be a wordform which has seemingly little connection to these mechanisms, the preposition *de*. We can therefore imagine that it will participate in many other phenomena and that the type we assign to it will grow beyond our abilities to reason about it effectively.

Furthermore, the current notation does not even make it obvious that these are independent mechanisms. One has to examine the type to make sure that there is no interaction.

In the next chapter, we will present our proposed solution to the problem of writing simple grammars that cover multiple linguistic constraints.

[3] See [10] for its introduction into the domain of ACGs, [9] for a small example grammar, which handles the interaction between number and gender agreement and coordination, and finally see [24] for using dependent types to handle several constraints regarding extraction (including the one we covered in 3.2).

Chapter 4

Graphical Abstract Categorial Grammars

In this chapter, we will introduce a generalization of abstract categorial grammars that gracefully solves the modularity issues that we saw in the previous chapter.

4.1 Intersections of Languages

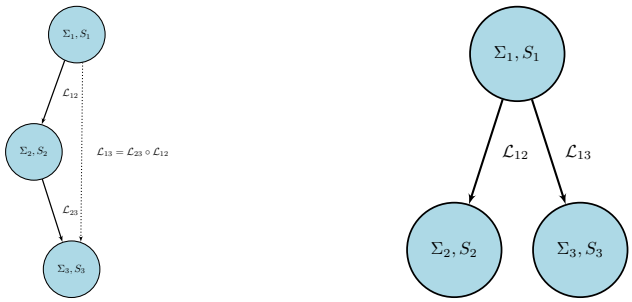
Before we proceed to define our extension of ACGs, we will first need to present the manner in which ACGs and their compositions are formulated in ACG literature.

4.1.1 The Usual Formalization of ACGs

Let us consider a system with three signatures Σ_1, Σ_2 and Σ_3, their respective distinguished types S_1, S_2 and S_3 and two lexicons $\mathcal{L}_{12} : \Sigma_1 \to \Sigma_2$ and $\mathcal{L}_{23} : \Sigma_2 \to \Sigma_3$ such that $\mathcal{L}_{12}(S_1) = S_2$ and $\mathcal{L}_{23}(S_2) = S_3$. See Figure 4.1(a) for a diagrammatic visualization of this structure.

We can now define languages A_1, A_2 and A_3 where A_i is the set of terms from $\Lambda(\Sigma_i)$ that have the type S_i. Such languages, which are defined only in terms of a signature and some distinguished type, are called *abstract languages* in the terminology of ACGs. They consist of terms whose structure is constrained only by the types assigned by the signature.

Further than that, we can look at the languages $\mathcal{L}_{12}(A_1)$ and $\mathcal{L}_{23}(A_2)$, images of the abstract languages given by the lexicons. Because of the homomorphism property of lexicons, all elements of $\mathcal{L}_{12}(A_1)$ will have the type $\mathcal{L}_{12}(S_1) = S_2$ and will therefore also belong to A_2 (similarly, we have that $\mathcal{L}_{23}(A_2) \subseteq A_3$). These new languages, which are formed by mapping an ACG abstract language using a lexicon, are called *object languages*. Their terms are constrained not only by the types of the signature they are built upon but also by the type system of the abstract signature and the lexicon connecting the two.



(a) Serial composition of ACGs. (b) Parallel composition of ACGs.

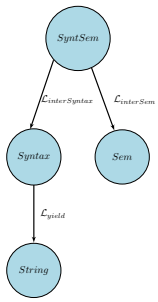
Figure 4.1: Two modes of composition for ACGs.

Finally, we can also take the composition of lexicons $\mathcal{L}_{13} = \mathcal{L}_{23} \circ \mathcal{L}_{12}$, which itself is also a lexicon, and map it over A_1. This gives us a language $\mathcal{L}_{13}(A_1) = \mathcal{L}_{23}(\mathcal{L}_{12}(A_1))$ which is at the same time the image of the object language $\mathcal{L}_{12}(A_1)$ given by $\mathcal{L}_{23}$ and also the image of the abstract language A_1 given by $\mathcal{L}_{13}$ (thus it is itself also an object language). This points to an interesting property of ACG languages: they are closed under transformation by a lexicon.

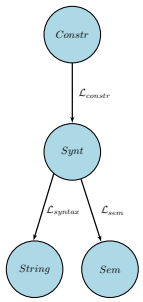
In the ACG literature, ACGs are defined as quadruples $\mathcal{G} = <\Sigma_A, \Sigma_O, \mathcal{L}, S>$ where Σ_A is the abstract signature, Σ_O is the object signature, $\mathcal{L} : \Sigma_A \to \Sigma_O$ is a lexicon linking the two and S is a distinguished type of the abstract signature Σ_A. From this grammar, we can define two languages. The abstract language $\mathcal{A}(\mathcal{G}) = \{t \in \Lambda(\Sigma_A) \mid \vdash_{\Sigma_A} t : S\}$ and the object language $\mathcal{O}(\mathcal{G}) = \mathcal{L}(\mathcal{A}(\mathcal{G})) = \{t \in \Lambda(\Sigma_O) \mid \exists u \in \mathcal{A}(\mathcal{G}), \mathcal{L}(u) = t\}$. The composition of two ACGs $\mathcal{G}_1 = <\Sigma_1, \Sigma_2, \mathcal{L}_{12}, S_1>$ and $\mathcal{G}_2 = <\Sigma_2, \Sigma_3, \mathcal{L}_{23}, S_2>$ is defined as $\mathcal{G}_2 \circ \mathcal{G}_1 = <\Sigma_1, \Sigma_3, \mathcal{L}_{23} \circ \mathcal{L}_{12}, S_1>$.

The definition of ACGs given above is quite practical in that it is complete, i.e. it gives a precise account of what set of languages are generated by ACGs (abstract and object languages for which there exist the quadruples described above). This contrasts with the presentation of ACGs that we gave in 1.3, where we defined ACGs as collections of signatures associated with distinguished types and connected with lexicons. Our definition only introduced the requisite machinery, but delayed the delimitation of the ways languages can be defined. Leaving this question open will let us answer it now by providing an extension of ACGs which will help us overcome the modularity issues that have been the theme of Chapter 3.[1]

[1]Incidentally, this was not the reason for the way we presented ACGs in 1.3. We chose this style as it grouped together the information in a more natural way. We approach signatures as descriptions of domains (strings, syntax, semantics), each one being associated with a distinguished type telling us which objects the domain has set out to describe (strings, sentences, truth values). Having distinguished types directly attached to signatures allows us to omit them from the ACG quadruples, $<\Sigma_A, \Sigma_O, \mathcal{L}, \not{S}>$. However, this only leaves us with triples containing the two signatures and the lexicon connecting them, and since we can infer the signatures themselves from the domain and the range of the lexicon $\mathcal{L} : \Sigma_A \to \Sigma_O$, we do not have much motivation to introduce these tuples, $<\not{\Sigma_A}, \not{\Sigma_O}, \mathcal{L}, \not{S}>$. Instead of composing



(a) Transduction from an abstract signature.



(b) Constraint on an abstract signature.

Figure 4.2: Mixing the ACG modes of composition.

4.1.2 Patterns of Composition in ACGs

Before we turn to our proposal, we will quickly recall some of the other styles of composition used in ACGs (a more detailed exposition can be found in [24]). Besides making the object signature of one grammar be the abstract signature of another as we saw before (Figure 4.1(a)), we can have two grammars share the same abstract signature allowing us to transduce between terms in the two object signatures (Figure 4.1(b)).

NB: *Transduction* is a process in which mapping an object o_1 to (potentially many) other objects o_2 is realized by first inverting one function f_1 to find the object's antecedents a and then applying to them another function f_2, i.e. an object o_1 is transduced to an object o_2 whenever there exists an a such that $f_1(a) = o_1$ and $f_2(a) = o_2$. In our example, terms from $\mathcal{O}(<\Sigma_1, \Sigma_2, \mathcal{L}_{12}, S>)$ are transduced to terms from $\mathcal{O}(<\Sigma_1, \Sigma_3, \mathcal{L}_{13}, S>)$ by inverting the lexicon $\mathcal{L}_{12}$ to find an antecedent in $\mathcal{A}(<\Sigma_1, \Sigma_2, \mathcal{L}_{12}, S>)$ and then applying to it the lexicon $\mathcal{L}_{13}$.

These two modes of composition can be fruitfully mixed. Consider the graph on Figure 4.2(a) in which we introduce a syntactic signature $Syntax$ where different terms are only allowed to yield the same string whenever they represent different parses of a syntactically ambiguous phrase. This means that purely semantic distinctions such as scope ambiguity have to be moved to a more abstract signature ($SyntSem$) from whose terms the semantic representation can be derived by a function ($\mathcal{L}_{interSem}$). This approach is studied more deeply in [23].

ACGs, we compose just the lexicons.

On top of that, we will want to define signatures and lexicons to yield multiple different object languages (at least one for the string representations of forms and one for the semantic representations of meanings), which would mean having at least two different ACGs which are somehow linked together (e.g. by sharing the same abstract signature). However, since these two grammars are so interrelated, we would like to reason about them in a way that makes these relationships explicit and thus easier to study as opposed to considering a collection of grammars which are somehow implicitly similar.

(a) Stacking the constraints vertically.

(b) Laying out the constraints side by side.

Figure 4.3: Constraining a syntactic signature with multiple constraints.

Another interesting composition pattern is to take an existing structure of grammars and add a new abstract signature on top (Figure 4.2(b)). This new signature ($Constr$) can be used to further constrain the items in the previously abstract-most signature ($Synt$). In [24], this is what the authors use to develop their constraints on extraction. Each of their constraints is presented as a new abstract signature constraining the original syntactic signature.

However, a question that the authors do not pose, but which is of great interest to us, is how to combine all these constraints. If we were to apply the same pattern repeatedly to get a structure like the one on Figure 4.3(a), we would have to deal with successively more complex type systems on each level. This is due to the condition that a lexicon has to be a homomorphism and thus it must, among other things, always map well-typed terms to well-typed terms. Because of this, the terms in the abstract language of the signature we introduce to handle the second constraint must also satisfy the first constraint, since mapping them by $\mathcal{L}_{constr_2}$ must yield a well-typed term in the signature corresponding to the first constraint. In the end, whenever we wish to add a new signature to handle another constraint, we have to acknowledge and reimplement in it all of the existing constraints. This effectively renders the pattern useless since we might just as well consider the final abstract-most signature handling all the constraints ($Constr_{1\wedge 2}$) and attach it directly to the syntactic signature ($Synt$) by taking the composition of the lexicons in between ($\mathcal{L}_{constr_1} \circ \mathcal{L}_{constr_2}$).

What we would like instead is to put the signatures side by side such that they do not depend on each other (Figure 4.3(b)). However, our language of syntactic structures is no longer defined by mapping a single abstract language through a lexicon. It is outside of the scope of the common formalization of ACGs that we presented in 4.1.1. How do we define this language in a manner that is consistent with our current definitions of ACGs? We can arrive at the answer by examining how the graphical representations of ACGs are connected to the languages they define.

4.1.3 Interpreting ACG Diagrams

We will associate a language to every node of an ACG diagram. Before we begin, we notice that every ACG diagram formed by the above modes of composition is an arborescence[2] and therefore every node has at most one predecessor (the parent). To the root of the tree, the abstract-most signature in the graph, we will assign the abstract language generated by that signature and its distinguished type. To all the other nodes, we will assign the result of mapping their parent's language by the lexicon linking the two. Thanks to the homomorphism property of the lexicons, we can find distinguished types for all the non-root nodes. They are simply the images of their parents' distinguished types by the connecting lexicons. For all of the nodes, it is true that every term in the language assigned to the node has the node's distinguished type.

We can confirm that the above graphical model of defining ACGs is coherent.

First, we can show that our interpretation is sound, meaning that any arborescence adorned with signatures on the nodes, compatible lexicons on the edges and a distinguished type on the root can only define ACG languages (abstract or object languages). This is trivial, since the language corresponding to the root is an ACG abstract language and the class of ACG languages is closed under transformation by a lexicon and therefore all of the languages corresponding to the descendants of the root are ACG languages as well.

Second, we can try and show some kind of completeness. However, there is not much formalization done to describe a "system of ACGs that share some common signatures" (something that our generalization will fix). We can nevertheless show that it is possible to express any ACG using our graphical model. Given a grammar $\mathcal{G} = <\Sigma_A, \Sigma_O, \mathcal{L}, S>$, we just produce a tree whose root has the signature Σ_A and distinguished type S and which is connected to a single child node having the signature Σ_O by an edge labelled by the lexicon $\mathcal{L}$. Then the two nodes of our tree define both the abstract and the object language defined by $\mathcal{G}$. Similarly, we could argue that the languages we aim to define using ACG diagrams (such as the ones on Figures 4.1 and 4.2) are covered by our assignment of languages to nodes of the diagrams.

We have now given a formal account of how we can read language definitions from ACG diagrams. Still, we have kept our analysis to the arborescent diagrams used in current ACG literature, where every node has at most one parent. We will now proceed to generalize it to cases where nodes have more than just one parent.

To make the generalization more visible, we will switch from an algebraic manner of presentation to a relational one. Before, we stated that the language of a node is the image of its parent's language.

$$Language(u) = Lexicon(Language(Parent(u)))$$

Now, we will restate this as saying that the language of a node is the set of terms that have an antecedent in the parent's language.

[2] An arborescence is a directed rooted tree, where every edge points away from the root.

$$Language(u) = \{x \in \Lambda(Signature(u)) \mid \exists a \in Language(Parent(u)),\ Lexicon(a) = x\}$$

This definition works for non-root nodes of an ACG diagram. In the root case, we have to constrain the language by stating that that the terms of the language must have the root's distinguished type. This is a property that is also true in the non-root cases due to the homomorphism property of lexicons. If we wanted to unify the definitions of the root language and the non-root languages, we would have to check that a term of the language has the distinguished type and that it has an antecedent in the parent, if it has any parent. A relational statement of this unification could look like this.

$$\begin{aligned} Language(u) = \{x \in \Lambda(Signature(u)) \mid\ & Type(x) = DistinguishedType(u) \\ & \wedge \forall p \in Parents(u),\ \exists a \in Language(p),\ Lexicon_{(p,u)}(a) = x\} \end{aligned}$$

If the node has no parents, no constraint is enforced. If it does have some, we enforce the antecedent constraint. This shows us a natural generalization to the case where nodes can have an arbitrary number of parents. If we transpose this back to the algebraic style of presentation, we get the following expression.

$$Language(u) = DistinguishedlyTyped(u) \quad \cap \bigcap_{p \in Parents(u)} Lexicon_{(p,u)}(Language(p))$$

4.2 Definitions

In this section, we distill the reasoning about generalizing ACG diagrams to graphs in a definition of an extension for ACGs.

We define a *graphical abstract categorial grammar* as a quadruple $\mathcal{G} = <G, \Sigma, S, \mathcal{L}>$ where G is a directed graph with vertices $V(G)$ and edges $E(G)$, Σ and S are labelings assigning signatures and distinguished types, respectively, to the vertices $V(G)$ and $\mathcal{L}$ is a labeling assigning lexicons to the edges $E(G)$. Furthermore, a well-formed graphical ACG satisfies the following conditions:

- G is a directed acyclic graph.
- For all $(u, v) \in E(G)$, $\mathcal{L}_{(u,v)} : \Sigma_u \to \Sigma_v$.
- For all $(u, v) \in E(G)$, $\mathcal{L}_{(u,v)}(S_u) = S_v$.

We will say that a node u is *more abstract* than a node v whenever (u, v) belongs to the strict partial order induced by the edges of G (i.e. there is a path from u to v).

We let the nodes in a graphical ACG $\mathcal{G} = <G, \Sigma, S, \mathcal{L}>$ define two kinds of languages. The *intrinsic languages* $\mathcal{I}_\mathcal{G}$, which are constrained only by the type signature described in the node itself, and the *extrinsic languages* $\mathcal{E}_\mathcal{G}$, which are

also constrained by type signatures in more abstract nodes of the graph. These two notions correspond to those of abstract languages and object languages in ACGs.

$$\mathcal{I}_{\mathcal{G}}(v) = \{t \in \Lambda(\Sigma_v) \mid \vdash_{\Sigma_v} t : S_v\}$$
$$\mathcal{E}_{\mathcal{G}}(v) = \mathcal{I}_{\mathcal{G}}(v) \cap \bigcap_{(u,v)\in E} \mathcal{L}_{(u,v)}(\mathcal{E}_{\mathcal{G}}(u))$$

From the above definition, we see that the ACG diagrams we introduced before are a special case of graphical ACGs where the graph is an arborescence.

4.3 General Remarks on Graphical ACGs

In this section, we examine the key properties and implications of graphical ACGs.

4.3.1 On the Expressivity of Graphical ACGs

Observation. *The set of intrinsic languages definable by graphical ACGs is by definition the same as the set of abstract languages definable by ACGs.*

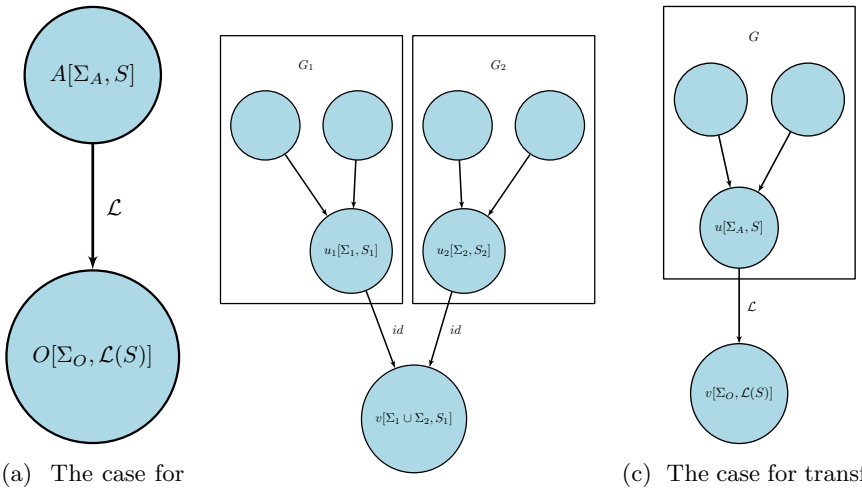
Theorem. *The set of extrinsic languages definable by graphical ACGs is the set of object languages definable by ACGs closed under intersection and transformation by a lexicon.*

Proof. First, we prove that an extrinsic language can be always constructed from object languages by intersections and transformations by lexicons. Let us consider any graphical ACG $\mathcal{G} = <G, \Sigma, S, \mathcal{L}>$ and some topological ordering $v_1, \ldots, v_n$ of its nodes $V(G)$ (an ordering such that more abstract nodes always precede less abstract ones). Before we start, we will remark that all the intrinsic languages $\mathcal{I}_{\mathcal{G}}(v_i)$ are object languages, since intrinsic languages are abstract languages which are in turn just a special case of object languages.

We proceed by induction on the topological ordering:

- For v_1, the case is trivial. v_1 has no predecessors and so $\mathcal{E}_{\mathcal{G}}(v_1) = \mathcal{I}_{\mathcal{G}}(v_1)$, which is an object language.
- For any other node v_n, we look at the definition of $\mathcal{E}_{\mathcal{G}}(v_n)$ and remark that the only operators are intersection and application of a lexicon with the operands being $\mathcal{I}_{\mathcal{G}}(v_n)$ (which is an object language) and the extrinsic languages of its predecessors (which can be constructed from object languages using intersections and lexicons thanks to the induction hypothesis). Therefore, even $\mathcal{E}_{\mathcal{G}}(v_n)$ can be constructed from object languages using only intersections and lexicons.

Now we have to prove the converse. We do so by showing that extrinsic languages contain object languages and are closed on intersection and transformation by a lexicon.



(a) The case for object languages.

(b) The case for intersections.

(c) The case for transformations by a lexicon.

Figure 4.4: Figures demonstrating that extrinsic languages contain object languages closed on transformation by a lexicon and intersection.

- Let L be an object language defined by the ACG $\mathcal{G} = <\Sigma_A, \Sigma_O, \mathcal{L}, S>$. We take the graph G with $V(G) = \{A, O\}$ and $E(G) = \{(A, O)\}$, we label the nodes A and O with Σ_A and Σ_O as signatures and S and $\mathcal{L}(S)$ as distinguished types respectively . Finally, we label the edge (A, O) with $\mathcal{L}$ and we get a graphical ACG G' in which $\mathcal{E}_{\mathcal{G}'}(O) = \mathcal{O}(G) = L$ (Figure 4.4(a)).

- Let L_1 and L_2 be two extrinsic languages defined by the nodes u_1 and u_2 in some graphical ACGs $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. We can construct a new graphical ACG $\mathcal{G}'$ by taking the union of the two ACGs (union of graphs (union of vertices and edges) and unions of the labelings).[3] We will then add a new node v such that $\mathcal{E}_{\mathcal{G}'}(v) = L_1 \cap L_2$ (Figure 4.4(b)).

  The signature labeling v will be the union of the two signatures labeling u_1 and u_2, ensuring that $\forall u \in \{u_1, u_2\}$, $\mathcal{I}_{\mathcal{G}'}(v) \supseteq \mathcal{I}_{\mathcal{G}'}(u) \supseteq \mathcal{E}_{\mathcal{G}'}(u)$. We can use either of the two distinguished types that label u_1 or u_2 to label v (if they are different, then the intersection $L_1 \cap L_2$ is trivially empty). Finally, we will add two edges leading from u_1 and u_2 to v, both labeled with the identity lexicon. We will now show that $\mathcal{E}_{\mathcal{G}'}(v) = L_1 \cap L_2$.

[3] Assuming, without loss of generality, that the sets of vertices of the two graphs are disjoint.

$$\begin{aligned}\mathcal{E}_{\mathcal{G}'}(v) &= \mathcal{I}_{\mathcal{G}'}(v) \cap \mathcal{L}_{(u_1,v)}(\mathcal{E}_{\mathcal{G}'}(u_1)) \cap \mathcal{L}_{(u_2,v)}(\mathcal{E}_{\mathcal{G}'}(u_2)) \\ &= \mathcal{I}_{\mathcal{G}'}(v) \cap \mathcal{E}_{\mathcal{G}'}(u_1) \cap \mathcal{E}_{\mathcal{G}'}(u_2) \\ &= \mathcal{E}_{\mathcal{G}'}(u_1) \cap \mathcal{E}_{\mathcal{G}'}(u_2) \\ &= \mathcal{E}_{\mathcal{G}_1}(u_1) \cap \mathcal{E}_{\mathcal{G}_2}(u_2) \\ &= L_1 \cap L_2\end{aligned}$$

The simplifications use the facts that: the two new lexicons are identities, that $\mathcal{I}_{\mathcal{G}'}(v) \supseteq \mathcal{E}_{\mathcal{G}'}(u_1) \cap \mathcal{E}_{\mathcal{G}'}(u_2)$, that $\mathcal{E}_{\mathcal{G}'}(u_1) = \mathcal{E}_{\mathcal{G}_1}(u_1)$ and $\mathcal{E}_{\mathcal{G}'}(u_2) = \mathcal{E}_{\mathcal{G}_2}(u_2)$ and finally that $\mathcal{E}_{\mathcal{G}}(u_1) = L_1$ and $\mathcal{E}_{\mathcal{G}}(u_2) = L_2$.

- Let $L \subseteq \Lambda(\Sigma_O)$ be a language which is the result of mapping an extrinsic language $L' \subseteq \Lambda(\Sigma_A)$ through the lexicon $\mathcal{L} : \Sigma_A \to \Sigma_O$. We will show that L is also an extrinsic language. Let u be the node that defines the extrinsic language L in some graphical ACG $\mathcal{G}$. We construct a new graphical ACG $\mathcal{G}'$ by adding a new vertex v to $\mathcal{G}$ labeled with the signature Σ_O and the distinguished type $\mathcal{L}(S)$ where S is the distinguished type labeling u. To this graph, we conjoin the edge (u, v) and label it with $\mathcal{L}$ (Figure 4.4(c)).

  Now we verify that $\mathcal{E}_{\mathcal{G}'}(v) = L$.

$$\begin{aligned}\mathcal{E}_{\mathcal{G}'}(v) &= \mathcal{I}_{\mathcal{G}'}(v) \cap \mathcal{L}(\mathcal{E}_{\mathcal{G}'}(u)) \\ &= \mathcal{L}(\mathcal{E}_{\mathcal{G}'}(u)) \\ &= \mathcal{L}(\mathcal{E}_{\mathcal{G}}(u)) \\ &= \mathcal{L}(L') \\ &= L\end{aligned}$$

  The assumptions used here are in turn: that $\mathcal{I}_{\mathcal{G}'}(v) \supseteq \mathcal{L}(\mathcal{E}_{\mathcal{G}'}(u))$, that $\mathcal{E}_{\mathcal{G}'}(u) = \mathcal{E}_{\mathcal{G}}(u)$, that $\mathcal{E}_{\mathcal{G}}(u) = L'$ and that $\mathcal{L}(L') = L$.

□

Corollary. *The set of extrinsic languages definable by graphical ACGs is the same as the set of object languages definable by ACGs if and only if the set of object languages definable by ACGs is closed on intersection.*

We conclude this subsection with a few words to motivate the need we feel for this (possibly) more expressive formalism. We believe that in order to manage the development of wide-coverage grammars, the grammars need to be kept as simple as possible, which means, among other things, that independent constraints should be implemented separately and not complected together (we saw the difficulties caused by this already on a microscopic fragment in section 3.4). This means that if we were to define, e.g., a partial grammar in which we would

only enforce agreement and another grammar in which we would only enforce the proper use of relative pronouns, we would like the language in which both agreement and proper use of relative pronouns are respected to be expressible in our formalism, and on top of that, we would like this language to be easily expressed in terms of the two grammars for the individual constraints. We now have a requirement on the formalism in which we would like to use: we want a formalism where the set of all the "useful" definable languages is closed on intersection. If the object languages of ACGs are indeed closed on intersection, then our extension is just a convenience for expressing intersection more easily. However, if they are not, we believe our extension adds a critical level of expressivity needed for simple and modular wide-coverage grammars.

4.3.2 On the Decidability of Graphical ACGs

We now turn to the problem of parsing with graphical ACGs. Parsing with ACGs is a problem on the limits of decidability. When we consider ACGs whose abstract signatures only contain types of order 2 or less, parsing is possible in polynomial time for both linear [26] and non-linear [15] type systems (such as the one we introduced in 1.3). If the order of the abstract signature exceeds 2, the membership problem for linear ACGs becomes decidable if and only if the satisfiability problem in the multiplicative exponential fragment of linear logic is decidable [8], which is an open problem. On the other hand, by extending the type system to include dependent types, parsing can become undecidable too since in such a system type checking itself is undecidable.

Our intent here is to show that by extending the formalism we have not made the problem of parsing more difficult, at least in terms of decidability if not in terms of complexity. Let us therefore imagine a graphical ACG where every edge (two signatures, distinguished types and a lexicon) is an ACG for which there is an effective parsing algorithm that lets us find all the abstract terms for a given object term in finite time and where type checking is decidable in all the signatures of the graphical ACG. Then we can prove the existence of a simple effective parsing algorithm for any of the extrinsic languages defined in the graphical ACG.

We will be testing whether the term o belongs to $\mathcal{E}_{\mathcal{G}}(v)$. Our algorithm first checks whether o belongs to $\mathcal{I}_{\mathcal{G}}(v)$ by checking its type. If it does, it proceeds to verify that it is also in $\mathcal{E}_{\mathcal{G}}(v)$ by finding antecedents in all of its abstract predecessors u_i, if there are any. We can use the presupposed parsing algorithm for the ACG $<\Sigma_{u_i}, \Sigma_v, \mathcal{L}_{(u_i,v)}, S_{u_i}>$ to find all the abstract terms $\alpha_{i,j}$ having type S_{u_i} and being antecedents to o w.r.t. the lexicon $\mathcal{L}_{(u_i,v)}$. Such terms belong to $\mathcal{I}_{\mathcal{G}}(u_i)$, but in order to prove that o is in $\mathcal{E}_{\mathcal{G}}(v)$, we need to find an antecedent that belongs to $\mathcal{E}_{\mathcal{G}}(u_i)$. What we do is we recursively apply our parsing algorithm to all of the antecedent-candidates $\{\alpha_{i,j} \mid \forall j\}$ until we find one which belongs $\mathcal{E}_{\mathcal{G}}(u_i)$. Since the graph is acyclic, we are guaranteed that this recursive traversal will terminate. We repeat this for all the predecessors u_i and if we succeed to find antecedents in all of their extrinsic languages, we have confirmed the membership of o in $\mathcal{E}_{\mathcal{G}}(v)$ and also built its abstract antecedent(s).

4.3.3 On Grammar Engineering with Graphical ACGs

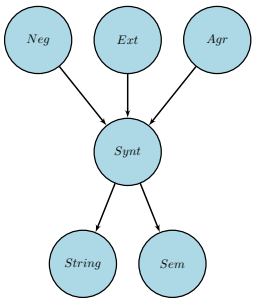
Having covered some of the formal properties of graphical ACGs, we will get back to building grammars. Armed with this formalism, we can solve the puzzle of incorporating the three constraints of sections 3.1, 3.2 and 3.3. Our graph G will consist of 6 nodes, $V(G) = \{Neg, Ext, Agr, Synt, Str, Sem\}$ with edges going from Neg, Ext and Agr to $Synt$ and from $Synt$ to Str and Sem (see Figure 4.5(a)). The signatures and distinguished types for the first three vertices are exactly those that have been presented in the first three sections of Chapter 3. The lexicons for the edges connecting these constraint nodes to $Synt$ all behave according to the scheme below:

$$\mathcal{L}_{(constraint,Synt)}(X_{constant_{index}}) = C_{constant}$$
$$\mathcal{L}_{(constraint,Synt)}(TYPE\_FEATURES) = TYPE$$

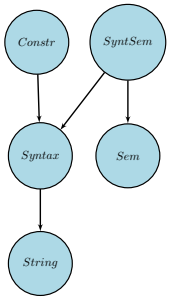
The $\{Synt, Str, Sem\}$ subgraph is identical to the pair of ACGs described in 1.3 except for some additions that need to be made so that the fragment covers the wordforms that we have introduced in the course of discussing the three constraints (prepositions, relative pronouns, the negative particle *ne*...). However, the types of these new constants will not be tied to any of those constraints.

$$\begin{aligned}
C_{de} &: NP \multimap N \multimap N \\
C_{qui} &: (NP \multimap S) \multimap N \multimap N \\
C_{dit\ que} &: NP \multimap S \multimap S \\
C_{dorment} &: NP \multimap S \\
C_{ne_{tv}} &: (NP \multimap NP \multimap S) \multimap (NP \multimap NP \multimap S) \\
C_{ne_{iv}} &: (NP \multimap S) \multimap (NP \multimap S) \\
C_{ne_{cv}} &: (NP \multimap S \multimap S) \multimap (NP \multimap S \multimap S)
\end{aligned}$$

This approach lets us describe the high-level combinatorics of language that are of interest to the categorial grammarian in a manner which is idiomatic in academic literature (using the simple and focused types N, NP and S). At the same time, we can actually use this textbook grammar as a basis for a more realistic grammar which handles constraints such as agreement and extraction islands. Furthermore, these additional constraints are also defined in a manner that has its precedent in existing research. When the authors of [24] decided to put the constraints of extraction under a microscope and try to encode them in ACGs, they defined their constraints in exactly the same way as our architecture does. This graphical architecture lets us write the syntactic kernel and the additional constraints independently and using two different styles that have both been shown useful by their own merits, which we see as indicative of enabling good modularity.



(a) The graph of a grammar enforcing all of the constraints discussed in Chapter 3.



(b) Combining the two ACG patterns from 4.1.2 in a single graphical ACG.

Figure 4.5: Practical architectures for graphical ACGs.

In the above approach, we have defined a language by writing a grammar which provides us with useful structures but which does not enforce complete correctness. We have then refined this grammar by tacking on constraints "on the side". This approach is generally useful and can be seen, e.g., in [27] to define the formal programming language kernel on which a formal semantics is established. In our case, we are interested in having intelligible syntactic structures too as we would like to compute their semantic representations. The syntax-semantics interface already presents a significant challenge and having less-than-ideal syntactic structures to start with does not make formalizing it any easier.

We will show one more persuasive scenario whose goal is to show the importance of graphical ACGs for having a clean syntax-semantics interface. In 4.1.2, we have seen an ACG pattern that put semantic ambiguities into an abstract signature that is higher than syntax. But there was also another pattern, that of adding a new abstract signature to constrain syntactic terms. We can combine both of them in graphical ACGs by superposing their two graphs to get the structure we can see on Figure 4.5(b).

If we try to reap the benefits of both patterns and try to replicate the same in classical ACGs (arborescent graphs), we hit a snag: we cannot have both the $Constr$ and $SyntSem$ nodes as the direct predecessors of the $Syntax$ node. We can make $Constr$ be the predecessor of $Syntax$, but then $SyntSem$ will be connecting $Constr$ and Sem and the syntax-semantics interface will have to deal with the types which were needed to implement syntactic constraints and have no place in a syntax-semantics interface. If we try it the other way, it is not much better. $Constr$ will end up having to work with the more complicated type system of $SyntSem$ instead of being able to talk only about the $Syntax$ it is interested in.

By putting the constraining signature in a part of the graph unreachable from the syntax-semantics transducer, we can shield the syntax-semantics interface from the gnarly implementation details of the rules of syntax. This

in turn gives us a guideline as to what information should be expressed in the syntactic signature itself and which should be hidden in constraints: the syntactic signature should only contain information relevant to the syntax-semantics interface.

This might then raise a question whether the chosen syntactic signature will end up being suitable for expressing all of the syntactic constraints. However, this can be resolved by deepening the constraint subgraph and adding a "compatibility layer" in the form of a new signature which provides different syntactic structures and maps them to the canonical ones. Constraints can then opt in to controlling the syntactic structures offered by this new signature.

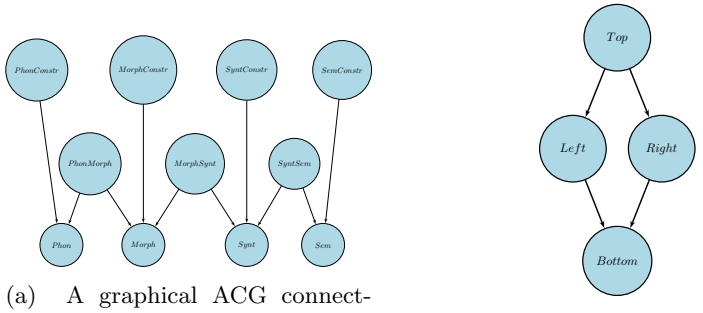
4.4 On Alternative Interpretations of Graphical ACGs

Our definitions of intrinsic and extrinsic languages have consequences which might be counter-intuitive and not correspond to the grammarian's intent. We will describe what these consequences are and propose an alternative notion of a graphical ACG language which aims to reconcile them.

The first phenomenon that deserves pointing out occurs whenever two paths from nodes u_1 and u_2 converge on a single node v. If we then take a term α from $\mathcal{E}_\mathcal{G}(u_1)$, the term $\beta = \mathcal{L}_{(u_1,v)}(\alpha)$ belongs to $\mathcal{I}_\mathcal{G}(v)$ but not necessarily to $\mathcal{E}_\mathcal{G}(v)$ since it might be the case that there is no antecedent for β in $\mathcal{E}_\mathcal{G}(u_2)$. In other words, mapping a member of $\mathcal{E}_\mathcal{G}(u)$ using the lexicon $\mathcal{L}_{(u,v)}$ does not guarantee that the result belongs to $\mathcal{E}_\mathcal{G}(v)$, which is a different behavior from that of object languages in ACGs (or that of extrinsic languages in arborescent graphical ACGs).

This highlights some limitations of our notion of an extrinsic language. Consider once more the graphical ACG of Figure 4.5(b). We have more than one path converging to $Syntax$. We will consider a term α of $\mathcal{E}_\mathcal{G}(Sem)$. α has an antecedent in $\mathcal{E}_\mathcal{G}(SyntSem)$ which has a descendant β in $\mathcal{I}_\mathcal{G}(Syntax)$. However, because of the constraint $Constr$ on $Syntax$, β can be a syntactic term outside of $\mathcal{E}_\mathcal{G}(Syntax)$ and yielding an ungrammatical sentence. The notion of an extrinsic language is insufficient to express the set of meanings from $\Lambda(\Sigma_{Sem})$ which are expressible only by grammatical sentences. Defining languages like this would be of particular interest in cases like the one in Figure 4.6(a) where we have a chain of signatures representing the different levels of linguistic description whose terms are in many-to-many relationships expressed through transducers. In such a scenario, it would be desirable to be able to express the set of descriptions belonging to one level, e.g. morphology, that correspond to valid descriptions on all the other levels.

Compared to ACG object languages (or to extrinsic languages of arborescent graphical ACGs), we lose another guarantee. In arborescences, the central role of the root abstract-most signature guarantees us that whenever a term belongs to an extrinsic language, we can find terms in all the other extrinsic languages that share the same abstract-most term. On the other hand, in non-arborescent



(a) A graphical ACG connecting (constrained) levels of linguistic description through transducers.

(b) A graphical ACG demonstrating two different paths between two nodes.

Figure 4.6: Graphical ACGs posing challenges to extrinsic languages.

graphical ACGs, having a term in an extrinsic language of some node v only guarantees the presence of corresponding terms in the extrinsic languages of its ancestors and the intrinsic languages of their descendants.

The second puzzling scenario is the diamond shape on Figure 4.6(b). If we consider a term δ of $\mathcal{E}_{\mathcal{G}}(Bottom)$, we know that there exist antecedents β and γ in $\mathcal{E}_{\mathcal{G}}(Left)$ and $\mathcal{E}_{\mathcal{G}}(Right)$. And while there definitely exist antecedents α and α' to both β and γ in $\mathcal{E}_{\mathcal{G}}(Top)$, these antecedents can be different, meaning there does not have to be a single $\alpha \in \mathcal{E}_{\mathcal{G}}(Top)$ which yields $\beta \in \mathcal{E}_{\mathcal{G}}(Left)$ and $\gamma \in \mathcal{E}_{\mathcal{G}}(Right)$ such that they both yield δ. This means that the result of parsing a term in some $\mathcal{E}_{\mathcal{G}}(v)$ is a collection of abstract terms with one term not just per every node u that is more abstract than v but per every different path leading to v. Parsing a term $\delta \in \mathcal{E}_{\mathcal{G}}(Bottom)$ in our diamond grammar would thus produce four antecedents β, γ, α and α' corresponding to the four paths $[Left, Bottom]$, $[Right, Bottom]$, $[Top, Left, Bottom]$ and $[Top, Right, Bottom]$, respectively. We contrast this again with the case of classical ACGs where the result of parsing can always be represented by associating a term to every node of the arborescence.[4]

We will propose a new way of assigning languages to nodes in a graphical ACG that will conserve the intuitive properties of ACGs we talked about above. We do not need to modify the definition of a graphical ACG in order to do so since the definition is only concerned with the structure of the graph and its decoration with ACG paraphernalia, not the languages defined. We will define a third kind of languages based on graphical ACGs.

Let $\mathcal{G} = <G, \Sigma, S, \mathcal{L}>$ be a graphical ACG. We define the *pangraphical language* of node u, $\mathcal{P}_{\mathcal{G}}(u)$, to be the set of terms t such that there exists a labeling

[4]One could also say that the result of parsing in an ACG is just the antecedent in the abstract-most signature and that the other terms can then be easily computed by application of lexicons. Similarly, we could say that the result of parsing in a general graphical ACG is a collection of abstract terms for every *maximal* path leading to the node defining the language in question. What might be more intuitive, though, is to have some theory of graphical ACGs where the result of parsing is a collection of antecedents in all the abstract-most nodes.

of nodes with terms T satisfying the following conditions:

- $T_u = t$.
- For all $v \in V(G)$, $\vdash_{\Sigma_v} T_v : S_v$.
- For all $(v, w) \in E(G)$, $\mathcal{L}_{(v,w)}(T_v) = T_w$.

It is easy to see that the newly introduced pangraphical languages solve all of the counter-intuitive quirks of extrinsic languages that we discussed in this subsection.

- Applying the lexicon $\mathcal{L}_{(u,v)}$ to an element of $\mathcal{P}_\mathcal{G}(u)$ always yields an element of $\mathcal{P}_\mathcal{G}(v)$, since the labeling of nodes with terms T that proved the former term's membership in the former language also proves the latter term's membership in the latter language.

  This definition allows us to directly establish the language of meanings which are expressible only by grammatical sentences in Figure 4.5(b) and the language of morphological terms that have valid corresponding representations on all the other levels of linguistic description in Figure 4.6(a).

- The diamond grammar of Figure 4.6(b) is also no longer problematic since the labeling of nodes with terms guarantee that there is always a single term per node that can generate the term being parsed.

In the previous section, we took a stab at formalizing ACG diagrams and finding the simplest extension capable of handling non-arborescent graphs. The notion of extrinsic languages that we arrived at allowed us to define a nicely delimited set of languages, the object languages of ACGs with intersection. Discrepancies in the behavior of extrinsic languages in arborescent and non-arborescent graphs led us to pangraphical languages, which behave more in line with our intuitions. However, the expressivity of pangraphical languages, especially w.r.t. to extrinsic languages, still remains to be explored. Due to limitations of both space and time, we will cover the interesting properties of pangraphical languages only briefly and partially here.

Extrinsic languages can be defined similarly to pangraphical languages

In the definition of $\mathcal{P}_\mathcal{G}(u)$, we replace the labeling of nodes with terms by a labeling of all the paths leading to u.

The three languages are increasingly constraining: $\mathcal{P}_\mathcal{G}(u) \subseteq \mathcal{E}_\mathcal{G}(u) \subseteq \mathcal{I}_\mathcal{G}(u)$

Given a labeling of nodes with terms such that it satisfies the conditions in the definition of a pangraphical language, we can show by induction on the topological ordering of the graph that every term assigned by the

labeling belongs to the extrinsic language of the same node. Therefore, $\alpha \in \mathcal{P}_{\mathcal{G}}(u) \implies \alpha \in \mathcal{E}_{\mathcal{G}}(u)$.

$\mathcal{E}_{\mathcal{G}}(u) \subseteq \mathcal{I}_{\mathcal{G}}(u)$ comes straight from the definition of $\mathcal{E}_{\mathcal{G}}(u)$.

We have thus came up with a series of successively more constrained languages: intrinsic languages are constrained by one node, extrinsic languages are also constrained by its ancestors and pangraphical languages are constrained by the entire graph.

On arborescences, $\mathcal{P}_{\mathcal{G}}(u) = \mathcal{E}_{\mathcal{G}}(u)$.

On an arborescence, being a member of an extrinsic language means there exists an antecedent in the abstract-most language of the arborescence. By virtue of the one-parent property of arborescences, we can simply apply all the lexicons in the arborescence to this abstract-most antecedent to find a labeling of nodes with terms which proves that the member of the extrinsic language is also a member of the pangraphical language.

This confirms that both extrinsic languages and pangraphical languages are reasonable generalizations of ACG diagrams since their behaviors on arborescences are consistent with each other and with that of ACG object languages.

The set of extrinsic languages is included in the set of pangraphical languages.

Let us have some node u in a graphical ACG $\mathcal{G}$. We build a graphical ACG $\mathcal{G}'$ where every node corresponds to a path in $\mathcal{G}$ which ends at u and two nodes are connected with an edge whenever one path is an immediate extension of the other. We then have $\mathcal{P}_{\mathcal{G}'}([u]) = \mathcal{E}_{\mathcal{G}}(u)$.

Is the set of pangraphical languages included in the set of extrinsic languages?

That is an interesting question indeed, one whose resolution is left as a possible avenue for future work.

Chapter 5

In Conclusion

In this final chapter, we first discuss some the consequences of using G-ACGs for writing grammars. We then summarize the contributions that we have put forward and suggest problems we believe to be interesting.

5.1 On the Practical Consequences of Graphical ACGs

We will enumerate some of the practical benefits and challenges of using graphical ACGs for implementing grammar-based systems.

- The lexicons that go from our constraint signatures (Neg, Ext and Agr in subsection 4.3.3) are all trivial in the sense they map constants to constants. Finding an antecedent to a term is then just a question of finding an antecedent for every constant in the term (i.e. "tagging" the constants of the term with antecedent constants). This means that there is an effective parsing algorithm for verifying the constraints. Furthermore, this algorithm could also be made efficient by introducing filtering techniques [12] and statistical supertagging [18].

- Constraints can introduce type extensions in their abstract signatures and the added complexity of these extensions is then contained within the constraint. Furthermore, the constraint signature can be treated as a "black box" that verifies a constraint and in the context of an implementation, it can be replaced with a special-purpose hand-written procedure that does the same more efficiently.

- The distributed nature of graphical ACGs has its own practical applications. Parsing can be made more robust by being able to parse sentences which violate some of the constraints. Parsing an ungrammatical sentence and identifying the constraints it violates can also serve as a basis for a syntax checker.

- If we decide to define all of our syntactic constraints in separate signatures, we will end up with a lot of signatures which look exactly like the syntactic signature but with some small refinements (see the signatures in sections 3.1, 3.2 and 3.3). Thus we will desire some concise way of formulating these refinements so that we will not need to replicate the entire syntactic signature for every constraint.

  We hope that the class of useful refinements is small and that these constraint signatures could be systematically constructed from the syntactic signature by applying a series of refining transformations. This is an interesting engineering challenge awaiting anyone willing to design a truly wide-coverage graphical ACG using constraints.

5.2 Conclusion

The contributions of our work can be summarized in the following three points:

1. We have highlighted the challenges inherent in translating an existing grammar in the formalism of IGs to the formalism of ACGs. Based on the breadth of different constraints built in to the IG formalism, we have conjectured that a direct translation of IG lexical items to ACG lexical items based on the deep underlying similarity between the two formalisms is not practical.
2. We have implemented a system for experimenting with (lexicalized) abstract categorial grammars that can serve as the basis for formulating wide-coverage graphical ACGs (`https://github.com/jirkamarsik/acg-clj`).
3. We have presented the graphical abstract categorial grammars (G-ACGs) which generalize ACGs. We have determined the set of languages definable by G-ACGs in terms of ACG languages and we have shown how G-ACGs allow us to write categorial grammars that handle multiple disparate linguistic constraints without sacrificing simplicity.

Here are some of the items that we believe warrant future investigation:

- Try to express the multiple independent polarized features of IGs using G-ACGs.
- Determine whether ACG object languages are closed on intersection or not.
- Make a closer examination of the relationship between extrinsic and pangraphical languages in G-ACGs.
- Write grammars using the G-ACG constraint architecture to verify its applicability and test the claims of modularity we have made here.
- Study the problem of concisely defining constraint signatures from syntactic signatures and rules of feature propagation.

Chapter 6

Bibliography

[1] S. Afantenos, N. Asher, F. Benamara, M. Bras, C. Fabre, M. Ho-dac, A. Le Draoulec, P. Muller, M.P. Péry-Woodley, L. Prévot, et al. An empirical resource for discovering cognitive principles of discourse organisation: the annodis corpus.

[2] N. Asher and S. Pogodalla. A montagovian treatment of modal subordination. In *Proceedings of SALT*, volume 20, pages 387–405, 2011.

[3] N. Asher, S. Pogodalla, et al. Sdrt and continuation semantics. In *New Frontiers in Artificial Intelligence JSAI-isAI 2010 Workshops, LENLS, JURISIN, AMBN, ISS, Tokyo, Japan, November 18-19, 2010, Revised Selected Papers*, volume 6797, pages 3–15, 2011.

[4] Nicholas Asher and Alex Lascarides. *Logics of conversation*. Cambridge University Press, 2003.

[5] William E Byrd. Relational programming in minikanren: techniques, applications, and implementations. 2010.

[6] P. De Groote. Towards abstract categorial grammars. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 252–259. Association for Computational Linguistics, 2001.

[7] P. De Groote. Towards a montagovian account of dynamics. In *Proceedings of SALT*, volume 16, pages 1–16, 2006.

[8] Philippe De Groote, Bruno Guillaume, and Sylvain Salvati. Vector addition tree automata. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 64–73. IEEE, 2004.

[9] Philippe de Groote and Sarah Maarek. Type-theoretic extensions of abstract categorial grammars. *ESSLLI 2007*, page 19, 2007.

[10] Philippe De Groote, Sarah Maarek, and Ryo Yoshinaka. On two extensions of abstract categorial grammars. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 273–287. Springer, 2007.

[11] Denys Duchier, Yannick Parmentier, Simon Petitjean, et al. Metagrammars as logic programs. In *7th International Conference on Logical Aspects of Computational Linguistics (LACL 2012)*, pages 1–4, 2012.

[12] Bruno Guillaume, Joseph Le Roux, Jonathan Marchand, Guy Perrier, Karën Fort, and Jennifer Planul. A toolchain for grammarians. In *22nd International Conference on on Computational Linguistics: Demonstration Papers*, pages 153–156. Association for Computational Linguistics, 2008.

[13] Bruno Guillaume and Guy Perrier. Interaction grammars. *Research on Language and Computation*, 7(2-4):171–208, 2009.

[14] Hans Kamp and Uwe Reyle. *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory*. Number 42. Kluwer Academic Pub, 1993.

[15] Makoto Kanazawa. Parsing and generation as datalog queries. In *ANNUAL MEETING-ASSOCIATION FOR COMPUTATIONAL LINGUISTICS*, volume 45, page 176, 2007.

[16] Ronald M Kaplan and Joan Bresnan. Lexical-functional grammar: A formal system for grammatical representation. *Formal Issues in Lexical-Functional Grammar*, pages 29–130, 1982.

[17] R. Montague. The proper treatment of quantification in ordinary english.

[18] Richard Moot. Categorial grammars and wide-coverage semantics with grail. *LACL 2012*, page 9, 2012.

[19] Joseph P Near, William E Byrd, and Daniel P Friedman. α lean tap: A declarative theorem prover for first-order classical logic. In *Logic Programming*, pages 238–252. Springer, 2008.

[20] G. Perrier. From intuitionistic proof nets to interaction grammars. 1999.

[21] Guy Perrier. Intuitionistic multiplicative proof nets as models of directed acyclic graph descriptions. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 233–248. Springer, 2001.

[22] Guy Perrier. A french interaction grammar. In *International Conference on Recent Advances in Natural Language Processing-RANLP 2007*, pages 463–467, 2007.

[23] Sylvain Pogodalla. Generalizing a proof-theoretic account of scope ambiguity. In *7th International Workshop on Computational Semantics-IWCS-7*, 2007.

[24] Sylvain Pogodalla and Florent Pompigne. Controlling extraction in abstract categorial grammars. In *Formal Grammar*, pages 162–177. Springer, 2012.

[25] Sai Qian and Maxime Amblard. Event in compositional dynamic semantics. In *Logical Aspects of Computational Linguistics*, pages 219–234. Springer, 2011.

[26] Sylvain Salvati. Problemes de filtrage et problemes d'analyse pour les grammaires catégorielles abstraites. *These de Doctorat. Institut National Polytechnique de Lorraine*, 2005.

[27] Peter Van Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[28] Ryo Yoshinaka and Makoto Kanazawa. The complexity and generative capacity of lexicalized abstract categorial grammars. In *Logical Aspects of Computational Linguistics*, pages 330–346. Springer, 2005.

Résumé

Le but final de notre travail est la création d'une grammaire catégorielle abstraite (ACG) à large échelle qu'on pourra utiliser pour construire les représentations au niveau du discours de manière automatique.

Nous commençons par examiner les ressources linguistiques qui existent déjà, en particulier la grammaire d'interaction Frigram et son lexique Frilex. Nous examinons ensuite leur utilité pour la construction d'une ACG à large échelle. Puis, nous présentons une implémentation des mécanismes ACG qui nous permet d'expérimenter des grammaires lexicalisées à partir de Frilex. Enfin, nous considérons les problèmes posés par le traitement de plusieurs contraintes linguistiques dans une unique ACG et nous proposons une généralisation du formalisme : les grammaires catégorielles abstraites graphiques.

Mots-clés

Grammaires catégorielles abstraites (ACG), Développement de grammaires, Grammaires d'interaction, Metagrammaires, Programmation relationnelle, Formalismes grammaticaux, Grammaires formelles, Linguistique computationnelle, Lambda-calcul, Théorie des types

Summary

We present work whose ultimate goal is the creation of a wide-coverage abstract categorial grammar (ACG) that could be used to automatically build discourse-level representations.

We first examine existing language resources, in particular the Frigram interaction grammar and its lexicon Frilex, and assess their utility to building a wide-coverage ACG. We then present our implementation of the ACG machinery which allows us to experiment with grammars lexicalized by Frilex. Finally, we consider the challenge of integrating the treatment of disparate linguistic constraints in a single ACG and propose a generalization of the formalism: graphical abstract categorial grammars.

Keywords

Abstract Categorial Grammars (ACG), Grammar Engineering, Interaction Grammars, Metagrammars, Relational Programming, Grammatical Formalisms, Formal Grammars, Computational Linguistics, Lambda Calculus, Type Theory