Extending Abstract Categorial Grammars with Feature Structures: Theory and Practice

Philippe de Groote, Maxime Guillaume, Agathe Helman, Sylvain Pogodalla, Raphaël Salmon

Extending Abstract Categorial Grammars with Feature Structures: Theory and Practice

Philippe de Groote[1], Maxime Guillaume[1,2], Agathe Helman[2], Sylvain Pogodalla[1], and Raphaël Salmon[2]

[1] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France,
[2] Yseop

Abstract. Abstract Categorial Grammars offer a versatile framework for modeling natural language syntax and semantics. However, they currently miss a key component in grammatical engineering: feature structures. This paper introduces a conservative extension to the framework, incorporating feature structures and shows the practical advantages of this integration on a French grammar.

Keywords: Text generation · Industrial grammars · Categorial grammars· Feature structures.

1 Introduction

1.1 Abstract Categorial Grammar

Abstract Categorial Grammars (ACGs) derive from the tradition of type-theoretic grammars. They were developed with the intent of being a kernel for a grammatical framework [15] capable of representing various grammatical formalisms [5, 6]. This includes, but is not limited to, well-known formalisms such as context-free grammars (CFGs) and tree-adjoining grammars [9].

ACGs are based on a small set of computational primitives: typed linear λ-calculus and high-order signatures.

Definition 1 (Implicative Types). *Let A be a set of atomic types, the set of implicative types over A, $\mathcal{T}(A)$ is defined by the following BNF grammar:*

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

Definition 2 (High-order Signature). *A high-order signature Σ is a triplet $\langle A, C, \tau \rangle$ where:*

- *A is a finite set of atomic types;*
- *C is a finite set of constants;*
- *$\tau : C \to \mathcal{T}(A)$ is a function that associates to each constant $c \in C$ an implicative type in $\mathcal{T}(A)$.*

Definition 3 (Terms). *Let $\mathcal{X}$ be a countable set of variables, the set of lambda-terms defined over a signature Σ, $\Lambda(\Sigma)$, is defined by the following BNF grammar:*

$$\Lambda(\Sigma) ::= c \mid x \mid \lambda^{\circ} x.\Lambda(\Sigma) \mid \Lambda(\Sigma)\ \Lambda(\Sigma)$$

The variable x needs to appear free exactly once in the expression $\Lambda(\Sigma)$ for the linear lambda abstraction $\lambda^{\circ} x.\Lambda(\Sigma)$ to be correctly formed. We adhere to the standard conventions, employing left associativity for applications and right associativity for abstractions and implications.

Paralleling traditional compiler theory, ACG offers a clear cut between the abstract syntax and the object syntax. Both of these syntaxes are defined through high-order signatures. The connection between the two is established through the lexicon, which functions essentially as an interpreter, translating abstract structures into object ones. Fundamentally, the lexicon can be conceptualized as an abstract embodiment of the compositionality principle.

Definition 4 (Lexicon). *Let $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ be two high-order signatures, a lexicon from Σ_1 to Σ_2, $\mathcal{L} : \Sigma_1 \to \Sigma_2$ is a pair $\langle F, G \rangle$ where*

- *$F : A_1 \to \mathcal{T}(A_2)$ is a morphism that interprets atomic types of Σ_1 into implicative types over Σ_2;*
- *$G : C_1 \to \Lambda(\Sigma_2)$ is a morphism that associates to each constant of Σ_1, a lambda-term defined over Σ_2.*
- *$\hat{F}$ and $\hat{G}$ are the unique homomorphic extensions of F and G to types and terms.*
- *for all constants $c \in C_1$, the following judgment is derivable:*

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

To simplify notations, we will use $\mathcal{L}$ instead of F or G. With the previously discussed elements, we can now define an ACG.

Definition 5 (Grammar). *An ACG $\mathcal{G}$ is a quadruplet $\langle \Sigma_1, \Sigma_2, \mathcal{L}, s \rangle$ where:*

- *Σ_1 is a high-order signature called the abstract signature;*
- *Σ_2 is a high-order signature called the object signature;*
- *$\mathcal{L}$ is a lexicon from Σ_1 to Σ_2;*
- *s is a distinguished type belonging to $\mathcal{T}(A_1)$.*

An ACG generates two distinct languages: the abstract language, specifying the underlying admissible structures, and the object language, representing the surface forms of linguistic expressions.

Definition 6 (Abstract language). *Let $\mathcal{G}$ be an ACG, the abstract language of $\mathcal{G}$, $\mathcal{A}(\mathcal{G})$, is the set of linear lambda-terms defined as follows:*

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

Definition 7 (Object language). *Let $\mathcal{G}$ be an ACG, the object language of $\mathcal{G}$, is the set of linear lambda-terms defined as follows:*

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

These generated languages of linear λ-terms generalize both string and tree languages. Strings of symbols may be encoded as function compositions. Figure 1a defines the high-order signature in which the strings: *the leverage increases* and *the leverages increase* can be represented. The parse tree of *the leverage increases*, may be represented as a term built over the high-order signature defined in Figure 1b.

$$\begin{aligned}
\mathsf{o} &:: \mathsf{Type} \\
\mathtt{String} &= \mathsf{o} \multimap \mathsf{o} \\
\mathsf{infix} + &= \lambda^{\circ}x.\lambda^{\circ}y.\lambda^{\circ}z.x\ (y\ z) \\
/the/ &: \mathtt{String} \\
/leverage/ &: \mathtt{String} \\
/leverages/ &: \mathtt{String} \\
/increase/ &: \mathtt{String} \\
/increases/ &: \mathtt{String}
\end{aligned}$$

(a) String signature

$$\begin{aligned}
\mathtt{Tree} &:: \mathsf{Type} \\
the &: \mathtt{Tree} \\
leverage &: \mathtt{Tree} \\
increases &: \mathtt{Tree} \\
\mathtt{Det}_1 &: \mathtt{Tree} \multimap \mathtt{Tree} \\
\mathtt{N}_1, \mathtt{NP}_1 &: \mathtt{Tree} \multimap \mathtt{Tree} \\
\mathtt{V}_1, \mathtt{VP}_1 &: \mathtt{Tree} \multimap \mathtt{Tree} \\
\mathtt{S}_2 &: \mathtt{Tree} \multimap \mathtt{Tree} \multimap \mathtt{Tree}
\end{aligned}$$

(b) Tree signature

Fig. 1: Uniform representation of languages

ACGs are inherently reversible [7], they can be used in both directions, either generating or interpreting expressions. ACGs offer two primary computational paradigms. First, there is the applicative paradigm, which involves calculating the interpretation of an abstract term using the grammar's lexicon. Second, there is the deductive paradigm, which focuses on verifying whether a term u defined over the object signature belongs to the object language. This verification process depends on morphism inversion since it involves identifying the abstract terms, represented as t, for which $\mathcal{L}(t) = u$. The process of inverting morphisms for this purpose is referred to as ACG parsing.

A third paradigm, called the transductive paradigm, emerges when composing two ACGs with a shared abstract signature. It allows moving from one object language to the other object language by applying a combination of the deductive and applicative paradigms. We will use this paradigm in section 4.

1.2 Morphosyntax and Grammars

The composable nature of ACGs allows for the creation of complex architectures for natural language modeling. However, modeling morphosyntactic phenomena, such as agreement, results in a substantial enlargement of grammars. To illustrate this point, suppose that we want to define the derivation structures that allow for the construction of the following sentence: *The leverage increases*. Such language may be defined using the high-order signature presented in Figure 2a. However, if we aim to introduce the plural version of this sentence and reject the ungrammatical ones, we will need a notion of morphological number in the types, which will double the signature size, as shown in Figure 2b.

$$\begin{aligned} \mathtt{S}, \mathtt{N}, \mathtt{NP} &:: \mathsf{Type} \\ \textsc{the} &: \mathtt{N} \multimap \mathtt{NP} \\ \textsc{leverage} &: \mathtt{N} \\ \textsc{increases} &: \mathtt{NP} \multimap \mathtt{S} \end{aligned}$$

(a) Derivation signature

$$\begin{aligned} \mathtt{S}, \mathtt{N}_{sg}, \mathtt{N}_{pl}, \mathtt{NP}_{sg}, \mathtt{NP}_{pl} &:: \mathsf{Type} \\ \textsc{the}_{sg} &: \mathtt{N}_{sg} \multimap \mathtt{NP}_{sg} \\ \textsc{the}_{pl} &: \mathtt{N}_{pl} \multimap \mathtt{NP}_{pl} \\ \textsc{leverage} &: \mathtt{N}_{sg} \\ \textsc{leverages} &: \mathtt{N}_{pl} \\ \textsc{increases} &: \mathtt{NP}_{sg} \multimap \mathtt{S} \\ \textsc{increase} &: \mathtt{NP}_{pl} \multimap \mathtt{S} \end{aligned}$$

(b) Derivation signature with morphological variants

Fig. 2: Derivation signatures

Although the size of this toy example is small, it demonstrates the combinatorial explosion that occurs when introducing morphosyntactic constraints. Consequently, creating and maintaining grammars becomes time-consuming. In addition, ACG parsing becomes less efficient due to the number of constants to consider.

It can get worse, especially for languages like French that have many morphological traits. ACGs lack a commonly utilized mechanism in grammatical engineering: feature structures. These structures are advantageous for succinctly describing the morphosyntactic rules of languages, such as agreement. The main goals of this paper are both theoretical and practical:

1. to introduce a conservative extension to the core ACGs that incorporates feature structures;
2. to experimentally demonstrate the benefits of the extension through a rewrite of a French grammar;
3. to empirically confirm that the size reduction impacts positively ACG parsing.

2 Proposed Extension

2.1 Modeling Feature Structures

The introductory example raises two questions: how can we effectively represent feature structures, and what are the best methods for associating these structures with atomic types? Our proposition involves the representation of feature structures by incorporating well-established constructs, namely records and enumerations. To annotate atomic types with feature structures, we advocate for the adoption of dependent types. Extending ACG with dependent types has already been explored [4, 14]. However, it has been demonstrated that parsing becomes undecidable with fully fledged dependent types. Our primary objective is to propose an extension that does not augment the expressive power of the framework. Parsing in a specific class of ACGs can be reduced to the polynomial evaluation of a Datalog program [10, 11], we aim to preserve this property.

Our extension, ACG with feature structures (f-ACG), utilizes typed feature structures, where the types can only be either an enumeration or a record type.

Definition 8 (Feature structure types). *The set of feature structures types, denoted as $\mathcal{F}$, is defined by the following BNF:*

$$
\begin{aligned}
e &\in E && \text{(Enumeral symbol)}\\
l &\in L && \text{(Label symbol)}\\
\mathcal{E} &\coloneqq \{e_1 \mid \ldots \mid e_n\} && \text{(Enumerated type)}\\
\mathcal{R} &\coloneqq \left[l_1 : \mathcal{F}, \ldots, l_n : \mathcal{F}\right] && \text{(Record type)}\\
\mathcal{F} &\coloneqq \mathcal{E}\\
&\mid \mathcal{R}
\end{aligned}
$$

Definition 9 (Feature structure terms). *The set of feature structure terms, F, is defined as follows:*

$$
\begin{aligned}
x &\in \mathcal{X} && \text{(Variable symbol)}\\
F &\coloneqq e && \text{(Enumeral)}\\
&\mid x && \text{(Feature structure variable)}\\
&\mid \left[l_1 = F, \ldots, l_n = F_n\right] && \text{(Record)}\\
&\mid F.l && \text{(Selection)}
\end{aligned}
$$

Figure 3 introduces the typing relation for feature structures. In these rules, Γ is a context of feature structure variables.

We extend ACG types with two new constructors: type abstraction and type application. Only feature structure terms are permitted as dependencies for types. This constraint is formulated using an upper level that is above types, which is referred to as *kind*. Figure 4 introduces the kinding judgment.

$$\frac{}{\Gamma, x : \mathcal{F} \vdash_\Sigma x : \mathcal{F}}\ \textsc{var-feat}$$

$$\frac{\vdash_\Sigma e \in \{e_1, \ldots, e_n\}}{\vdash_\Sigma e : \{e_1 \mid \ldots \mid e_n\}}\ \textsc{enumeral-feat}$$

$$\frac{\Gamma \vdash_\Sigma f_i : \mathcal{F}_i \qquad \forall i \in [1, n]}{\Gamma \vdash_\Sigma \left[l_i = f_i\right]_{i=1}^n : \left[l_i : \mathcal{F}_i\right]_1^n}\ \textsc{record-feat}$$

$$\frac{\Gamma \vdash_\Sigma f : \left[l_i : \mathcal{F}_i\right]_{i=1}^n \qquad l_k \in \{l_i\}_{i=1}^n}{\Gamma \vdash_\Sigma f.l_k : \mathcal{F}_k}\ \textsc{selection-feat}$$

Fig. 3: Feature structures typing rules

Definition 10 (Extended Types). *Let A be a set of atomic types, the set of extended types over A, $\mathcal{T}(A)$ is defined as follows:*

$$\begin{aligned}
\mathcal{T}(A) \coloneqq\ & A && \text{(Atomic type or Type family)}\\
\mid\ & \mathcal{T}(A) \multimap \mathcal{T}(A) && \text{(Linear functional type)}\\
\mid\ & \Lambda x : \mathcal{F}.\mathcal{T}(A) && \text{(Type abstraction)}\\
\mid\ & \mathcal{T}(A)\ F && \text{(Type application)}
\end{aligned}$$

Definition 11 (Kind). *The set of kinds, $\mathcal{K}$, is defined by the following BNF grammar:*

$$\begin{aligned}
\mathcal{K} \coloneqq\ & \mathsf{Type} && \text{(Kind of proper types)}\\
\mid\ & \mathcal{F} \Rrightarrow \mathcal{K} && \text{(Kind of families)}
\end{aligned}$$

We avoid the full expressiveness of dependent products by enforcing them to only appear in prenex form through the introduction of a new type level, referred to as *generalized types*, and a new kind level, referred to as *generalized kinds*. Figure 5 defines the rules for the interaction between these two generalized levels.

Definition 12 (Generalized types). *The set of generalized types over A, $\mathcal{D}(A)$, and the set of generalized kind, $\mathcal{K}_\mathcal{D}$, are defined by the following grammar:*

$$\begin{aligned}
\mathcal{D}(A) \coloneqq\ & \mathcal{T}(A) && \text{(Types)}\\
\mid\ & \Pi x : \mathcal{F}.\mathcal{D}(A) && \text{(Dependent product)}\\
\mathcal{K}_\mathcal{D} \coloneqq\ & \mathsf{DType} && \text{(Kind of proper generalized types)}\\
\mid\ & \mathcal{K} && \text{(Kind)}
\end{aligned}$$

$$\frac{a :: \mathcal{K} \in \Sigma}{\Gamma \vdash_\Sigma a :: \mathcal{K}} \text{ ATOMIC-TYPE}$$

$$\frac{\Gamma \vdash_\Sigma \alpha :: \mathsf{Type} \qquad \Gamma \vdash_\Sigma \beta :: \mathsf{Type}}{\Gamma \vdash_\Sigma \alpha \multimap \beta :: \mathsf{Type}} \text{ LFUNC-TYPE}$$

$$\frac{\Gamma \vdash_\Sigma \alpha :: \mathcal{F} \Rrightarrow \mathcal{K} \qquad \Gamma \vdash_\Sigma f : \mathcal{F}}{\Gamma \vdash_\Sigma \alpha\ f :: \mathcal{K}} \text{ APP-TYPE}$$

$$\frac{\Gamma, x : \mathcal{F} \vdash_\Sigma \alpha :: \mathcal{K}}{\Gamma \vdash_\Sigma \Lambda x : \mathcal{F}.\alpha :: \mathcal{F} \Rrightarrow \mathcal{K}} \text{ ABSTR-TYPE}$$

Fig. 4: Type formation rules

$$\frac{\Gamma \vdash_\Sigma \alpha :: \mathsf{Type}}{\Gamma \vdash_\Sigma \alpha :: \mathsf{DType}} \text{ PROMOTION-TYPE}$$

$$\frac{\Gamma, x : \mathcal{F} \vdash_\Sigma \alpha :: \mathsf{DType}}{\Gamma \vdash_\Sigma \Pi x : \mathcal{F}.\alpha :: \mathsf{DType}} \text{ PI-TYPE}$$

Fig. 5: Generalized type formation rules

2.2 Extended Abstract Categorial Grammar

The notion of a signature is redefined to consider the kinding relation and generalized types.

Definition 13 (Extended signature). *An extended signature is a quadruplet* $\Sigma = \langle A, \kappa, C, \tau \rangle$ *where:*

- A *is a finite set of atomic types;*
- $\kappa : A \to \mathcal{K}$ *is a function that assigns to each atomic type a kind;*
- C *is a finite set of constants;*
- $\tau : C \to \mathcal{D}(a)$ *is a function that assigns to each constant a generalized type;*
- $\tau(c)$ *is in* β*-normal form and the following kinding judgment holds:*

$$\vdash_\Sigma \tau(c) :: \mathsf{DType}$$

The inclusion of dependent products leads at the term level to the introduction of feature abstractions, feature applications, and case analysis. Figure 6 defines the typing relation for terms. A second context, Δ, appears in these rules for linear assumptions. This kind of type systems where linear and dependent types coexist has already been explored in an extension of the logical framework [3].

$$\frac{c : \mathcal{D} \in \Sigma}{\Gamma; \Delta \vdash_\Sigma c : \mathcal{D}}\ \textsc{const-term}$$

$$\frac{}{\Gamma; x : \alpha \vdash_\Sigma x : \alpha}\ \textsc{lvar-term}$$

$$\frac{\Gamma; \Delta_1 \vdash_\Sigma t : \alpha \multimap \beta \qquad \Gamma; \Delta_2 \vdash_\Sigma u : \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash_\Sigma t\ u : \beta}\ \textsc{lapp-term}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma t : \Pi x : \mathcal{F}.\alpha \qquad \Gamma; \vdash_\Sigma f : \mathcal{F}}{\Gamma; \Delta \vdash_\Sigma t \bullet f : \alpha[x := f]}\ \textsc{app-term}$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda^\circ x : \alpha . t : \alpha \multimap \beta}\ \textsc{labstr-term}$$

$$\frac{\Gamma, x : \mathcal{F}; \Delta \vdash_\Sigma t : \alpha}{\Gamma; \Delta \vdash_\Sigma \lambda x : \mathcal{F}.t : \Pi x : \mathcal{F}.\alpha}\ \textsc{abstr-term}$$

$$\frac{\Gamma, x : \mathcal{E} \vdash_\Sigma \alpha :: \mathsf{Type} \qquad \Gamma; \vdash_\Sigma f : \mathcal{E} \qquad \Gamma; \vdash_\Sigma t_i : \alpha[x := e_i]}{\Gamma; \vdash_\Sigma \mathsf{case}\ f\ \{e_i \to t_i\}_{i=1}^n : \alpha[x := f]}\ \textsc{case-var-term}$$

Fig. 6: Typing rules

Definition 14 (Terms). *Let $\mathcal{X}$ be a countable set of variables, the set of lambda-terms defined over a signature Σ, $\Lambda(\Sigma)$, is defined by the following BNF grammar:*

$$\begin{aligned}
\Lambda(\Sigma) \coloneqq\ & c && \text{(Constant)}\\
\mid\ & x && \text{(Variable)}\\
\mid\ & \Lambda(\Sigma)\ \Lambda(\Sigma) && \text{(Application)}\\
\mid\ & \Lambda(\Sigma) \bullet F && \text{(Feature structure application)}\\
\mid\ & \lambda^\circ x : \mathcal{T}(A).\Lambda(\Sigma) && \text{(Linear lambda abstraction)}\\
\mid\ & \lambda x : \mathcal{F}.\Lambda(\Sigma) && \text{(Feature structure abstraction)}\\
\mid\ & \mathit{case}\ F\ \{e_1 \to \Lambda(\Sigma) \mid \cdots \mid e_n \to \Lambda(\Sigma)\} && \text{(Case analysis)}
\end{aligned}$$

A noteworthy aspect of this extension lies in the interpretation of abstract feature structure terms and types. They are interpreted in types and terms as they appear in abstract expressions. As a result, the interpretation of types must preserve the kinding judgment.

Definition 15 (Extended lexicon). *Let $\Sigma_1 = \langle A_1, \kappa_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, \kappa_2, C_2, \tau_2 \rangle$ be two extended signatures. An extended lexicon $\mathcal{L}$ is a pair $\langle F, G \rangle$ where:*

- *$F : A_1 \to \mathcal{T}(\Sigma_2)$ is a function that interprets atomic types with types defined over the object signature.*

- $G : C_1 \to \Lambda(\Sigma_2)$ *is a function that interprets abstract constants as terms defined over the object signature.*
- *For each* a *in* A_1*, the following kinding judgment is derivable:*

$$\vdash_{\Sigma_2} F(a) :: \kappa_1(a)$$

- *For each* c *in* C_1*, the following typing judgment is derivable:*

$$\vdash_{\Sigma_2} G(c) : H(\tau_1(c))$$

where $H : \mathcal{D}(A_1) \to \mathcal{D}(A_2)$ *is a morphism defined such that:*

- *If* $\vdash_{\Sigma_1} \alpha :: \mathsf{Type}$*, then* $H(\alpha) = \hat{F}(\alpha)$
- *If* $\vdash_{\Sigma_1} \alpha :: \mathsf{DType}$ *and* $\alpha \equiv \Pi x : \mathcal{F}.\beta$ *then* $H(\alpha) = \Pi x : \mathcal{F}.H(\beta)$

Definition 16 (Extended Grammar). *An extended ACG* $\mathcal{G}$ *is a quadruplet* $\langle \Sigma_1, \Sigma_2, \mathcal{L}, s \rangle$ *where:*

- Σ_1 *is an extended signature called the abstract signature;*
- Σ_2 *is an extended signature called the object signature;*
- $\mathcal{L}$ *is an extended lexicon from* Σ_1 *to* Σ_2*;*
- s *is a distinguished type belonging to* $\mathcal{T}(A_1)$ *in* β*-normal form.*

In Figure 7, we revisit the introductory derivation signature using an extended signature. Figure 8 provides a definition of the lexicon from this extended abstract signature to the string signature defined in Figure 1a. The categories `N` and `NP` are interpreted as type abstractions. They accept a number but return a string type, disregarding the value of the parameter. Furthermore, the interpretation of INCREASE and LEVERAGE involves case analysis to modify the string according to its associated number.

$$\begin{aligned}
\mathtt{S} &:: \mathsf{Type} \\
Num &= \{sg, pl\} \\
\mathtt{N} &:: \left[n : Num\right] \Rrightarrow \mathsf{Type} \\
\mathtt{NP} &:: \left[n : Num\right] \Rrightarrow \mathsf{Type} \\
\textsc{leverage} &: \Pi x : Num.\mathtt{N}\left[n = x\right] \\
\textsc{the} &: \Pi x : Num.\mathtt{N}\left[n = x\right] \multimap \mathtt{NP}\left[n = x\right] \\
\textsc{increase} &: \Pi x : Num.\mathtt{NP}\left[n = x\right] \multimap \mathtt{S}
\end{aligned}$$

Fig. 7: Extended derivation signature

$$
\begin{aligned}
\mathcal{L}(\mathtt{S}) &= \mathtt{String} \\
\mathcal{L}(\mathtt{N}) &= \Lambda x : \mathit{Num}.\mathtt{String} \\
\mathcal{L}(\mathtt{NP}) &= \Lambda x : \mathit{Num}.\mathtt{String} \\
\mathcal{L}(\textsc{the}) &= \lambda x : \mathit{Num}.\lambda^{\circ} n : \mathtt{String}./\mathit{the}/ + n \\
\mathcal{L}(\textsc{increase}) &= \lambda x : \mathit{Num}.\lambda^{\circ} \mathit{np} : \mathtt{String}. \\
&\qquad \mathsf{case}\ x\ \{\mathit{sg} \to \mathit{np} + /\mathit{increases}/ \mid \mathit{pl} \to \mathit{np} + /\mathit{increase}/\} \\
\mathcal{L}(\textsc{leverage}) &= \lambda x : \mathit{Num}. \\
&\qquad \mathsf{case}\ x\ \{\mathit{sg} \to /\mathit{leverage}/ \mid \mathit{pl} \to /\mathit{leverages}/\}
\end{aligned}
$$

Fig. 8: Extended lexicon

3 Theoretical Results

3.1 Expressive Power

This extension does not augment the expressive power of ACG, as it can be established that any f-ACG is transformable into a core ACG. Formally, the following proposition holds.

Proposition 1. *Let $\mathcal{G}$ be a f-ACG grammar, there exists a core ACG grammar, $\mathcal{G}'$, wherein the abstract and object languages generated by $\mathcal{G}'$ are isomorphic to those generated by $\mathcal{G}$.*

Due to space limitations, we are not presenting the proof in this paper. The intuition about the transformation from a f-ACG to a core ACG is very similar to the reduction of context-free grammars augmented with feature structures into core CFGs. Rules with feature structure variables are instantiated and duplicated with all the possible feature values, and new non-terminals are created with all the combination of the traits.

The transformation of f-ACGs utilizes the fact that feature structures are finite. This property allows for the use of a partial generation mechanism to remove feature structures from terms and types. The transformation can be achieved through a three-step construction, which is based on the dependency graph of judgments presented in Figure 9. First, the procedure begins with the elimination of dependent products, which removes the dependency of terms on feature structures. The second step involves the elimination of type families, after which feature structures are no longer present in types. Finally, the construction concludes with the type erasure of typed linear abstractions, resulting in a core ACG.

3.2 Parsing

This transformation generates grammars whose size renders parsing inefficient for practical real-world applications due to the elimination of factorizations.

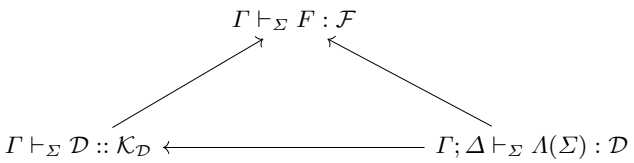


Fig. 9: Judgment dependency graph

One potential alternative approach involves utilizing the underlying context-free barebone and subsequently filtering out inappropriate solutions. However, encoding feature structures directly into the Datalog reduction is more natural and likely more efficient.

Drawing inspiration from definite clause grammar [12], our approach to enhancing the Datalog reduction incorporates several key principles for integrating feature structures. First, atomic feature structure terms are encoded as Datalog constants. This allows for their integration into predicates corresponding to abstract types.

Since pure Datalog does not support records, they are flattened and subsequently, labels are eliminated. Additionally, feature types are encoded as extensional predicates, which allows for modeling the feature structure parameters of dependent products. The record treatment prioritizes simplicity and efficiency by reducing complex data structures to their most basic form. However, it's important to note that certain extensions of Datalog have emerged to address the need for modeling complex objects [2], such as records, without sacrificing essential properties.

Figure 10 shows the Datalog reduction of the introductory example. The construction of a noun phrase with a determiner stays factorized in its corresponding Datalog rule. In contrast, the remaining rules have been divided based on the case analysis that occurs in the extended lexicon.

```
Number(sg).
Number(pl).
S(a,c) :- NP(a,b,sg), increases(b,c).
S(a,c) :- NP(a,b,pl), increase(b,c).
NP(a,c,x) :- Number(x), the(a,b), N(b,c,x).
N(a,b,sg) :- leverage(a,b).
N(a,b,pl) :- leverages(a,b).
```

Fig. 10: Datalog reduction

4 Experimental Results

4.1 Grammar Factorization

To measure experimentally the performance and factorization gains of the overall extension in practical applications, we have developed a f-ACG compiler in Java. This proprietary compiler is designed for a company that uses ACG to automatically produce financial and pharmaceutical reports [16] from data. The extension of ACG with feature structures will, however, soon be available in the open-source Abstract Categorial Grammar Toolkit [8], ACGtk.

We've opted to explore how the extension affects the development of French grammars, given its abundance of rich morphosyntactic combinations. The grammar utilized in this experiment draws its foundations from the ACG encoding [13] of a wide-coverage French tree-adjoining grammar[1]. From an architectural perspective, the grammar is defined as the composition of two main ACGs as shown in Figure 11. The level of derivation trees acts as the pivot between derived trees and the semantic level. The first ACG $\mathcal{G}_{\text{sem.}}$ defines the correspondence between the derivation and the semantic level, while the other, $\mathcal{G}_{\text{derived trees}}$, establishes the interpretation of derivation trees into derived trees. A third ACG, $\mathcal{G}_{\text{yield}}$, is used to flatten derived trees into strings.

The grammar employed here serves as the core component of an engine crafted for generating texts based on data. It consists of multiple unanchored, modular parts that can be dynamically assembled to suit different text generation tasks. This method enables the creation of lexicalizations in real-time, tailored to match the input. The goal of this approach is to develop extremely compact grammars for rapid text production.

The design philosophy behind the semantic level emphasizes simplicity and minimal reliance on linguistic knowledge. This approach results in a reduction in expressiveness, particularly when compared to more complex formalizations of semantics. Σ_{semantic} is defined utilizing a sole atomic type, represented by `Concept`. Within this high-order signature, entities are defined as `Concept`. Furthermore, elements such as relations, predicates, and modifiers are modeled as functions within this system. Consequently, the constructs formulated within Σ_{semantic} can be seen as elementary semantic graphs.

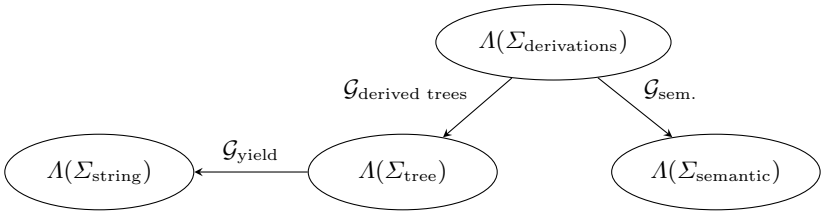


Fig. 11: French Grammar Architecture

We conducted a complete manual rewrite of the industrial French grammar within the f-ACG extension. Figure 12 shows the derivation tree families in the grammar with their associated number of derivation trees. Overall, the original high-order signature $\Sigma_{\text{derivations}}$ consists of 51,246 constants (unanchored derivation trees). The translation in f-ACG results in a grammar that is 18 times smaller, reducing the original count of 51,246 unanchored derivation trees to 2,792. The parts that are the most impacted by the addition of feature structures are the verbal, prepositional and nominal phrases. The category *other* is also significantly affected. It includes trees for various purposes, such as punctuation and linking words between sentences.

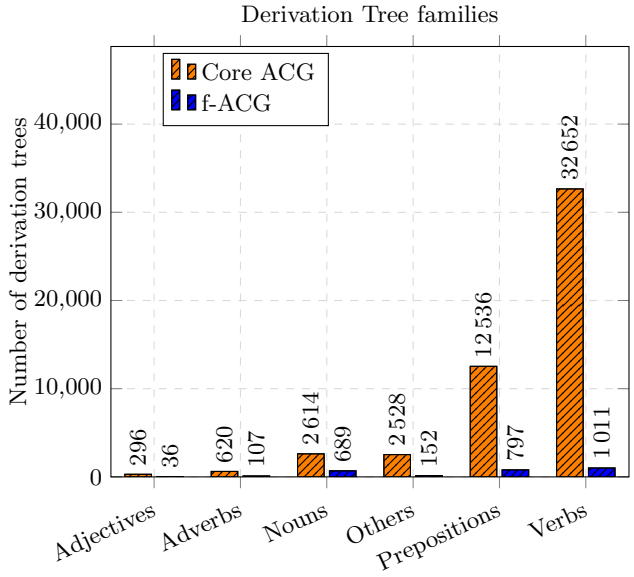


Fig. 12: Comparison of the numbers of trees in the two derivation signatures

4.2 Text Generation Performances

To evaluate the performance gains of this factorized French grammar, we have set up two distinct financial text generation scenarios, each of which has been executed 30 times to ensure reliability in our findings. Each scenario features semantic graphs of varying sizes, providing an examination of the system's performance across different complexities. Text generation for these scenarios is purely based on ACG operations from a semantic graph (a term in $\Lambda(\Sigma_{\text{semantic}})$) into texts (terms in $\Lambda(\Sigma_{\text{string}})$).

In the first scenario, called *Describe Value*, the textual content is generated based on a relatively simple semantic graph consisting of 6 nodes, as depicted in

Figure 13. The graph contains information about NewCo's financial performance, specifically stating that: *NewCo had recorded satisfactory revenues of $921,283.5.*

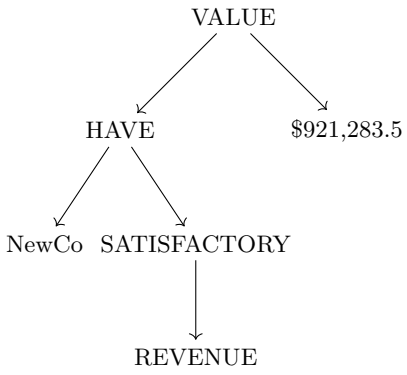
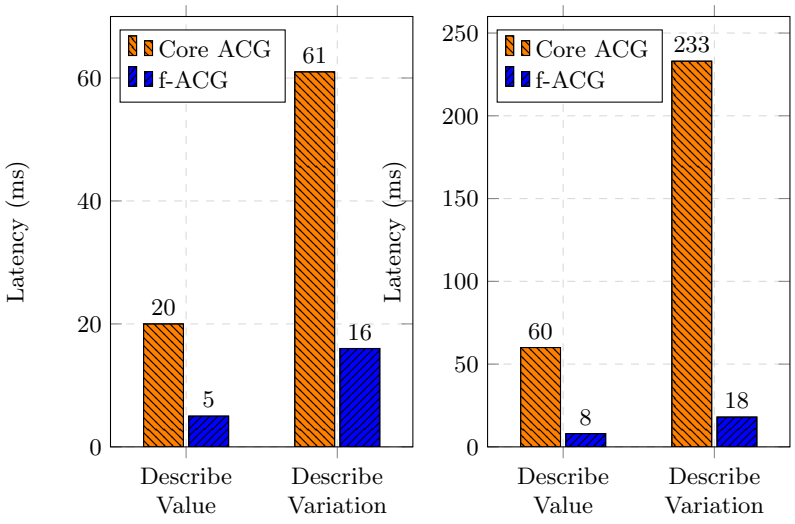


Fig. 13: Semantic graph of *Describe Value*

In the second scenario, referred to as *Describe Variation*, a semantic graph of greater complexity, consisting of 15 nodes, is employed. This graph provides historical context by describing NewCo's gross margin in 2017, which was unsatisfactory at a value of $1,463,000, and the subsequent decrease by 0.5% until 2018, signaling poor sales performance. For instance, a translation of one of the French texts generated from this graph could read: *In 2017, NewCo's gross margin of $1,463,000 was unsatisfactory and declined by 0.5% until 2018, indicating poor sales.* Text generation in this condition aims to not only convey the numerical data, but also to provide a narrative that explains the significance of these figures in the context of NewCo's business operations and sales trends.

The time spent to generate text (the latency) using this grammar heavily depends on the size of the input semantic graph and the number of used lexicalizations. However, we have found that using the compiler mentioned above, when comparing the time spent on generation with a core ACG and its factorized version in f-ACG for the two scenarios, the latency is reduced by 4 as Figure 14a shows. In addition to having an interesting factorization power, the proposed extension also has a positive effect on performances.

When we integrate the lexicalization process and the compilation of the compact grammar tailored to the input with the text generation phase, a significant reduction in processing time is achieved. This efficiency becomes even more visible when dealing with complex syntactic structures and semantic graph inputs, as Figure 14b shows. As these elements increase in complexity, the gains observed in the process become more pronounced.



(a) Text generation

(b) Grammar initialization and text generation

Fig. 14: Benchmarks

4.3 Conclusion

In summary, the extension we've introduced provides significant advantages. It reduces the size of grammars and enhances the performance of ACG parsing. It does so while still maintaining the fundamental characteristics of the ACG formalism.

This paper has detailed the basic mathematical framework of the extension. However, it's important to note that in practice, writing grammars requires some facilities in terms of concrete syntax, notably pattern matching and modules. The incorporation of pattern matching in the ACG lexicon enables concise and clear expression of grammatical variations based on feature structure terms. Lastly, a modular system is critical for managing the complexity of grammatical systems. By structuring the ACG grammar into modular components, each responsible for a specific aspect of the language, we enhance reusability and extensibility.

It should be mentioned that our approach to feature structures is unconventional compared to most unification-based approaches to grammar. We have not explored elements like reentrancy and the f-ACG encoding of feature structures-based formalisms in this study, leaving them as potential areas for further developments.

References

1. Anne Abeillé. *Une grammaire électronique du français*. Sciences du langage. CNRS Éditions, 2002.
2. Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Extensions of Pure Datalog*, pages 208–245. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.
3. Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and computation*, 179(1):19–75, 2002.
4. Ph. de Groote and S. Maarek. Type-theoretic Extensions of Abstract Categorial Grammars. In *New Directions in Type-theoretic Grammars (NDTTG 2007), ESSLLI 2007 workshop*, Dublin, 2007.
5. Philippe de Groote. Towards Abstract Categorial Grammars. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics*, pages 148–155, 2001.
6. Philippe de Groote and Sylvain Pogodalla. On the expressive power of Abstract Categorial Grammars: Representing context-free formalisms. *Journal of Logic, Language and Information*, 13(4):421–438, 2004.
7. Marc Dymetman. Inherently reversible grammars. In Tomek Strzalkowski, editor, *Reversible Grammars in Natural Language Processing*, chapter 2, pages 33–57. Kluwer Academic Publishers, 1994.
8. Maxime Guillaume, Sylvain Pogodalla, and Vincent Tourneur. ACGtk: A Toolkit for Developing and Running Abstract Categorial Grammars. In Jeremy Gibbons and Dale Miller, editors, *17th International Symposium on Functional and Logic Programming (FLOPS 2024)*, Kumamoto, Japan, May 2024.
9. Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto K. Salomaa, editors, *Handbook of formal languages*, volume 3, chapter 2. Springer, 1997.
10. Makoto Kanazawa. Parsing and generation as datalog queries. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL 2007)*, pages 176–183, Prague, Czech Republic, June 2007. Association for Computational Linguistics.
11. Makoto Kanazawa. Parsing and generation as datalog query evaluation. *IfCoLog Journal of Logics and their Applications*, 4(4):1103–1211, 2017. Special Issue Dedicated to the Memory of Grigori Mints.
12. Fernando CN Pereira and David HD Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, 13(3):231–278, 1980.
13. Sylvain Pogodalla. A syntax-semantics interface for Tree-Adjoining Grammars through Abstract Categorial Grammars. *Journal of Language Modelling*, 5(3):527–605, 2017.
14. Florent Pompigne. *Modélisation logique de la langue et Grammaires Catégorielles Abstraites*. PhD thesis, Université de Lorraine, December 2013.
15. Aarne Ranta. Grammatical Framework: A type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.
16. Raphael Salmon. *Natural Language Generation Using Abstract Categorial Grammars*. PhD thesis, Sorbonne Paris Cité, 2017.