UNIVERSITÉ DE LORRAINE | BIBLIOTHÈQUES UNIVERSITAIRES

AVERTISSEMENT
DROIT D'AUTEUR

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document. Toute contrefaçon, plagiat ou reproduction illicite encourt une poursuite pénale.

Contact bibliothèque : ddoc-theses-contact@univ-lorraine.fr
(Cette adresse ne permet pas de contacter les auteurs)

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10

UNIVERSITÉ DE LORRAINE
34 COURS LÉOPOLD — CS 25233
54052 NANCY CEDEX
TÉL : 03 54 50 54 00
CONTACT@UNIV-LORRAINE.FR
WWW.UNIV-LORRAINE.FR

UNIVERSITY OF LORRAINE

DOCTORAL THESIS

Abstract Categorial Grammars and Feature Structures

Maxime GUILLAUME

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy in Informatique

Publicly defended on 21 novembre 2025

Sémagramme & Yseop
Laboratoire Lorrain de Recherche en Informatique et ses Applications
IAEM-Lorraine

Jury

President:	DR Claire GARDENT	University of Lorraine, France
Reviewers:	Pr. Laura KALLMEYER	Duesseldorf University, Germany
	CR HDR Richard MOOT	LIRMM, France
Examiners:	MCF HDR Annie FORÊT	University of Rennes, France
	Pr. Aarne RANTA	Chalmers University, Sweden
Supervisors:	DR Philippe de GROOTE	Inria Grand Est, France
	Dr. Raphaël SALMON	Yseop, France

UNIVERSITY OF LORRAINE

Abstract

Abstract Categorial Grammars and Feature Structures

by Maxime GUILLAUME

Abstract Categorial Grammars (ACGs) are a powerful grammatical formalism based on the simply typed linear λ-calculus, capable of uniformly representing several levels of linguistic analysis. However, they suffer from a practical weakness: the lack of intrinsic mechanisms for handling feature structures, which are essential for modeling the morphosyntactic phenomena of natural languages. This thesis introduces Affix Abstract Categorial Grammars (AACGs), an extension of ACGs enriched by the integration of feature structures.

First, we introduce an enriched λ-calculus that extends the simply typed linear λ-calculus with enumerations, records, and dependent products. On this basis, we define AACGs and demonstrate their strong equivalence with classical ACGs through a series of formal transformations.

We then examine the algorithmic implications of this equivalence for parsing. In this regard, we present an adaptation of Kanazawa's reduction, which converts the parsing of certain classes of ACGs into the evaluation of Datalog programs. This approach guarantees polynomial-time complexity while preserving the factorization benefits specific to AACGs.

To validate the industrial applicability of this approach, a dedicated compiler for AACGs was designed and implemented, integrated into a text generation engine. Experiments conducted on a large-coverage French grammar highlight a significant reduction in grammar size as well as a notable improvement in parsing and generation performance.

UNIVERSITÉ DE LORRAINE

Résumé

Structures de traits pour les Grammaires Catégorielles Abstraites

par Maxime GUILLAUME

Les Grammaires Catégorielles Abstraites (ACG) constituent un formalisme grammatical puissant reposant sur le λ-calcul linéaire simplement typé, capable de représenter de manière unifiée plusieurs niveaux d'analyse linguistique. Toutefois, elles souffrent d'une faiblesse pratique: l'absence de mécanismes intrinsèques pour gérer les structures de traits, pourtant essentielles à la modélisation des phénomènes morphosyntaxiques des langues naturelles. Cette thèse introduit les Grammaires Catégorielles Abstraites à Affixes (AACG), une extension des ACG enrichie par l'intégration de structures de traits.

Nous introduisons d'abord un λ-calcul enrichi qui étend le λ-calcul linéaire simplement typé avec des énumérations, des enregistrements et des produits dépendants. Sur cette base, nous définissons les AACG et démontrons leur équivalence forte avec les ACG classiques par une série de transformations formelles.

Les retombées algorithmiques de cette équivalence pour l'analyse syntaxique sont ensuite examinées. À cet égard, une adaptation de la réduction de Kanazawa est présentée, qui convertit l'analyse syntaxique de certaines classes d'ACG en évaluation de programmes Datalog. Cette démarche garantit une complexité en temps polynomial tout en préservant les bénéfices de factorisation propres aux AACG.

Pour valider l'applicabilité industrielle de cette approche, un compilateur dédié aux AACG a été conçu et implémenté, intégré à un moteur de génération de texte. Les expérimentations menées sur une grammaire française à large couverture mettent en évidence une réduction significative de la taille de la grammaire ainsi qu'une amélioration notable des performances d'analyse et de génération.

Acknowledgements

This research was made possible by the generous financial support of the ANRT through the CIFRE thesis program, for which I am truly grateful.

Completing this thesis has been a significant endeavor, and I could not have done it without the crucial support I received along the way. My sincere thanks go to my supervisors, Philippe and Raphaël, for their guidance and consistent support. I also wish to thank Sylvain for his valuable insights and informal supervision, which greatly benefited my work.

I would like to thank Richard Moot and Laura Kallmeyer for accepting to review my doctoral thesis, and Annie Forêt, Claire Gardent, and Aarne Ranta for accepting to participate in my jury.

My sincere thanks go to the entire Yseop Team. This research was significantly enriched by the opportunity to work on practical applications of Abstract Categorial Grammars, and I am particularly indebted to Agathe for her dedicated assistance in rewriting the French grammar, a critical component of my evaluation.

I also wish to acknowledge my fellow PhD students from both the Sémagramme and Carbon teams. Their stimulating discussions, collaborative spirit, and shared experiences provided a vibrant and supportive environment for my work.

Finally, I want to express my deepest appreciation to my friends and family. Your constant encouragement, understanding, and unwavering belief in me were a source of strength.

Contents

Abstract **iii**

Résumé **v**

Acknowledgements **vii**

Introduction **1**

I Mathematical Preliminaries **5**

1 Lambda-Calculus **7**
1.1 Pure Lambda-Calculus 7
1.1.1 Syntax 7
1.1.2 Equivalence Up to Variable Renaming 9
1.1.3 Simplifying Applications 9
1.1.4 Eliminating Redundant Abstractions 11
1.1.5 Syntactic Restrictions 12
1.2 Simply Typed Lambda-Calculus 13
1.2.1 Types 13
1.2.2 Properties of Typing 16
1.2.3 Principal Typing 18

2 Datalog **21**
2.1 Syntax 22
2.1.1 Primitives and Operations 22
2.1.2 Program 23
2.2 Semantics 25

2.2.1 Approaches to Datalog Semantics 25
2.2.2 Evaluation Algorithms 27

II ACGs **33**

3 Abstract Categorial Grammars **35**
3.1 Definition of the Formalism 36
3.1.1 Primitives 36
3.1.2 Grammar 38
3.1.3 Paradigms 40
3.2 Uniform languages 42
3.2.1 Encoding Strings 43
3.2.2 Encoding Trees 44
3.2.3 ACG Hierarchy 47
3.3 Encoding Context-Free Grammars 48
3.3.1 Context-Free Grammars 48
3.3.2 ACG Encoding 48
3.4 Extensions 51
3.4.1 Enriching the Kernel 51
3.4.2 Relaxing Linearity Constraints 51

4 Parsing **53**
4.1 Algorithmic Foundations 53
4.1.1 Typed Terms 53
4.1.2 Types 55
4.2 Kanazawa's Reduction 56
4.2.1 Program Construction 57
4.2.2 Instance Generation 59
4.2.3 Datalog Proof Trees and Antecedents 60
4.3 Extensions of the Reduction 62
4.3.1 Beyond Linear λ-terms 62
4.3.2 Beyond Second-Order 63

III Feature Structure Integration **65**

5 A Lambda-Calculus for Modelling Feature Structures **67**
5.1 Motivating Feature Structures in ACGs . . . 67
5.1.1 A Motivating Example: CFG Augmented with Feature Structures . . 67
5.1.2 Limitations of Direct ACG Encoding . . . 68
5.2 Modelling Feature Structures . . . 71
5.2.1 Atomic And Compound Feature Structures . . . 71
5.2.2 Feature Structure Annotations as Type Families . . . 74
5.2.3 Variable Sharing as Dependent Products . . . 76
5.2.4 Interpretations with Feature Structures . . . 77
5.3 A Feature-Structure Enriched Linear Lambda Calculus . . . 81
5.3.1 Definition . . . 81
5.3.2 Static Properties . . . 87
5.3.3 Strong Normalization . . . 95

6 Affix Abstract Categorial Grammars **97**
6.1 Grammars . . . 97
6.1.1 Definitions . . . 97
6.1.2 Defining Equivalence Between Grammars . . . 100
6.1.3 A Conservative Extension . . . 101
6.2 Expressivity of AACGs . . . 101
6.2.1 Pi-Elimination . . . 103
6.2.2 Record Elimination . . . 107
6.3 AACG Parsing . . . 112
6.3.1 Context-Free Backbone with Feature Filtering . . . 114
6.3.2 Feature-Aware Datalog Program . . . 117
6.4 Modelling Omissions . . . 124
6.4.1 Omissions as Coercions . . . 124
6.4.2 Type Hierarchies . . . 126

7 Implementation and Evaluation **129**
7.1 Compiler Overview . . . 129
7.1.1 Description . . . 129
7.1.2 Syntactic Sugar . . . 130
7.2 Derivation Recovery: Custom Prover vs. Standard Solver . . . 134

7.2.1 A Transformation-Based Solution for Standard Solvers 134
7.2.2 Practical Implications and Implementation Decision 137
7.3 Experiments 138
7.3.1 French Grammar 138
7.3.2 Text generation 140
7.3.3 Parsing 142

Conclusion **145**

Résumé Étendu **147**

Bibliography **155**

Introduction

Natural Language Generation (NLG), a specialized subfield of Natural Language Processing (NLP), focuses on the automated production of human-readable text. As a complement to *Natural Language Understanding* (NLU), which interprets human language input, NLG represents the generative side of NLP, creating text that meets human standards of fluency and coherence. NLG systems operate through two main approaches: transforming structured data into coherent narratives, and reformulating existing text into new forms. In data-to-text scenarios, these systems convert information from databases, knowledge graphs, or other formal representations into natural language descriptions. In text-to-text scenarios, they perform tasks like summarization, simplification, or stylistic adaptation while preserving essential meaning.

NLG has gained significant commercial traction across industries, with particularly impactful applications in finance for automated market analyses and financial reports, and in healthcare for patient summaries and clinical documentation. Modern approaches range from template-based systems to sophisticated neural architectures that generate increasingly natural and contextually appropriate text. These systems face complex challenges: maintaining factual accuracy, ensuring logical coherence across longer passages, and adapting linguistic style and complexity to match the intended audience.

Yseop, a commercial NLG technology company, exemplifies the practical application of formal grammar methods by developing systems that transform complex datasets into clear, structured narratives. Their flagship product employs a formal grammar engine (Salmon 2017) based on Abstract Categorial Grammars (ACGs, de Groote 2001), a formalism originally developed by the Sémagramme research team. ACGs provide a mathematically rigorous foundation for generating grammatically correct texts across multiple languages. This implementation demonstrates the successful commercial deployment of symbolic methods in NLG, enabling automated production of sophisticated reports while maintaining both linguistic accuracy and natural flow.

The ACG formalism rests on a minimalist mathematical foundation in simply typed linear λ-calculus (Girard 1995), supports both surface forms and derivation structures,

and provides inherent reversibility (Dymetman 1994). This theoretical foundation enables the formalism to serve effectively in both syntactic analysis and text generation tasks. ACG can express languages beyond the power of Context-Free Grammars (CFGs), while maintaining computational tractability. Like the Grammatical Framework (GF, Ranta 2004), ACG functions as a grammatical framework with grammars as first-class objects. This treatment enables grammar composition, where complex linguistic structures can be built from simpler grammatical components. The compositionality of ACG grammars provides a framework for modeling language at different linguistic levels.

Despite ACG's power as a grammatical formalism, its lack of feature structures (Carpenter 1992) presents a significant limitation. Feature structures are fundamental to modern grammar engineering, encoding morpho-syntactic properties like agreement, gender, number, and tense through efficient unification mechanisms. These structures enable systematic constraint management in syntactic and semantic analysis. The absence of direct feature structure support in ACG necessitates alternative encoding strategies that increase the complexity of linguistic modeling. In commercial applications, such as Yseop's NLG systems, this theoretical limitation introduces additional considerations for grammar implementation and maintenance.

Therefore, the goal of this research is to enhance the ACG formalism by incorporating feature structures through an extended type system. This integration maintains ACGs' mathematical foundations while making them more accessible for practical grammar engineering and improving processing efficiency through native support for linguistic properties like agreement and tense.

This thesis is structured into three main parts:

1. Part I establishes the mathematical foundations that underpin ACGs. Chapter 1 introduces λ-calculus, examining its syntax, reduction rules, and syntactic restrictions before developing the simply typed λ-calculus and its key properties. Chapter 2 presents Datalog, a declarative programming language that can elegantly encode parsing algorithms for certain classes of ACGs. The chapter covers Datalog's syntax, semantics, and optimization techniques that enable efficient program evaluation.

2. Part II introduces ACGs. Chapter 3 first defines ACGs and shows how they can represent fundamental linguistic structures like strings and trees. This chapter then demonstrates the versatility of ACGs by encoding other grammatical formalisms like CFGs. It also explores existing theoretical extensions to ACGs that enhance their expressivity through enriched type systems. Chapter 4 then presents Kanazawa's

transformation (Kanazawa 2007) that reduces the parsing of certain ACG classes to Datalog program evaluation.

3. Part III presents the main contribution: the extension of ACGs with feature structures. In Chapter 5, we first design $\lambda^{-\circ\mathcal{F}\Pi}$, a typed λ-calculus that models feature structures through enumerations, records, and dependent products. Building on this foundation, we introduce in Chapter 6 an extension of ACGs that incorporates feature structures, which we call Affix ACGs (AACGs). The name is inspired by the resemblance of this extension to a generalization of affix grammars (Koster 1991), which originally extended context-free grammars by introducing parameter-like elements in nonterminals that enforce agreement constraints across grammatical rules. We further demonstrate that AACGs are equivalent to core ACGs, thereby inheriting their desirable properties. Leveraging these theoretical results, we describe in Chapter 7 the implementation of an industrial compiler for AACGs and evaluate its performance through text generation experiments using Yseop's large-scale French grammar.

Part I

Mathematical Preliminaries

Chapter 1

λ-calculus

This chapter presents the simply typed λ-calculus (Church 1936), a computational model that treats functions as first-class citizens. Understanding λ-calculus is essential for comprehending ACGs, which is why a chapter is dedicated to it. The content of this chapter is primarily based on the book by Hindley and Seldin (Hindley 1997).

1.1 Pure λ-Calculus

1.1.1 Syntax

We start our overview of λ-calculus by introducing its syntax.

Definition 1.1 (λ-terms)**.** Let $\mathcal{X}$ be a countable set of variables. The set of λ-terms defined over $\mathcal{X}$ is defined inductively as follows:

- Every variable $x \in \mathcal{X}$ is a λ-term;
- If t_1 and t_2 are λ-terms, then $(t_1\, t_2)$ is a λ-term;
- If t is a λ-term and $x \in \mathcal{X}$, then $(\lambda x.\, t)$ is a λ-term.

The λ-term $(t_1\, t_2)$ corresponds to the application of the term t_1 to the argument t_2, while $(\lambda x.t)$ corresponds to the abstraction of the variable x in the term t.

We use the letters t, u, and v, possibly with subscripts, to denote λ-terms. Additionally, the letters x, y, and z are used to denote variables.

To streamline notation, we omit outermost parentheses in λ-terms. Application is treated as left-associative, meaning $t_1\, t_2\, t_3$ is interpreted as $((t_1\, t_2)\, t_3)$. Additionally, we

employ a compact representation for sequential abstractions, writing $\lambda x\, y\, z.\, t$ to denote $\lambda x.(\lambda y.(\lambda z.t))$.

The evaluation of the λ-term $(\lambda x.\, t)\, u$ intuitively results in replacing every occurrence of x in t with u. This replacement operation is crucial for modeling function application, as it captures the essence of passing arguments to functions. This process is known as *substitution*, a fundamental operation that must be defined carefully to avoid accidental overlaps or misinterpretations of variable scope.

Central to this definition is the distinction between *free* and *bound* variables.

Definition 1.2 (Variable occurrence)**.** An occurrence of variable x in an abstractor λx is called a *binding occurrence* of x.

Definition 1.3 (Bound and free occurrence)**.** A non-binding variable-occurrence x in a term t is said to be *bound* in t if it is in the scope of an occurrence of λx in t; otherwise it is free in t.

From this definition, the set of free variables of a λ-term t, denoted $\mathrm{FV}(t)$, can be characterized inductively as follows:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\}, \\ \mathrm{FV}(t_1\, t_2) &= \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2), \\ \mathrm{FV}(\lambda x.t) &= \mathrm{FV}(t) \setminus \{x\}. \end{aligned}$$

A λ-term is said to be *closed* if it has no free variables. Example 1.1 presents common closed λ-terms, referred to as combinators, which originate from the works in combinatory logic (H. B. Curry 1930). The combinator **I** is known as the identity combinator because it returns its argument unchanged. The combinator **K** takes two arguments and returns the first, discarding the second. The combinator **B** is used for composition of λ-terms. The combinator **C** swaps the last two arguments, altering the order of application. Lastly, the combinator ω is a self-application combinator, applying a λ-term to itself.

Example 1.1 (Common λ-terms).

$$\begin{aligned} \mathbf{I} &= \lambda x.\, x & \mathbf{K} &= \lambda x\, y.\, x \\ \mathbf{B} &= \lambda x\, y\, z.\, x\, (y\, z) & \mathbf{C} &= \lambda x\, y\, z.\, x\, z\, y \\ \omega &= \lambda x.\, x\, x \end{aligned}$$

Definition 1.4 (Substitution)**.** Define $t[x := u]$ to be the result of substituting u for each free occurrence of x in t, with appropriate renaming of bound variables to prevent variable capture (i.e., to prevent variables free in u from becoming bound in $t[x := u]$).

$$
\begin{aligned}
x[x := u] &= u && \\
y[x := u] &= y && \text{if } x \neq y, \\
t_1\, t_2[x := u] &= t_1[x := u]\, t_2[x := u] && \\
(\lambda x.t)[x := u] &= \lambda x.t && \\
(\lambda y.t)[x := u] &= \lambda y.t[x := u] && \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(u), \\
(\lambda y.t)[x := u] &= \lambda z.t[y := z][x := u] && \text{if } x \neq y \text{ and } y \in \mathrm{FV}(u) \text{ and } z \text{ is fresh.}
\end{aligned}
$$

1.1.2 Equivalence Up to Variable Renaming

Definition 1.4 shows that we may need to rename bound variables to avoid capturing free variables.

Definition 1.5 (α-conversion)**.** An *α-conversion* in a λ-term t is the process of replacing an abstraction $\lambda x.t'$ in t with $\lambda y.t'[x := y]$, where $y \notin \mathrm{FV}(t')$.

α-conversion preserves the structure of the term. Since bound variable names are placeholders with no intrinsic meaning, renaming them does not alter the term's interpretation. Thus, λ-terms that differ only in the names of their bound variables can be treated as equivalent.

Definition 1.6 (α-equivalence)**.** Two λ-terms t and u are α-equivalent, denoted $t \equiv_\alpha u$, if one can be transformed into the other by a finite sequence of α-conversions.

Example 1.2 (α-equivalences)*.*

$$
\begin{aligned}
\lambda x.\, x &\equiv_\alpha \lambda y.\, y \\
\lambda x\, y.\, x &\equiv_\alpha \lambda y\, z.\, y
\end{aligned}
$$

1.1.3 Simplifying Applications

To explore computation in λ-calculus, we formalize the step-by-step process by which λ-terms are evaluated.

Definition 1.7 (β-redex and Contractum)**.** A term of the form $(\lambda x.\, t)\, u$ is called a *β-redex*. The result of substituting u for x in t is referred to as its *contractum*.

To determine whether β-redexes are present within λ-terms, we must formalize the concept of occurrence.

Definition 1.8 (Occurrence)**.** For λ-terms u and v, the relation *u occurs in v* (or u is a subterm of v) is defined inductively as follows:

1. u occurs in v if $u = v$.
2. If u occurs in t or u occurs in t', then u occurs in $v = (t\, t')$.
3. If u occurs in t, then u occurs in $v = (\lambda x.t)$.

Definition 1.9 (β-contraction)**.** If a term t contains an occurrence of a β-redex and we replace that occurrence with its contractum, resulting in the term t', we say that *t β-contracts* to t'. This is denoted as:

$$t \rhd_{1\beta} t'.$$

While a single β-contraction evaluates a λ-term stepwise, most computations involve multiple β-contractions applied in sequence.

Definition 1.10 (β-reduction)**.** Given a term t, a finite sequence of β-contractions applied to t is called a β-reduction. If t β-reduces to u, we denote it by $t \rhd_\beta u$. Conversely, the reverse of a β-reduction is called a *β-expansion*.

The ultimate goal of β-reduction is often to reduce λ-terms to a canonical form. A term t is in *β-normal form* if it contains no β-redexes. The lemma below characterizes the structure of β-normal forms for both open and closed terms.

Lemma 1.1 (β-normal forms)**.** *Every term t in β-normal form can be expressed uniquely in the form*

$$t = \lambda x_1 \ldots x_m.\, y\, t_1 \ldots t_n \quad (m \geq 0, n \geq 0),$$

where $t_1, \ldots, t_n$ are in β-normal forms. Note that if t is a closed term then $y \in \{x_1, \ldots, x_m\}$.

So far, we have not specified how the β-redex to be reduced is chosen. While β-reduction permits any redex to be contracted at any time, it is often useful to adopt a strategy that determines which redex should be reduced first at each step. A common

strategy is *leftmost reduction*, where, at each step, the redex that appears furthest to the left and outermost in the term's structure is always contracted first.

While strategies such as leftmost reduction offer a systematic method for selecting the next β-redex to contract, the choice of strategy does not influence the final outcome. Intuitively, this means that no matter which path of reductions we take, we can always "merge" different reduction sequences: if we can reach two different terms from the same starting point, there is always a way to continue reducing both terms until they meet at a common result. This independence is formalized by the Church-Rosser theorem, a cornerstone of λ-calculus that establishes the confluence property of β-reduction.

Theorem 1.1 (Church-Rosser (Church 1936))**.** *If $t \rhd_\beta u$ and $t \rhd_\beta v$, then there exists a term w such that:*

$$u \rhd_\beta w \quad \textit{and} \quad v \rhd_\beta w.$$

The relationship between β-reduction and β-expansion gives rise to the concept of equivalence between λ-terms.

Definition 1.11 (β-conversion)**.** We say that two terms t and u are β-equivalent, denoted by $t =_\beta u$, if t can be transformed into u through a finite sequence of β-reductions and β-expansions.

1.1.4 Eliminating Redundant Abstractions

In addition to β-reduction, another rewriting rule called eta-contraction can be introduced to further simplify expressions. η-contraction removes unnecessary function abstractions, reducing a function of the form $\lambda x.(f\ x)$ to f, provided that the variable x does not appear in f. Formally, the eta-contraction rule is defined as follows:

$$\lambda x.(t\,x) \rhd_{1\eta} t \quad \text{if } x \notin \mathrm{FV}(t) \tag{η}$$

This simplification is important because it equates functions that behave identically, making expressions more concise and easier to manipulate. The concepts of η-reduction ($\rhd_\eta$), η-expansion, and η-conversion ($=_\eta$) are defined analogously to their β counterparts.

Example 1.3 (η-contraction). Consider the λ-term $t = \lambda y.\, \lambda x.\, (f\ x)$, where $x \notin \mathrm{FV}(f)$. In this λ-term, the abstraction $\lambda x.\, (f\ x)$ is the inner most abstraction, and it takes an argument x to produce $(f\ x)$.

Applying η-contraction to the term t yields:

$$\lambda y.\, \lambda x.\, f\ x \rhd_{1\eta} \lambda y.\, f.$$

The validity of the Church-Rosser theorem for β-reduction also extends to η-reduction. Crucially, the theorem also holds for the combined relation, for which we use the notation $\rhd_{\beta\eta}$.

While reductions can be arbitrarily interleaved, a more structured simplification strategy is often desirable. The following theorem provides the formal basis for such a strategy by allowing for the postponement of all η-reductions.

Theorem 1.2 (η-Postponement)**.** *If $t \rhd_{\beta\eta} v$ then there exists a term u such that $t \rhd_\beta u$ and $u \rhd_\eta v$.*

Proof. See Nederpelt (1973), Theorem 7.28. □

1.1.5 Syntactic Restrictions

λ-terms can be classified into distinct classes based on their syntactic properties.

Definition 1.12 ($\lambda\mathbf{I}$-terms)**.** A λ-term t is called a $\lambda\mathbf{I}$-term if, for every subterm of the form $\lambda x.\, u$, the body u contains at least one free occurrence of the variable x.

These terms are named after the identity combinator $\mathbf{I} = \lambda x.x$, which exhibits the property that every bound variable must have at least one free occurrence in the term's body. In contrast, general λ-terms are referred to as $\lambda\mathbf{K}$-terms, named after the combinator $\mathbf{K} = \lambda x.\lambda y.x$, which demonstrates the ability to ignore an argument.

Definition 1.13 (Affine λ-term)**.** A λ-term t is said to be *affine* if for any subterm of t of the form $u\, v$, we have $\mathrm{FV}(u) \cap \mathrm{FV}(v) = \emptyset$.

Definition 1.14 (Linear λ-term)**.** A λ-term t is said to be *linear* if the following conditions hold:

1. For any subterm of t of the form $\lambda x.\, u$, we have $x \in \mathrm{FV}(u)$;
2. For any subterm of t of the form $u\, v$, we have $\mathrm{FV}(u) \cap \mathrm{FV}(v) = \emptyset$.

Linear λ-terms form a strict subset of both $\lambda\mathbf{I}$-terms and affine λ-terms. The first inclusion holds because condition (1) ensures that every bound variable has at least one free occurrence in the term's body. The second inclusion follows from condition (2), which is precisely the defining property of affine λ-terms.

A β-contraction, $t[x := u]$, is said to *cancel* u if x does not appear freely in t. Conversely, it *duplicates* u if x appears freely in t at least twice.

A β-reduction is termed *non-duplicating* if none of its contractions result in duplication, and *non-canceling* if none result in cancellation.

The nature of β-contraction differs across subclasses of λ-terms. In $\lambda\mathbf{I}$-terms, β-contractions are non-canceling. Linear λ-terms, being more restrictive, enforce both non-canceling and non-duplicating β-contractions. Affine λ-terms relax the non-canceling requirement but remain non-duplicating.

1.2 Simply Typed λ-Calculus

1.2.1 Types

Simply typed λ-calculus restricts λ-calculus by assigning *simple types* to terms.

Definition 1.15 (Simple Types)**.** Let A be a set of atomic types. The set of simple types $\mathcal{T}(A)$ defined over A, is defined inductively as follows:

- Every atomic type $a \in A$ is a simple type.
- If α and β are types, then $(\alpha \to \beta)$ is a simple type.

We use α, β, and γ as metavariables to range over types. By convention, the arrow $\to$ associates to the right. That is, $\alpha \to \beta \to \gamma$ is interpreted as $\alpha \to (\beta \to \gamma)$.

There are two primary approaches to typing in λ-calculus: *Church-style* and *Curry-style*. *Church-style typing*, named after Alonzo Church, involves explicit type annotations within the λ-terms. This approach embeds type information directly into the λ-terms, making types an integral part of the lambda expressions themselves. *Curry-style typing*, named after Haskell Curry, assigns types to λ-terms using an external typing system based on inference rules, without embedding type annotations within the lambda expressions. This method, known as *type assignment*, relies on automatically determining types through a type inference process. In this exposition, we define the simply typed calculus using Curry-style typing, which we denote by $\lambda^{\to}$.

Definition 1.16 (Type Assignment)**.** A *type assignment* is an expression of the form $t : \alpha$, where t is a λ-term, and α is a type. We call t the *subject* of the type assignment and α its *predicate*.

Definition 1.17 (Typing Context)**.** A *typing context* Γ is a finite, possibly empty, set of type assignments of the form $\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$, in which the variables $x_1, \ldots, x_n$ are distinct, and $\alpha_1, \ldots, \alpha_n$ are simple types.

The *subjects* of Γ are the variables appearing in the context. Formally, we define:

$$\text{Subjects}(\Gamma) = \{x_1, \ldots, x_n\}.$$

The distinctness of variables ensures that no variable is the subject of more than one assignment in the context.

Definition 1.18 (Type Assignment Formula)**.** For any typing context Γ, term t, and simple type α, the triple (Γ, t, α) is called a *type assignment formula* and is written as:

$$\Gamma \vdash t : \alpha.$$

When the context Γ is empty, we write $\vdash t : \alpha$. To simplify notation, $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash t : \alpha$ is used as a shorthand for $\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\} \vdash t : \alpha$. Similarly, if Γ is extended with $y_1 : \beta_1, \ldots, y_n : \beta_n$, we write $\Gamma, y_1 : \beta_1, \ldots, y_n : \beta_n \vdash t : \alpha$ instead of $\Gamma \cup \{y_1 : \beta_1, \ldots, y_n : \beta_n\} \vdash t : \alpha$.

System 1 introduces the simply typed λ-calculus $\lambda^{\to}$, specifying its syntax, notations, typing rules, and evaluation semantics.

System 1 (Simply typed λ-calculus, $\lambda^{\to}$)**.**

Syntax:

$$\begin{aligned}
\mathcal{T}(A) &:= && \text{simple type:} \\
&\quad A && \textit{atomic type} \\
&\quad \mid \mathcal{T}(A) \to \mathcal{T}(A) && \textit{implicative type} \\
T &:= && \lambda\text{-term:} \\
&\quad \mathcal{X} && \textit{variable} \\
&\quad \mid \lambda\mathcal{X}.\, T && \textit{abstraction} \\
&\quad \mid (T\ T) && \textit{application}
\end{aligned}$$

Notations:

$$
\begin{aligned}
&a \in A \\
&\alpha, \beta, \gamma \in \mathcal{T}(A) \\
&x \in \mathcal{X} \\
&t, u, v \in T
\end{aligned}
$$

Typing:

$$\frac{}{\Gamma, x : \alpha \vdash x : \alpha}\ \text{VAR-TERM} \qquad \frac{\Gamma_1 \vdash t : \alpha \to \beta \qquad \Gamma_2 \vdash u : \alpha}{\Gamma_1 \cup \Gamma_2 \vdash t\, u : \beta}\ \text{APP-TERM}$$

$$\frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.\, t : \alpha \to \beta}\ \text{ABSTR-TERM}$$

Evaluation:

$$
\begin{aligned}
&(\lambda x.\, t)\, t' \rhd_{1\beta} t[x := t'] \\
&\lambda x.(t\ x) \rhd_{1\eta} t \quad \text{if } x \notin \mathrm{FV}(t)
\end{aligned}
$$

Reasoning about the types of λ-terms in this system utilizes the three inference rules shown in System 1: a single axiom for variables and two rules for application and abstraction. Because these rules closely follow the syntactic structure of λ-terms, the construction of a type assignment proceeds in a syntax-directed manner. This process generates a typing derivation.

Definition 1.19 (Typing Derivation)**.** A *typing derivation*, denoted by $\mathscr{D}$, is a tree composed of type assignment formulas, where:

- *Leaves* correspond to instances of the variable axiom.
- Each *internal node* represents a type assignment formula derived as the conclusion of applying one of the inference rules to its immediate predecessor nodes.

Figure 1.1 shows an example of such a derivation for the combinator **B**.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha \to \beta \vdash x : \alpha \to \beta} \quad \dfrac{\dfrac{}{y : \gamma \to \alpha \vdash y : \gamma \to \alpha} \quad \dfrac{}{z : \gamma \vdash z : \gamma}}{y : \gamma \to \alpha, z : \gamma \vdash y\, z : \alpha}}{x : \alpha \to \beta, y : \gamma \to \alpha, z : \gamma \vdash x\,(y\, z) : \beta}}{x : \alpha \to \beta, y : \gamma \to \alpha \vdash \lambda z.\, x\,(y\, z) : \gamma \to \beta}}{x : (\alpha \to \beta) \vdash \lambda y\, z.\, x\,(y\, z) : (\gamma \to \alpha) \to (\gamma \to \beta)}}{\vdash \lambda x\, y\, z.\, x\,(y\, z) : (\alpha \to \beta) \to (\gamma \to \alpha) \to (\gamma \to \beta)}$$

FIGURE 1.1: Typing Deduction for the Combinator **B**.

The primary motivation for introducing types is to exclude problematic or ill-behaved λ-terms. For instance, consider the application of the combinator $\omega = \lambda x.\,(x\, x)$ to itself, $\omega\,\omega$. In the untyped λ-calculus, attempting to obtain a β-normal form for the application $\omega\,\omega$ results in non-termination.

In contrast, the simply typed λ-calculus prevents such ill-behaved terms from being constructed in the first place. To illustrate why, suppose we try to assign a type to the term $\omega = \lambda x.x\, x$. According to the type assignment rules for a λ-abstraction, we must be able to assign a type to its body, $(x\, x)$, under the assumption that x has some type α.

For the self-application $x\, x$ to be well-typed, x must simultaneously have a function type of the form $\sigma \to \tau$ and be the argument to that function, requiring it to have the type σ. Since x is the same term in both positions, its type, α, must be identical to both σ and $\sigma \to \tau$. This creates the paradoxical requirement that a type must be equivalent to a function type that takes itself as an argument:

$$\alpha = \alpha \to \tau$$

In other words, we would need a type that equals its own function space, which is impossible in simply typed systems. No such type α can be constructed in the simply typed λ-calculus. Because ω itself cannot be well-typed, it is not a valid term in this system, and consequently, any application involving it, such as $\omega\,\omega$, is also not a valid term.

1.2.2 Properties of Typing

We now shift our focus to exploring some fundamental properties of typing. When performing inductive proofs on typing derivations, it is often necessary to introduce fresh variables into the existing context. It is essential to ensure that these newly introduced

variables do not interfere with the well-typedness of terms that have already been proven correct within a smaller context. This requirement is formalized by the following lemma:

Lemma 1.2 (Weakening)**.** *If $\Gamma \vdash t : \alpha$ is a derivable type assignment formula in $\lambda^{\rightarrow}$ (System 1) and $x \notin \mathit{Subjects}(\Gamma)$, then $\Gamma, x : \beta \vdash t : \alpha$ is also derivable.*

Proof. The proof is standard and proceeds by induction on the structure of the typing derivation for $\Gamma \vdash t : \alpha$. □

While the Lemma 1.2 addresses the static stability of typing under an expanded context, it is crucial to establish its dynamic stability during computation. This brings us to another fundamental property: ensuring the type of a λ-term remains unchanged throughout its evaluation. This property, known as *subject reduction*, guarantees that if a term has a certain type, any reduction will result in a term of the exact same type.

Theorem 1.3 (Subject Reduction (Haskell B. Curry, Feys, and Craig 1959))**.** *If $\Gamma \vdash t : \alpha$ and $t \rhd_\beta t'$, then $\Gamma \vdash t' : \alpha$.*

It is equally important to examine the reverse direction: whether a λ-term prior to reduction retains the same type as the reduced λ-term. This, however, does not hold universally; we must impose assumptions about the kind of β-reductions involved.

Theorem 1.4 (Subject Expansion)**.** *If $\Gamma \vdash t' : \alpha$ and $t \rhd_\beta t'$ through a non-erasing and non-duplicating β-reduction, then $\Gamma \vdash t : \alpha$.*

Proof. See Seldin (1968), §3D Theorem 3. □

While Theorem 1.3 and Theorem 1.4 address type preservation during single steps of reduction, another fundamental concern in type theory is the termination of computation. This leads to the distinction between *weak normalization* and *strong normalization*. Weak normalization guarantees that a typable λ-term will eventually reach a β-normal form, though it doesn't specify how. Strong normalization, a more powerful property, asserts that all β-reduction sequences for a typable λ-term are finite. This is formalized in the following theorem:

Theorem 1.5 (Strong Normalization (Tait 1967))**.** *For any typable term t, every β-reduction starting from t will terminate after a finite number of steps.*

1.2.3 Principal Typing

A λ-term in simply typed λ-calculus can have an infinite set of assignable types, but these are typically derived from a single, most general type: *the principal type*. Broadly speaking, identifying the existence of such a type and developing an algorithm to determine it are fundamental aspects of designing type systems.

This section begins by refining the definition of simple types to include type variables, which serve as placeholders for types, allowing general type structures to be expressed.

Definition 1.20 (Parametric Types)**.** Let A be a set of atomic types and P a set of type variables. The set of *parametric types*, denoted $\mathcal{T}(A, P)$, is defined inductively as follows:

- Every atomic type $a \in A$ is a parametric type.
- Every type variable $p \in P$ is a parametric type.
- If α and β are parametric types, then $(\alpha \to \beta)$ is also a parametric type.

A parametric type is considered *closed* if it does not contain any type variables. Conversely, a parametric type is referred to as *open* if it consists solely of type variables.

To formalize the instantiation of a parametric type, we define a mechanism for replacing type variables in a parametric type.

Definition 1.21 (Type Substitution)**.** A *type substitution*, σ, is a mapping of the form $\sigma = [p_1 := \mathcal{T}_1, \ldots, p_n := \mathcal{T}_n]$, where $p_1, \ldots, p_n$ are distinct type variables and $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are arbitrary parametric types.

For any type α, the application of the substitution σ to α, denoted as $\alpha\sigma$, is defined inductively as follows:

$$
\begin{aligned}
a\sigma &= a \\
p\sigma &= \begin{cases} \mathcal{T}_i & \text{if } p = p_i \text{ for some } i \in \{1, \ldots, n\} \\ p & \text{otherwise.} \end{cases} \\
(\beta \to \gamma)\sigma &= \beta\sigma \to \gamma\sigma.
\end{aligned}
$$

The type $\alpha\sigma$ is called an *instance* of α.

For a context $\Gamma = \{x_1 : \tau_1, \ldots, x_k : \tau_k\}$, applying a type substitution σ means applying it to each type binding: $\Gamma\sigma = \{x_1 : \tau_1\sigma, \ldots, x_k : \tau_k\sigma\}$. A context Γ' is an instance of Γ if

there exists a type substitution σ such that $\Gamma\sigma = \Gamma'$. Finally, a deduction $\mathscr{D}'$ is an instance of $\mathscr{D}$ if there exists a type substitution σ such that applying σ to every type, context, and judgment in $\mathscr{D}$ yields $\mathscr{D}'$.

Definition 1.22 (Principal Typing and Principal Deduction)**.** Let t be a term in a typed λ-calculus with context Γ, and let α be a parametric type.

1. We say that $\Gamma \vdash t : \alpha$ is a *principal typing* if, for every other type assignment $\Gamma' \vdash t : \beta$, $\Gamma' \vdash t : \beta$ is an instance of $\Gamma \vdash t : \alpha$. That is, there exists a type substitution σ such that $\Gamma' = \Gamma\sigma$ and $\beta = \alpha\sigma$.

2. Similarly, a deduction $\mathcal{D}$ deriving $\Gamma \vdash t : \alpha$ is *principal* if, for every other deduction $\mathcal{D}'$ deriving $\Gamma' \vdash t : \beta$, $\mathscr{D}'$ is an instance of $\mathscr{D}$.

We denote the existence of a principal deduction for the principal typing $\Gamma \vdash t : \alpha$ by $\Gamma \vdash_{PT} t : \alpha$.

Theorem 1.6 (Principal Type theorem (Hindley 1969))**.** *In $\lambda^{\to}$, each term that can be typed possesses a principal type and a principal deduction. Moreover, there exists an algorithm capable of determining whether a given term t is typable in $\lambda^{\to}$ and outputting its principal deduction.*

This result underscores the importance of algorithms for computing principal types and deductions. However, we will not explore the implementation details, as they are not strictly necessary for understanding the remainder of this thesis.

Chapter 2

Datalog

In this chapter, we present a concise overview of Datalog, a declarative logic programming language widely employed as a query language for deductive databases. Since Datalog can be used to implement ACG parsing for specific classes of ACGs, this chapter focuses on providing the necessary background. For a more comprehensive treatment of Datalog, readers are encouraged to consult Abiteboul, Hull, and Vianu (1995), Ceri, Gottlob, and Tanca (1990b), and Ullman (1988, 1990).

Essentially, Datalog programs consist of logical rules that operate on an initial collection of known information. These rules specify how to systematically derive new pieces of information or conclusions, and define how different elements within the data are connected. Unlike relational algebra, Datalog provides a recursive mechanism that allows rules to define predicates that reference themselves in their conditions, either directly or through a chain of dependencies, enabling iterative computations of arbitrary depth. For example, recursion in Datalog enables the computation of transitive closures, a property that standard relational algebra is not equipped to model.

Syntactically, Datalog is a subset of Prolog, consisting of terms, predicates, and rules. However, unlike Prolog, Datalog does not support function symbols, which means it lacks the ability to construct compound terms with functors. This restriction, while limiting expressiveness in some cases, ensures that all queries terminate.

The language Datalog is at the fruitful intersection of database theory and logic programming. We will follow mainly the terminology of logic programming. However, we will use the notions of relations and databases where appropriate to explain some concepts.

2.1 Syntax

2.1.1 Primitives and Operations

Definition 2.1 (Datalog Terms)**.** Let $\mathcal{X}$ be a countable set of variables and $\mathcal{C}$ be a countable set of constants. The set of Datalog terms, $\mathcal{T}$, is defined as the union of $\mathcal{X}$ and $\mathcal{C}$.

Definition 2.2 (Atom)**.** Let $\mathcal{P}$ be a set of predicate symbols, where each symbol $p \in \mathcal{P}$ is assigned a nonnegative integer $ar(p)$, called its *arity*. A *Datalog atom* is an expression of the form $p(\vec{t})$, where p has arity n, and $\vec{t} = (t_1, \ldots, t_n)$ is an n-tuple of terms, called the *arguments* of the atom.

Definition 2.3 (Rule)**.** A *Datalog rule* r is an expression of the form

$$[r] \quad p_0(\vec{t}_0) \leftarrow p_1(\vec{t}_1), \ldots, p_n(\vec{t}_n).$$

for $i \in [0, n]$, each $p_i(\vec{t}_i)$ is an *atom*. The *head* of r is the single atom $p_0(\vec{t}_0)$. The *body* of r is the conjunction of the atoms $p_1(\vec{t}_1), \ldots, p_n(\vec{t}_n)$.

Definition 2.4 (Fact)**.** A *Datalog fact* is a single Datalog atom that represents an unconditionally true statement. It has the following form:

$$p(\vec{t}).$$

A fact $p(\vec{t})$. is *grounded* if all terms in its argument vector $\vec{t}$ are constants.

Having established the syntax of Datalog rules, we now introduce the dynamic operations required to instantiate them.

Definition 2.5 (Binding)**.** A *binding* is a mapping of a variable to a Datalog term, denoted as $x \mapsto t$.

Definition 2.6 (Datalog Substitution)**.** A *substitution* in Datalog is a partial function $\theta : \mathcal{X} \to \mathcal{T}$ that assigns Datalog variables to Datalog terms. Formally, we can express θ as:

$$\theta = \{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n\},$$

where $x_i \in \mathcal{X}$ and $t_i \in \mathcal{T}$ for each $i \in [1, n]$.

A substitution θ is *grounded* if every term $t_1, \ldots, t_n$ in the mappings is a constant.

For any substitution θ, we assume that $x_i \neq t_i$ for all i. This ensures that no variable maps to itself, guaranteeing that substitutions represent meaningful transformations of expressions.

Definition 2.7 (Substitution Application)**.** A substitution θ is applied to terms, predicates, and rules according to the following definitions:

1. The application of θ to a term t, denoted as $t\theta$, is defined as follows:
   - If t is a variable x and $x \mapsto t' \in \theta$, then $t\theta = t'$.
   - If t is a variable x and $x \mapsto t' \notin \theta$, then $t\theta = t$.
   - If t is a constant, then $t\theta = t$.
2. The application of θ to a predicate $p(t_1, \ldots, t_n)$, denoted by $p(t_1, \ldots, t_n)\theta$, results in the predicate $p(t_1\theta, \ldots, t_n\theta)$, where θ is applied to each term t_i individually.
3. Let r be a rule of the form:

$$[r] \quad p_0(\vec{t_0}) \leftarrow p_1(\vec{t_1}), \ldots, p_n(\vec{t_n}).$$

   The application of θ to r, denoted by $r\theta$, produces the rule:

$$[r\theta] \quad p_0(\vec{t_0})\theta \leftarrow p_1(\vec{t_1})\theta, \ldots, p_n(\vec{t_n})\theta.$$

   where θ is applied to each atom in the rule.

Definition 2.8 (Rule Instance)**.** Let r and r' be rules. If there exists a substitution θ such that $r\theta = r'$, then r' is said to be an *instance* of r.

2.1.2 Program

In Datalog, a *program* (P) is simply a finite set of rules. The input to a program, its *instance* (**I**), consists of a set of *grounded facts*. Example 2.1 illustrates a classic Datalog program used to compute the transitive closure of a graph, along with a corresponding input. In this program, the predicate symbol G represents a direct edge from one node to another, while the symbol T signifies that a path exists between two nodes. Rule r_1 provides the base case, stating that any direct edge is itself a path. Rule r_2 then recursively computes longer paths by chaining an edge with an existing path, ultimately producing the full transitive closure.

Example 2.1 (Transitive Closure). Let P be the following Datalog program, which computes the transitive closure of a graph:

$$[r_1] \quad T(x,y) \leftarrow G(x,y) \qquad \text{(Base case: direct edges in } G\text{)}$$
$$[r_2] \quad T(x,z) \leftarrow G(x,y), T(y,z) \qquad \text{(Recursive case: transitive closure)}$$

Let **I** be an instance consisting of the following facts, representing the edges of a directed graph:

$$G(3,4).$$
$$G(4,5).$$
$$G(5,6).$$

The set of predicate symbols in a Datalog program, P, is partitioned into two disjoint sets: *extensional* and *intensional* predicates. Extensional predicates are those defined over the input database instance. Syntactically, they are restricted to appear only in the bodies of rules and never in a rule head. In contrast, intensional predicates are those defined by the program's rules themselves. They must appear in the head of at least one rule and represent the relations computed by the program as its output.

Definition 2.9 (Program Schemas)**.** Let P be a Datalog program. The *extensional schema* of P, denoted $\mathsf{EDB}(P)$, is the set of all extensional predicates, each associated with its arity. Similarly, the *intensional schema*, denoted $\mathsf{IDB}(P)$, is the set of all intensional predicates and their respective arities. The complete *schema* of P, denoted $\mathsf{SCH}(P)$, is the union of its extensional and intensional schemas.

To illustrate, let us consider Example 2.1. In this program, the predicate G is extensional, as it represents the input graph edges provided as facts. The predicate T is intensional, as it is recursively defined by the program's rules. Consequently, for this program, the extensional schema is $\mathsf{EDB}(P) = \{G^2\}$, the intensional schema is $\mathsf{IDB}(P) = \{T^2\}$, and the program schema is $\mathsf{SCH}(P) = \{G^2, T^2\}$.

To prevent infinite computations during evaluation, we impose a syntactic constraint on programs known as *safety*.

Definition 2.10 (Safe Program)**.** A Datalog program P is *safe* if, for every rule r in P, every variable x that appears in the head of r also appears in the body of r.

If a variable were to appear in the head but not the body, its value would be unconstrained, potentially leading to the generation of an infinite number of new facts.

2.2 Semantics

2.2.1 Approaches to Datalog Semantics

Datalog programs may be interpreted using three main semantic approaches: *least fixed-point*, *model-theoretic*, and *proof-theoretic semantics*. Each method provides distinct insights into the operational and theoretical aspects of Datalog.

The least fixed-point semantics approach iteratively applies program rules to converge upon a stable set of facts, where no additional facts can be derived.

The model-theoretic semantic approach provides a framework for interpreting Datalog programs by examining their possible models. In this framework, Datalog rules are viewed as logical statements that establish relationships between facts. The main goal is to identify model-specific interpretations that satisfy all the program's rules.

Central to this approach is the concept of the least model, which is the smallest set of facts that meets the program's logic without including any superfluous information. Focusing on the least model ensures that the interpretation is both sound (only true statements are derived) and complete (all true statements are derived) with respect to the Datalog program. This minimal model contains exactly the essential facts required to satisfy the program's conditions.

The proof-theoretic semantics introduces a different angle by focusing on the provability of facts.

Definition 2.11 (Proof tree)**.** A proof tree for a given fact f, based on the input $\mathbf{I}$ and the program P, is a finite tree where:

1. Every node in the tree is labeled with a grounded fact;
2. The root node is labeled with the fact f.
3. Each leaf of the tree is labeled with a fact belonging to the instance $\mathbf{I}$;
4. For every non-leaf node f_0, there is an instance of a rule $r : f_0 \leftarrow f_1, \ldots, f_n$ and each f_i is a child node of f_0.

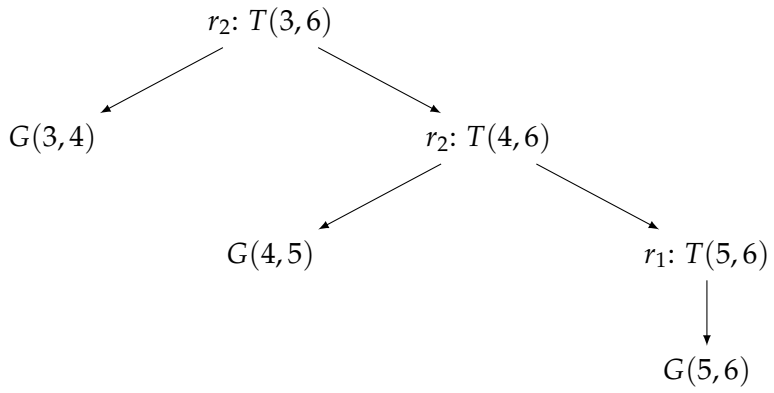


FIGURE 2.1: Proof tree for the fact $T(3,6)$ using Example 2.1.

According to proof-theoretic semantics, a fact is included in the program's output when it can be derived through at least one proof tree. To construct these proof trees, two methodologies can be employed: the *bottom-up* approach and the *top-down* approach.

Bottom-Up Approach This method treats rules as mechanisms that generate new facts from existing ones. Starting with the known facts in the instance $\mathbf{I}$, rules are applied iteratively to derive additional facts, progressively building the proof tree from the ground up.

Top-Down Approach This strategy begins with the fact that needs to be proven. It decomposes the target fact into subgoals, identifies the premises required for each subgoal, and recursively constructs the proof tree by addressing these subproofs until reaching the foundational facts present in $\mathbf{I}$.

To illustrate these approaches, consider the proof tree for the fact $T(3,6)$ in Figure 2.1.

A bottom-up construction would begin by deriving fundamental facts like $G(5,6)$. Subsequently, using rule r_1, it derives $T(5,6)$, and then with rule r_2, it obtains $T(4,6)$. Finally, applying rule r_2 again, it concludes with $T(3,6)$.

Conversely, the top-down construction for the fact $T(3,6)$ starts by applying rule r_2, which decomposes $T(3,6)$ into the subgoals $G(3,4)$ and $T(4,6)$. To prove $T(4,6)$, rule r_2 is again applied, breaking it down into $G(4,5)$ and $T(5,6)$. Finally, to establish $T(5,6)$, rule r_1 is used, which relies on the base fact $G(5,6)$ from the input instance $\mathbf{I}$.

2.2.2 Evaluation Algorithms

We present the most common bottom-up algorithms for evaluating Datalog programs, grounded in fixpoint theory, which is well-suited for computational implementation.

Definition 2.12 (Immediate consequence)**.** Given a program P and an instance $\mathbf{I}$ defined over the schema of P, A fact f is said to be an immediate consequence of P and $\mathbf{I}$ if:

1. the fact f belongs to $\mathbf{I}$;
2. or there exist an instantiation of a rule r in P, of the form $[r] \quad f \leftarrow f_1, \ldots, f_n$ where each f_i belongs to $\mathbf{I}$.

For each intensional predicate p in P, the *immediate consequence operator* $c_p(\mathbf{I})$ is defined by obtaining the immediate consequence to all the rules in P that have p as their head.

The immediate consequence operator for the entire program P, denoted $\mathcal{C}_P(\mathbf{I})$, is then the union of the immediate consequence operators for all intensional predicates in P.

Note that this definition is abstract, as the immediate consequence can be formalized either through a substitution-based approach or using relational algebra techniques.

Algorithm 1, referred to as the *naive evaluation algorithm*, iteratively applies the immediate consequence operator, to compute the set of derivable facts from a given Datalog program and an initial instance. The process starts with an empty set of facts and repeatedly applies the operator until no new facts can be derived, signifying that a fixpoint has been reached. Given an instance $\mathbf{I}$, we denote the output of a Datalog program P by $P(\mathbf{I})$.

Algorithm 1: Naive evaluation of P

Input: Datalog program P with $\mathsf{IDB}(P) = \{p_1, \ldots, p_m\}$ and input instance I
Output: $P(I)$
for *every* $p \in \mathit{IDB}(P)$ **do**
  $p^0 := \emptyset$
end
$i := 0$;
repeat
  $i := i + 1$;
  $p^i := c_p(\mathbf{I} \cup p_1^{i-1} \cup \ldots \cup p_m^{i-1})$
until *for every* p, $p^i = p^{i-1}$;

Example 2.2 (Naive Evaluation of Transitive Closure). Let P be the transitive closure program and $\mathbf{I}$ the instance as defined in Example 2.1. The steps of the naive evaluation are as follows:

$$
\begin{aligned}
T^0 &= \emptyset \\
T^1 &= \{T(3,4), T(4,5), T(5,6)\} \\
T^2 &= \{T(3,4), T(4,5), T(5,6), T(3,5), T(4,6)\} \\
T^3 &= \{T(3,4), T(4,5), T(5,6), T(3,5), T(4,6), T(3,6)\} \\
T^4 &= T^3
\end{aligned}
$$

The evaluation halts when $T^i = T^{i-1}$, which occurs at iteration $i = 4$. Therefore, the transitive closure of G is:

$$P(\mathbf{I}) = \{T(3,4), T(4,5), T(5,6), T(3,5), T(4,6), T(3,6)\}$$

In the naive evaluation of the transitive closure program presented in Example 2.2, each iteration applies all rules to the entire set of known facts, including those that were already derived in previous steps. For example, once $T(3,5)$ and $T(4,6)$ are established in T^2, these facts are redundantly reconsidered in subsequent iterations T^3 and T^4.

This repetition leads to unnecessary computations, as the algorithm repeatedly processes facts that have already been derived. To mitigate this inefficiency, we introduce the following rule transformation to target only the newly inferred facts from the most recent iteration.

Definition 2.13 (Rule Transformation)**.** Let P be a Datalog program. Consider a rule in P of the form:

$$p(\vec{t}) \leftarrow e_1(\vec{u_1}), \ldots, e_n(\vec{u_n}), p_1(\vec{t_1}), \ldots, p_m(\vec{t_m}),$$

where e_k are extensional predicates, and p_j are intensional predicates. For each $j \in [1, m]$ and for each iteration $i \geq 1$, the following rule is constructed:

$$temp_p^{i+1}(\vec{t}) \leftarrow e_1(\vec{u_1}), \ldots, e_n(\vec{u_n}),\ p_1^i(\vec{t_1}), \ldots, p_{j-1}^i(\vec{t_{j-1}}),\ \Delta_{p_j}^i(\vec{t_j}),\ p_{j+1}^i(\vec{t_{j+1}}), \ldots, p_m^i(\vec{t_m}).$$

where:

- $\Delta_{p_j}^i$ is the predicate representing the *newly inferred facts* for the IDB predicate p_j during the i-th iteration.

- $temp_p^{i+1}$ is the predicate used to store intermediate results for p during the $(i+1)$-th iteration.

Let P be a Datalog program. For an intensional predicate p, we denote by P_p^i the set of rules generated for p at iteration i. Specifically, this set contains all rules defining $temp_p^{i+1}$ created by the transformation central to the *seminaive algorithm*.

Rather than using a single, general rule for an intensional predicate p, this algorithm generates a collection of more specialized rules for each iteration i. The fundamental insight behind this approach is that any new fact can only be derived if at least one of the intensional predicates in the body of a rule contributes a fact that was newly discovered in the immediately preceding iteration.

Algorithm 2: Seminaive Algorithm (Abiteboul, Hull, and Vianu 1995)

Input: Datalog program P and input instance I
Output: $P(I)$

Set P' to be the rules in P with no IDB predicate in the body;
$p^0 \leftarrow \emptyset$, for each predicate $p \in \mathsf{IDB}(P)$;
$\Delta_p^1 \leftarrow P'(I)$, for each predicate $p \in \mathsf{IDB}(P)$;
$i \leftarrow 1$;
repeat
  for *each predicate* $p \in \mathit{IDB}(P)$, *where* $p_1, \ldots, p_n$ *are the IDB predicates involved in rules defining* p **do**
    $p^i \leftarrow p^{i-1} \cup \Delta_p^i$;
    $\Delta_p^{i+1} \leftarrow \mathcal{C}_{P_p^i}(\mathbf{I} \cup p_1^{i-1} \cup \ldots \cup p_n^{i-1} \cup p_1^i \ldots \cup p_n^i \cup \Delta_{p_1}^i \cup \ldots \cup \Delta_{p_n}^i) \setminus p^i$;
  end
  $i \leftarrow i+1$;
until $\Delta_p^i = \emptyset$ *for each predicate* $p \in \mathit{IDB}(P)$;
$p \leftarrow p^i$ for each predicate $p \in \mathsf{IDB}(P)$;

Example 2.3 (Seminaive Evaluation of Transitive Closure). Let P be the transitive closure

program and **I** the instance as defined in Example 2.1. The steps of the seminaive evaluation are as follows:

$$
\begin{aligned}
T^0 &= \emptyset \\
\Delta_T^1 &= \{T(3,4), T(4,5), T(5,6)\} \\
T^1 &= T^0 \cup \Delta_T^1 = \{T(3,4), T(4,5), T(5,6)\} \\
\Delta_T^2 &= \{T(3,5), T(4,6)\} \\
T^2 &= T^1 \cup \Delta_T^2 = \{T(3,4), T(4,5), T(5,6), T(3,5), T(4,6)\} \\
\Delta_T^3 &= \{T(3,6)\} \\
T^3 &= T^2 \cup \Delta_T^3 = \{T(3,4), T(4,5), T(5,6), T(3,5), T(4,6), T(3,6)\} \\
\Delta_T^4 &= \emptyset
\end{aligned}
$$

The evaluation halts when $\Delta T^4 = \emptyset$. Therefore, the transitive closure of G is:

$$P(\mathbf{I}) = \{T(3,4), T(4,5), T(5,6), T(3,5), T(4,6), T(3,6)\}$$

The seminaive algorithm shows better performance, but in general there is still room to eliminate unnecessary computations through static analysis of programs. This approach leverages the concept of dependencies between predicates by constructing a *precedence graph*.

Definition 2.14 (Precedence Graph)**.** Given a datalog program P, the precedence graph of P, G_P is defined as follows:

1. Each IDB predicate in P is represented as a node.
2. An edge is drawn from node p to node p' if p appears in the body of a rule where p' is the head predicate.

Analyzing the structure and behavior of Datalog programs fundamentally relies on recursion.

Definition 2.15 (Recursion and Mutual Recursion)**.** Let P be a Datalog program and G_P its precedence graph.

1. The program P is *recursive* if its precedence graph G_P contains at least one directed cycle.

2. Two predicates p and p' are *mutually recursive* if they are identical ($p = p'$) or if both belong to the same directed cycle in G_P. This mutual recursion establishes an equivalence relation among the intensional predicates of P.

Note that each equivalence class under this relation corresponds to a strongly connected component of the precedence graph G_P. This sets the stage for the *improved seminaive algorithm*, which improves upon the basic seminaive approach by leveraging these equivalence classes.

The core strategy of this algorithm is to break the overall evaluation into a sequence of smaller stages. Instead of applying the semi-naive method to the entire program at once, it first performs a static analysis to determine an optimal evaluation order. This analysis phase begins by partitioning the program's intensional predicates into their respective strata (the strongly connected components). A condensation graph is then constructed by representing each stratum as a single node. Since this graph is guaranteed to be acyclic, we can perform a topological sort on it to produce an ordered list of strata that dictates the precise sequence for evaluation.

The evaluation phase proceeds sequentially through the topologically sorted strata. For each stratum, a standard seminaive computation is applied to its corresponding rules, iterating until a local fixpoint is achieved. The input for each such computation is the union of the initial database and the complete set of facts derived for all preceding strata. This stratified method prevents redundant work by ensuring that predicates are evaluated only after their dependencies have been fully resolved.

Further optimization of Datalog evaluation is achievable; for an introduction, see Ceri, Gottlob, and Tanca (1990b). A prominent technique relevant to ACGs is Magic Sets rewriting (Beeri and Ramakrishnan 1987). This method transforms the original Datalog program based on a specific query, generating an optimized version that targets relevant data subsets. It introduces auxiliary predicates as filters, ensuring that only pertinent facts are processed and thereby reducing computational overhead, especially in large databases. The transformed program is then evaluated using a bottom-up algorithm such as the seminaive algorithm. This approach emulates top-down evaluation within a bottom-up framework, harnessing the strengths of both strategies.

Part II

ACGs

Chapter 3

Abstract Categorial Grammars

The distinction between syntax and semantics plays a pivotal role in linguistics. Syntax describes sentence structures, while semantics focuses on the meanings that sentences convey. In traditional type-logical grammars, syntax is computed from lexical entries and their associated syntactic categories, built from a set of atomic and fraction types (Lambek 1958). In contrast, semantics is usually represented through simply typed λ-calculus (Montague et al. 1970).

This division into syntax and semantics creates a clear but rigid framework for understanding linguistic structures and meanings. The idea behind Abstract Categorial Grammars (de Groote 2001, ACGs) is to recognize that in a computational model, this distinction does not need to be maintained and we may use a unique mathematical kernel to encode both levels. In ACGs, the asymmetry between syntax and semantics is addressed in two ways. First, by allowing λ-terms on the syntactic side to acknowledge that lexical entries can be considered particular cases of λ-terms. Second, by using the same type theory to model both syntactic categories and semantic types. Resolving this asymmetry provides greater uniformity and flexibility, enabling easier integration across diverse linguistic models and contexts.

This chapter begins by establishing the mathematical kernel of ACGs and introducing their key definitions. We then outline the three fundamental paradigms of ACGs that collectively enable flexible linguistic transformations. To highlight the versatility of ACGs, we show how both strings and trees can be represented uniformly. We then examine the generative capabilities of ACGs, highlighting how different factors affect the range of languages they can represent. Finally, we illustrate ACG's ability to encode various grammatical formalisms, such as CFGs, and explore extensions that enhance its expressivity.

3.1 Definition of the Formalism

3.1.1 Primitives

The type theory used in core ACGs is a subset of the Curry-Howard interpretation of linear logic (Girard 1995), specifically the intuitionistic multiplicative fragment, which corresponds to the simply typed linear λ-calculus.

Definition 3.1 (Linear Implicative Types)**.** Let A be a finite set of atomic types. The set of *linear implicative types* built upon A, $\mathcal{T}(A)$, is inductively defined as follows:

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

The calculus requires constants to represent atomic expressions. These constants are introduced through *higher-order signatures*.

Definition 3.2 (Linear Higher-order Signature)**.** A linear higher-order signature (HOS), Σ, is a triplet $\langle A, C, \tau \rangle$ where:

1. A is a finite set of atomic types;
2. C is a set of term constants;
3. $\tau : C \to \mathcal{T}(A)$ is a function assigning types to constants.

Intuitively, a higher-order signature can be viewed as a vocabulary for a linguistic level, providing the atomic elements and their types needed to construct more complex expressions. Building upon this vocabulary, we introduce the notion of linear λ-terms.

Definition 3.3 (Linear λ-terms)**.** Given a countable set of variables $\mathcal{X}$, The set of linear *λ-terms* defined over $\Sigma = \langle A, C, \tau \rangle$, $T(\Sigma)$ is defined by induction as follows:

1. If $c \in C$, then $c \in T(\Sigma)$;
2. If $x \in \mathcal{X}$, then $x \in T(\Sigma)$;
3. If $x \in \mathcal{X}$, $t \in T(\Sigma)$ and x occurs exactly once free in t, then $\lambda^{\circ} x.\, t \in T(\Sigma)$;
4. If $t, u \in T(\Sigma)$, and their sets of free variables are disjoint ($\mathrm{FV}(t) \cap \mathrm{FV}(u) = \emptyset$), then $(t\, u) \in T(\Sigma)$.

System 2 defines the syntax, typing rules, and evaluation rules for the multiplicative fragment of linear λ-calculus, denoted as $\lambda^{-\circ}$. In this system, the symbol Δ represents linear contexts, where each assumption corresponds to a resource that must be used exactly once.

System 2 (Simply Typed Linear λ-calculus defined over $\Sigma = \langle A, C, \tau \rangle$, $\lambda^{-\circ}$)**.**

Syntax:

$\mathcal{T}(A) :=$	simple linear type:
$\quad A$	*atomic type*
$\quad \mid \mathcal{T}(A) \multimap \mathcal{T}(A)$	*linear implicative type*
$T(\Sigma) :=$	linear λ-term:
$\quad C$	*constant*
$\quad \mid \mathcal{X}$	*linear variable*
$\quad \mid \lambda^{\circ}\mathcal{X}.\, T(\Sigma)$	*linear abstraction*
$\quad \mid (T(\Sigma)\, T(\Sigma))$	*linear application*

Notations:

$$
\begin{aligned}
&a \in A \\
&\alpha, \beta, \gamma \in \mathcal{T}(A) \\
&c \in C \\
&x \in \mathcal{X} \\
&t, u, v \in T(\Sigma)
\end{aligned}
$$

Typing:

$$\frac{}{x : \alpha \vdash_\Sigma x : \alpha}\ \text{LVAR-TERM}$$

$$\frac{}{\vdash_\Sigma c : \tau(c)}\ \text{CONST-TERM} \qquad \frac{\Delta, x : \alpha \vdash_\Sigma t : \beta}{\Delta \vdash_\Sigma \lambda^\circ x\colon \alpha.\, t : \alpha \multimap \beta}\ \text{LABSTR-TERM}$$

$$\frac{\Delta_1 \vdash_\Sigma t : \alpha \multimap \beta \qquad \Delta_2 \vdash_\Sigma u : \alpha \qquad \Delta_1 \cap \Delta_2 = \emptyset}{\Delta_1 \cup \Delta_2 \vdash_\Sigma t\, u : \beta}\ \text{LAPP-TERM}$$

Evaluation:

$$(\lambda^\circ x.\, t)\, u \rhd_{1\beta} t[x := u] \qquad \text{(linear } \beta\text{-contraction)}$$
$$\lambda^\circ x.\, (t\ x) \rhd_{1\eta} t \quad \text{if } x \notin \mathrm{FV}(t) \qquad \text{(linear } \eta\text{-contraction)}$$

3.1.2 Grammar

An ACG is defined over two higher-order signatures. The connection between these signatures is determined by the *lexicon*, which acts as a homomorphic interpreter that maps terms and types defined over one signature to another.

Definition 3.4 (Type Homomorphism)**.** Let A_1 and A_2 be finite sets of atomic types, and let

$$G\colon A_1 \to \mathcal{T}(A_2)$$

be a function sending each atomic type $a \in A_1$ to some type $G(a) \in \mathcal{T}(A_2)$. We extend G to a function

$$\hat{G}\colon \mathcal{T}(A_1) \to \mathcal{T}(A_2)$$

inductively as follows:

1. If $\alpha = a \in A_1$, then $\hat{G}(\alpha) = \hat{G}(a) = G(a)$;
2. If $\alpha = \beta \multimap \gamma$ for some $\beta, \gamma \in \mathcal{T}(A_1)$, then $\hat{G}(\alpha) = \hat{G}(\beta \multimap \gamma) = (\hat{G}(\beta)) \multimap (\hat{G}(\gamma))$.

By construction, $\hat{G}$ is the *unique* type homomorphism extending G, since it preserves the inductive structure of $\mathcal{T}(A_1)$.

Definition 3.5 (λ-term Homomorphism)**.** Let $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ be two higher-order signatures and let $\mathcal{X}$ be a countably infinite set of variables. Suppose we have a function

$$H \colon C_1 \to T(\Sigma_2)$$

sending each constant $c \in C_1$ to some linear λ-term $H(c) \in T(\Sigma_2)$. We extend H to a function

$$\hat{H} \colon T(\Sigma_1) \to T(\Sigma_2)$$

inductively as follows, making $\hat{H}$ the unique homomorphism extending H:

1. If $t = c \in C_1$, then
$$\hat{H}(t) = \hat{H}(c) = H(c);$$
2. If $t = \lambda^\circ x.\, u$ for some $u \in T(\Sigma_1)$ and x occurs exactly once in u, then
$$\hat{H}(t) = \hat{H}(\lambda^\circ x.\, u) = \lambda^\circ x.\, \hat{H}(u);$$
3. If $t = u\, v$ for some $u, v \in T(\Sigma_1)$, then
$$\hat{H}(t) = \hat{H}(u\, v) = \hat{H}(u)\, \hat{H}(v).$$

Definition 3.6 (Lexicon)**.** Given two higher-order signatures $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$, a lexicon $\mathcal{L} : \Sigma_1 \to \Sigma_2$ is a pair $\langle G, H \rangle$ such that:

1. $G : A_1 \to \mathcal{T}(A_2)$ is a function that interprets the atomic types of Σ_1 as linear implicative types built upon A_2;
2. $H : C_1 \to T(\Sigma_2)$ is a function that interprets the constants of Σ_1 as linear λ-terms built upon Σ_2;
3. For any constant c in C_1, the following typing judgment is derivable:
$$\vdash_{\Sigma_2} H(c) : \hat{G}(\tau_1(c)).$$

In the following chapters, whenever we use $\mathcal{L}$ to denote a lexicon, it should be understood that $\mathcal{L}$ also implicitly includes the homomorphisms $\hat{G}$ and $\hat{H}$ associated with this lexicon. The homomorphism induced by $\mathcal{L} : \Sigma_1 \to \Sigma_2$ commutes with the typing relation. Thus, if

$$x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash_{\Sigma_1} t : \alpha$$

The following judgment also holds:

$$x_1 : \mathcal{L}(\alpha_1), \ldots, x_n : \mathcal{L}(\alpha_n) \vdash_{\Sigma_2} \mathcal{L}(t) : \mathcal{L}(\alpha).$$

Definition 3.7 (ACG)**.** An Abstract Categorial Grammar is a quadruple $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ where:

1. Σ_1 is the *abstract* signature;
2. Σ_2 is the *object* signature;
3. $\mathcal{L}$ is a lexicon from Σ_1 to Σ_2;
4. $\mathbf{S}$ is called the *distinguished type* and belongs to $\mathcal{T}(A_1)$.

The set of terms defined over the abstract signature, $T(\Sigma_1)$, are the *abstract λ-terms*, and the set of terms defined over the object signature, $T(\Sigma_2)$, are the *object λ-terms*.

An ACG, $\mathcal{G}$, generates two languages of λ-terms, the *abstract language* and the *object language*. The former is composed of closed linear λ-terms defined over Σ_1 of type $\mathbf{S}$ and represents the abstract grammatical structures. The latter is defined as the image of the abstract language through the term homomorphism induced by the lexicon $\mathcal{L}$ and can be thought of as the concrete forms derived from the abstract structures.

Definition 3.8 (ACG Languages)**.** Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be an ACG. The abstract language $\mathcal{A}(\mathcal{G})$ and the object language $\mathcal{O}(\mathcal{G})$ of $\mathcal{G}$ are defined as follows:

$$\mathcal{A}(\mathcal{G}) = \{t \in T(\Sigma_1) \mid \vdash_{\Sigma_1} t : \mathbf{S}\}$$

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

Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be an ACG, we call the set of terms defined over the abstract signature, $T(\Sigma_1)$, *abstract λ-terms* and the set of terms defined over the object signature, $T(\Sigma_2)$, *object λ-terms*.

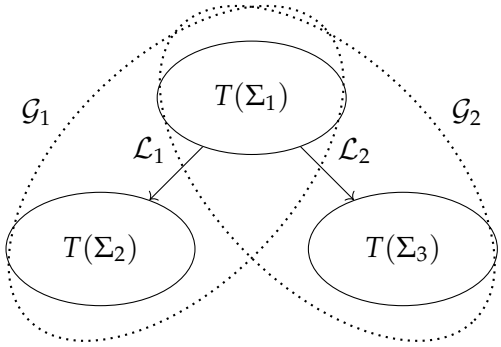
3.1.3 Paradigms

ACGs offer two paradigms for manipulating the abstract and object languages. The first paradigm, the *applicative paradigm*, addresses the following question: what is the image

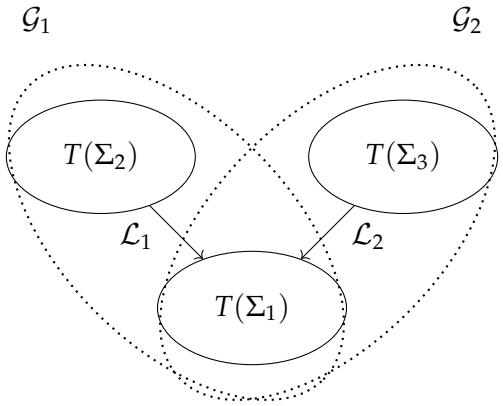
of an abstract term in the object language? This paradigm is computationally straightforward, relying on substitution-based operations to replace each constant with its interpretation, followed by β-reduction. The second paradigm, the *deductive paradigm*, is the inverse of the applicative paradigm. This involves an inverse problem called *ACG parsing* that amounts to a proof search and can be formulated as follows: Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be an ACG and $t_2 \in T(\Sigma_2)$. Does there exist a $t_1 \in \mathcal{A}(\mathcal{G})$ such that $\mathcal{L}(t_1) \rhd_\beta t_2$?

One of the key properties of ACGs is their ability to be composed when they share a signature (see Figure 3.1). In particular, two ACG grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ can be composed under three main conditions: they may share the same abstract signature (Figure 3.1a), share the same object signature (Figure 3.1b), or the abstract signature of $\mathcal{G}_2$ coincides with the object signature of $\mathcal{G}_1$ (Figure 3.1c).

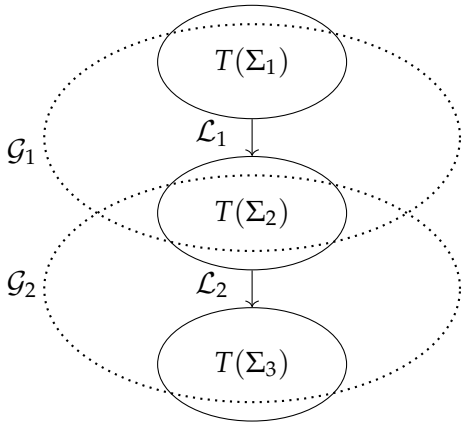
Figures 3.1a and 3.1b illustrate how these composition strategies yield a third paradigm, the *transductive paradigm*, by chaining the deductive and applicative paradigms in either order. Using the pivot signature Σ_1 as an intermediary, one can transform a λ-term from $T(\Sigma_2)$ into a λ-term in $T(\Sigma_3)$, and vice versa. This flexibility supports the construction of complex architectures that capture subtle linguistic phenomena, particularly at the syntax-semantics interface (Pogodalla 2017a).



(A) Composition of $\mathcal{G}_1$ and $\mathcal{G}_2$ with Σ_1 as the shared abstract signature.



(B) Composition of $\mathcal{G}_1$ and $\mathcal{G}_2$ with Σ_1 as the shared object signature.



(C) Composition of $\mathcal{G}_1$ and $\mathcal{G}_2$ using the object signature Σ_2 of $\mathcal{G}_1$ as the abstract signature of $\mathcal{G}_2$.

FIGURE 3.1: Composition of ACGs

3.2 Uniform languages

One reason for using ACGs is their ability to create uniform linguistic representations using a minimal set of primitives. This section illustrates how to encode two common

linguistic structures using linear λ-terms: strings and trees.

3.2.1 Encoding Strings

Strings are linear sequences of symbols that appear ubiquitously in natural language processing. The key insight for encoding strings in ACGs is to represent each symbol not as a static element, but as a function that takes a continuation and prepends itself to it.

Definition 3.9 (String Modelling)**.** Let V be an alphabet. The signature of strings Σ_V defined over V is a triplet $\langle A, C, \tau \rangle$ where:

- the set of atomic types A is a singleton containing the atomic type o;
- the set of constants C is equal to the alphabet V;
- for each $v \in V$, $\tau(v) = o \multimap o$.

In Σ_V, the empty string, denoted as ϵ, is defined as the identity function $\lambda^\circ x.\, x$. The concatenation operator, represented by $+$, is defined through function composition, corresponding to the combinator **B**: $\lambda^\circ a.\, \lambda^\circ b.\, \lambda^\circ z.\, a\,(b\,z)$. For convenience, we use $+$ as a infix operator. Additionally, we use String as a shorthand for the type $o \multimap o$.

Example 3.1 (String as Term). Let $V = \{\mathit{The}, \mathit{revenues}, \mathit{increase}\}$ be an alphabet. The string *The revenues increase* can be expressed as the following linear λ-term:

$$\lambda^\circ z.\, \mathit{The}\,(\mathit{revenues}\,(\mathit{increase}\,z)\,).$$

To show that the presented encoding of strings, together with the empty string ϵ and the concatenation operator $+$, forms a monoid, we must establish that:

1. ϵ is the neutral element for $+$;
2. $+$ is associative.

Lemma 3.1 (Neutral element of $+$)**.** $s + \epsilon = \epsilon + s = s$.

Proof.
Claim. $s + \epsilon = s$

1.	$\lambda^\circ z.\, s\,(\epsilon\, z)$	$\rhd_\beta$
2.	$\lambda^\circ z.\, s\,((\lambda^\circ x.\, x)\, z)$	By definition of ϵ
3.	$\lambda^\circ z.\, s\, z$	$\rhd_\beta$
4.	s	$\rhd_{1\eta}$

Claim. $\epsilon + s = s$

1.	$\lambda^\circ z.\, \epsilon\,(s\, z)$	$\rhd_\beta$
2.	$\lambda^\circ z.\, (\lambda^\circ x.\, x)\,(s\, z)$	By definition of ϵ
3.	$\lambda^\circ z.\, s\, z$	$\rhd_\beta$
4.	s	$\rhd_{1\eta}$

□

Lemma 3.2 (Associativity of $+$)**.** $(c + d) + e = c + (d + e)$.

Proof. For $(c + d) + e$, we have the following computations:

1.	$\lambda^\circ z.\, (c + d)\,(e\, z)$	$\rhd_\beta$
2.	$\lambda^\circ z.\, (\lambda^\circ x.\, c\,(d\, x))\,(e\, z)$	By lemma $\rhd_\beta$ and α-renaming
3.	$\lambda^\circ z.\, c\,(d\,\,(e\, z))$	$\rhd_\beta$

For $c + (d + e)$, we have:

1.	$\lambda^\circ z.\, c\,((d\, +\, e)\, z)$	$\rhd_\beta$
2.	$\lambda^\circ z.\, c\,((\lambda^\circ x.\, d\,(e\, x))\, z)$	By $\rhd_\beta$ and α-renaming
3.	$\lambda^\circ z.\, c\,(d\,\,(e\, z))$	$\rhd_\beta$

□

3.2.2 Encoding Trees

The encoding is not limited to linear sequences like strings but extends to hierarchical structures such as trees.

Definition 3.10 (Ranked Alphabet)**.** A ranked alphabet Σ_V is a pair $\langle V, R\rangle$ where:

- V is an alphabet;
- $R : V \to N$ is a function that assigns an arity (or a rank) to each symbol in V.

Definition 3.11 (Tree Language)**.** Let $\mathcal{X}$ be a countable set of variables. The set of trees defined over $\mathcal{X}$ and a ranked alphabet $\Sigma_V = \langle V, R \rangle$, $\mathfrak{T}(\Sigma_V, \mathcal{X})$, is inductively defined as follows:

1. If $x \in \mathcal{X}$, then $x \in \mathfrak{T}(\Sigma_V, \mathcal{X})$;
2. If $v \in V$ and $R(v) = 0$, then $v \in \mathfrak{T}(\Sigma_V, \mathcal{X})$;
3. If $v \in V$ and $R(v) = k$, then $v(t_1, \ldots, t_k) \in \mathfrak{T}(\Sigma_V, \mathcal{X})$ if $t_1, \ldots, t_k \in \mathfrak{T}(\Sigma_V, \mathcal{X})$.

To exemplify these definitions, consider the following parse tree in Figure 3.2, which represents an element of a tree language. The ranked alphabet $\Sigma_V = \langle V, R \rangle$ is defined implicitly, with V including symbols such as S, NP, VP, N, *Stakeholders*, *sell*, and *stocks*, and R assigning ranks based on the number of child nodes each symbol can have.

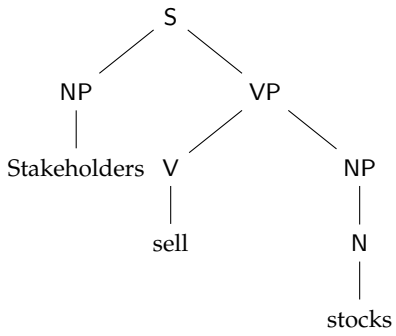


FIGURE 3.2: Parse tree representing the sentence: *Stakeholders sell stocks*.

Definition 3.12 (Tree Encoding)**.** Let $\Sigma_V = \langle V, R \rangle$ be a ranked alphabet. The signature of trees Σ_{Tree} defined over Σ_V is a triplet $\langle A, C, \tau \rangle$ where:

- the set of atomic types A is a singleton containing the atomic type Tree;
- the set $C = \{v_k \mid v \in V, R(v) = k\}$ is such that for each $v \in V$ with $R(v) = k$, there exists $v_k \in C$;
- for each $v_k \in C$, $\tau(v_k) = \underbrace{\mathsf{Tree} \multimap \cdots \multimap \mathsf{Tree}}_{k \text{ times}} \multimap \mathsf{Tree}$.

When the arity of a symbol is 0, we write v instead of v_0 by abuse of notation.

Example 3.2 (Encoded Tree with Signature). The tree introduced in Figure 3.2 can be encoded as:

$$\mathsf{S}_2\ (\mathsf{NP}_1\ \textit{Stakeholders})\ (\mathsf{VP}_2\ (\mathsf{V}_1\ \textit{sell})\ (\mathsf{NP}_1\ (\mathsf{N}_1\ \textit{stocks}))).$$

This encoding uses the signature:

$$\Sigma_{\mathsf{Tree}} = \langle \{\mathsf{Tree}\}, \{\mathsf{S}_2, \mathsf{NP}_1, \mathsf{VP}_2, \mathsf{V}_1, \mathsf{N}_1, \mathsf{Stakeholders}, \mathsf{sell}, \mathsf{stocks}\}, \tau \rangle,$$

where the type assigning function τ is defined as:

$$\begin{aligned}
\tau(\mathsf{S}_2) &= \mathsf{Tree} \multimap \mathsf{Tree} \multimap \mathsf{Tree},\\
\tau(\mathsf{NP}_1) &= \mathsf{Tree} \multimap \mathsf{Tree},\\
\tau(\mathsf{VP}_2) &= \mathsf{Tree} \multimap \mathsf{Tree} \multimap \mathsf{Tree},\\
\tau(\mathsf{V}_1) &= \mathsf{Tree} \multimap \mathsf{Tree},\\
\tau(\mathsf{N}_1) &= \mathsf{Tree} \multimap \mathsf{Tree},\\
\tau(\textit{Stakeholders}) &= \mathsf{Tree},\\
\tau(\textit{sell}) &= \mathsf{Tree},\\
\tau(\textit{stocks}) &= \mathsf{Tree}.
\end{aligned}$$

To perform surface realization from trees into strings, we may construct an ACG denoted by $\mathcal{G}_{\text{SURFACE}} = \langle \Sigma_{\mathsf{Tree}}, \Sigma_{\mathsf{String}}, \mathcal{L}, \mathsf{Tree} \rangle$. The tree signature $\Sigma_{\mathsf{Tree}} = \langle A_{\mathsf{Tree}}, C_{\mathsf{Tree}}, \tau_{\mathsf{Tree}} \rangle$ is a higher-order signature constructed using the tree encoding over the ranked alphabet $\Sigma_V = \langle V, R \rangle$, representing the syntactic structures of trees. The string signature Σ_{String} models strings and is constructed by using the leaves as the alphabet, representing the linearized strings.

We define a lexicon $\mathcal{L}$ by mapping the tree type Tree to the string type String, that is, $\mathcal{L}(\mathsf{Tree}) = \mathsf{String}$. For each constant $c \in C_{\mathsf{Tree}}$ with type $\tau_{\mathsf{Tree}}(c) = \mathsf{Tree}$, we set $\mathcal{L}(c) = c$. If a constant $c \in C_{\mathsf{Tree}}$ has a type of the form

$$\underbrace{\mathsf{Tree} \multimap \cdots \multimap \mathsf{Tree}}_{k \text{ times}} \multimap \mathsf{Tree},$$

we define

$$\mathcal{L}(c) = \lambda^{\circ} x_0\ \ldots\ x_k.\ x_0 + \cdots + x_k.$$

3.2.3 ACG Hierarchy

With the encoding methods for string and tree languages established, we can now delve into the ACG hierarchy (de Groote and Pogodalla 2004), which provides insights into the generative power and complexity of different classes of ACGs. This hierarchy is based on the concept of *type order*.

Definition 3.13 (Type Order)**.** Given a type α, its order, $ord(\alpha)$, is defined by induction on α as follows:

1. If $\alpha = a$ and $a \in A$, then $ord(\alpha) = 1$;
2. If $\alpha = \beta \multimap \gamma$, then $ord(\alpha) = \max(1 + ord(\beta), ord(\gamma))$.

Definition 3.14 (ACG Hierarchy)**.** The *order* of an ACG is defined as the highest order of its abstract constants' types, while its *complexity* is determined by the highest order of the interpretations assigned to its atomic abstract types. We denote by $\mathbf{G}(n, m)$ the class of ACGs whose order is no more than n and whose complexity is no more than m.

Table 3.1 shows the generative power of an ACG given its order and its complexity and the nature of the object language it can generate. The second order case is a class of interest since it has been shown that parsing is polynomial (Salvati 2005). Moreover, second order ACGs allow for the generation of mildly context-sensitive languages. The next chapter will describe an algorithm to parse second order ACGs. For orders greater than 2, parsing becomes equivalent to the decidability of the multiplicative exponential fragment of linear logic (MELL, Girard 1987).

$\mathbf{G}$(order, complexity)	String language	Tree language
$\mathbf{G}(1, n)$	finite	finite
$\mathbf{G}(2, 1)$	regular	regular
$\mathbf{G}(2, 2)$	context-free	context-free
$\mathbf{G}(2, 3)$	well-nested multiple context-free	$\subset$ 1-visit attribute grammar
$\mathbf{G}(2, 4)$	mildly context-sensitive	tree-generating hyperedge replacement grammar
$\mathbf{G}(2, 4 + n)$	$\mathbf{G}(2, 4)$	$\mathbf{G}(2, 4)$
$\mathbf{G}(3, n)$	MELL decidability	MELL decidability

TABLE 3.1: Generative power of ACGs.

3.3 Encoding Context-Free Grammars

3.3.1 Context-Free Grammars

ACGs are not designed to serve as another grammatical formalism. Instead, they provide a foundation for building a grammatical framework (inspired by Ranta 2004) that can encode various grammatical models. This approach supports uniform parsing and generation algorithms across multiple grammar formalisms. This section outlines the encoding of CFGs.

Definition 3.15 (CFG)**.** A CFG is a tuple $G = \langle V, N, R, S \rangle$ where

1. V is a finite set of terminal symbols;
2. N is a finite set of non-terminal symbols, disjoint from V;
3. R is a set of production rules of the form $a \to \alpha$ with $a \in N$ and $\alpha \in (V \cup N)^\star$;
4. S is the start symbol and belongs to N.

As an example, the CFG $G^{a^n b^n}$ in Example 3.3 generates the language $\{a^n b^n \mid n \geq 0\}$, where each string has an equal number of a's followed by b's.

Example 3.3 (CFG for generating $a^n b^n$). Consider the CFG $G^{a^n b^n} = \langle V, N, R, S \rangle$, where:

$$V = \{a, b\}, \quad N = \{S\}, \quad \text{and } R \text{ is defined as follows:}$$

$$S \to aSb \tag{1}$$
$$S \to \epsilon \tag{2}$$

3.3.2 ACG Encoding

Definition 3.16 (Rule Skeleton)**.** Let $G = \langle V, N, R, S \rangle$ be a CFG and let $a \to \alpha$ be a production rule in R. The skeleton of this rule is a pair $\langle a, \lceil \alpha \rceil \rangle$ where $\lceil \alpha \rceil$ is a string of non-terminal symbols defined by induction as follows:

1. $\lceil \epsilon \rceil = \epsilon$;
2. if $v \in V$ and $\beta \in (V \cup N)^\star$, then $\lceil v\beta \rceil = \lceil \beta \rceil$;

3. if $n \in N$ and $\beta \in (V \cup N)^\star$, then $\lceil n\beta \rceil = n\lceil \beta \rceil$.

Definition 3.17 (Derivation Tree Level)**.** Let $G = \langle V, N, R, S \rangle$ be a CFG. The higher-order signature used to construct the derivation trees of G, $\Sigma_G = \langle A, C, \tau \rangle$, is defined as follows:

1. The set of atomic types A is equal to the set of non-terminal symbols N;
2. The set of constants C is exactly the set of production rules R;
3. For every production rule $r \in R$ with a skeleton in the form $\langle a, a_1 \dots a_n \rangle$, the type of the constant associated to r, $\tau(r)$ is defined as $a_1 \multimap \cdots \multimap a_n \multimap a$.

The application of Definition 3.17 to Example 3.3 yields Example 3.4. In this representation of a higher-order signature, the definition of an atomic type $a \in A$ is denoted by $a :: \mathsf{Type}$. For every constant c and for α, a linear implicative type, we define $\tau(c)$ in extension using the notation $c : \alpha$.

Example 3.4 ($\Sigma_{G^{a^n b^n}}$).

$$S :: \mathsf{Type}$$
$$r_1 : \mathsf{S} \multimap \mathsf{S}$$
$$r_2 : \mathsf{S}$$

Definition 3.18 (Rules as linear λ-terms)**.** Let $r : a \to \alpha$ be a production rule of a CFG. Let $[\![\alpha]\!]$ be a λ-term defined inductively on α as follows:

1. If $\alpha = \epsilon$, then $[\![\alpha]\!] = \lambda x.x$;
2. If $\alpha = n\beta$ where n is a non-terminal symbol, then $[\![\alpha]\!] = x_i + [\![\beta]\!]$ where x_i is a fresh variable;
3. If $\alpha = v\beta$ where v is a terminal symbol, then $[\![\alpha]\!] = v + [\![\beta]\!]$.

The λ-term associated with r, denoted by $[\![r]\!]$, is defined as:

$$[\![r]\!] = \lambda x_1, \dots, x_n.[\![\alpha]\!]$$

where $x_1, \dots, x_n$ is the sequence of free fresh variables occurring in $[\![\alpha]\!]$.

Definition 3.19 (CFG Encoding)**.** Let $G = \langle V, N, R, S \rangle$ be a CFG. The ACG corresponding to G, $\mathcal{G}_G = \langle \Sigma_G, \Sigma_V, \mathcal{L}_G, S \rangle$ is defined as follows:

1. Σ_G is formed according to Definition 3.17;
2. Σ_V is constructed according to Definition 3.9;
3. $\mathcal{L}_G : \Sigma_G \to \Sigma_V$ is defined such that:
   (a) $\mathcal{L}_G(a) = \mathsf{String}$;
   (b) $\mathcal{L}_G(r) = [\![r]\!]$

The string signature $\Sigma_{\{a,b\}}$ for the alphabet $V = \{a, b\}$ is defined in Example 3.5. Within this signature, String is introduced as a type alias for $o \multimap o$, representing strings using the equal symbol. Similarly, string concatenation, $+$, is defined as a λ-term alias. Finally, Example 3.6 presents the lexicon $\mathcal{L}_{G^{a^n b^n}}$, completing the encoding of the CFG $G^{a^n b^n}$ into an ACG:

$$\mathcal{G}_{G^{a^n b^n}} = \langle \Sigma_{G^{a^n b^n}}, \Sigma_{\{a,b\}}, \mathcal{L}_{G^{a^n b^n}}, \mathsf{S} \rangle.$$

Example 3.5 ($\Sigma_{\{a,b\}}$).

$$\begin{aligned} o &:: \mathsf{Type} \\ \mathsf{String} &= o \multimap o \\ a &: \mathsf{String} \\ b &: \mathsf{String} \\ + &= \lambda^\circ x.\, \lambda^\circ y.\, \lambda^\circ z.\, x\,(y\,z) \\ \epsilon &= \lambda^\circ x.\, x \end{aligned}$$

Example 3.6 ($\mathcal{L}_{G^{a^n b^n}} : \Sigma_{G^{a^n b^n}} \to \Sigma_T$).

$$\begin{aligned} \mathsf{S} &\mapsto \mathsf{String} \\ r_1 &\mapsto \lambda^\circ x.\, a + x + b \\ r_2 &\mapsto \epsilon \end{aligned}$$

Theorem 3.1. *For every CFG G and its corresponding ACG $\mathcal{G}_G$, the following propositions hold:*

1. *$\mathcal{A}(\mathcal{G}_G)$ is isomorphic to the set of parse trees of G;*
2. *$\mathcal{O}(\mathcal{G}_G)$ is isomorphic to the language generated by G (via the string encoding of Definition 3.9).*

Proof. See Propositions 1 and 2 of de Groote and Pogodalla (2004). □

This construction can be adapted to encode linear Context-Free Tree Grammars and Linear Context-Free Rewriting Systems (de Groote and Pogodalla 2004). The key difference lies in how the rules are interpreted.

3.4 Extensions

3.4.1 Enriching the Kernel

The simply typed linear λ-calculus provides a solid theoretical foundation for ACGs, but modeling scenarios often require additional expressive capabilities. Following de Groote (2001), ACGs can be extended by enriching their type system with new logical connectives, which become type constructors via the Curry-Howard isomorphism. These extensions must satisfy two key requirements:

Strong normalization to ensure β-reduction terminates in the applicative paradigm;

Decidable parsing to maintain practical utility as a grammatical framework

While connectives from linear logic are natural candidates for such extensions, other formal systems could also serve as sources for new type constructors. However, while strong normalization is often straightforward to verify, establishing parsing decidability for extended ACGs remains a major open challenge.

3.4.2 Relaxing Linearity Constraints

Another avenue for increasing the expressive power of ACGs involves reconsidering one of their fundamental properties: linearity. The linear constraint can sometimes be limiting, especially in semantics.

Relaxing linearity has led to the development of affine (Yoshinaka 2006) and non-linear (Salvati 2007) ACGs. *Affine ACGs* allow terms that do not necessarily use all variables introduced by an abstraction, making them less constrained than linear ACGs. Affine ACGs can be transformed into equivalent linear ACGs without losing generative power, though this process results in exponential growth in grammar size. *Non-linear ACGs*, on the other hand, allow the reuse of variables within expressions, which boost their expressivity, especially for semantic purposes. Importantly, despite the removal of the linearity constraint, parsing for non-linear ACGs remains decidable.

Chapter 4

Parsing

Kanazawa (2007) introduced a transformation that reduces the parsing problem of second-order ACGs to Datalog program evaluation. This reduction yields a declarative parsing algorithm expressed in Datalog, achieving polynomial-time complexity (for a detailed proof of this complexity result, refer to Kanazawa 2017).

Since this reduction applies to all second-order ACGs, it is relevant to context-free derivation systems, as these grammars encode mildly context-sensitive languages. Additionally, this transformation may leverage the extensive optimizations available for Datalog programs, enhancing overall parsing efficiency.

The chapter begins by establishing the algorithmic foundations necessary for understanding this reduction. The core of the chapter details Kanazawa's reduction, explaining how to construct a Datalog program from a given second-order ACG and how to generate instances and queries corresponding to terms to parse. The chapter concludes by exploring briefly extensions to Kanazawa's reduction.

4.1 Algorithmic Foundations

4.1.1 Typed Terms

Recall the ACG parsing problem: Given an ACG $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ and a term $t_2 \in T(\Sigma_2)$, determine if there exists a term $t_1 \in \mathcal{A}(\mathcal{G})$ such that $\mathcal{L}(t_1) \rhd_\beta t_2$. The overall idea behind the reduction is to use typing properties to eliminate the need to search for β-expansion strategies to find the term $\mathcal{L}(t_1)$.

Definition 4.1 (Type Polarity)**.** Let $\Gamma \vdash t : \alpha$ be a type assignment formula in simply typed λ-calculus. The polarity of a type is defined inductively as follows:

1. α has a positive occurrence in $\Gamma \vdash t : \alpha$.
2. Every type β in a type assignment in Γ has a negative occurrence in $\Gamma \vdash t : \alpha$.
3. For a positive type $\beta \to \gamma$ in $\Gamma \vdash t : \alpha$, β has a negative occurrence and γ has a positive occurrence.
4. For a negative type $\beta \to \gamma$ in $\Gamma \vdash t : \alpha$, β has a positive occurrence and γ has a negative occurrence.

Definition 4.2 (Balanced Type Assignment)**.** A type assignment formula $\Gamma \vdash t : \alpha$ in simply typed λ-calculus is *balanced* if:

1. Every atomic type has at most one positive occurrence in the type assignment formula;
2. Every atomic type has at most one negative occurrence in the type assignment formula.

Theorem 4.1 (Coherence (Babaev and Solov'ev 1982; Mints 1979))**.** *All inhabitants of a balanced type assignment formula are equal under $\beta\eta$-equivalence.*

Theorem 4.2 (Type assignment formula of linear λ-terms (Hirokawa 1991))**.** *The type assignment formula of the principal type of a pure linear λ-term is balanced.*

The goal is therefore to simplify membership testing by reducing it to type inference and inhabitation checking. To achieve this, both subject reduction and subject expansion are required, which hold in the linear setting because β-reduction is non-erasing and non-duplicating.

Since Theorem 4.2 applies only to pure linear λ-terms, we describe a method for transforming any linear λ-term into its pure equivalent. This transformation relies on extracting the term's constants and free variables into ordered sequences. In the following definitions, we use $\frown$ to denote tuple concatenation.

Definition 4.3 (Constant Sequence)**.** Let t be a linear lambda-term. The sequence of constants occurring in t, denoted $\overrightarrow{\text{cons}}(t)$, is defined inductively as follows:

1. If $t = c$ for a constant c, then $\overrightarrow{\text{cons}}(t) = (c)$;
2. If $t = x$ for a variable x, then $\overrightarrow{\text{cons}}(t) = ()$ (the empty sequence);

3. If $t = \lambda^\circ x.\, t'$ for an abstraction, then $\overrightarrow{\text{cons}}(t) = \overrightarrow{\text{cons}}(t')$;
4. If $t = t_1\, t_2$ for an application, then $\overrightarrow{\text{cons}}(t) = \overrightarrow{\text{cons}}(t_1) \frown \overrightarrow{\text{cons}}(t_2)$.

Definition 4.4 (Free Variable Sequence)**.** The *free variable sequence* of a term t, denoted $\overrightarrow{\text{FV}}(t)$, is defined by the following rules:

- If $t = c$ for a constant c, then $\overrightarrow{\text{FV}}(t) = ()$.
- If $t = x$ for a variable x, then $\overrightarrow{\text{FV}}(t) = (x)$.
- If $t = u\, v$ for an application, then $\overrightarrow{\text{FV}}(t) = \overrightarrow{\text{FV}}(u) \frown \overrightarrow{\text{FV}}(v)$.
- If $t = \lambda^\circ x.\, t_1$ for an abstraction, then $\overrightarrow{\text{FV}}(t)$ is the sequence $\overrightarrow{\text{FV}}(t_1)$ with all occurrences of the variable x removed.

Definition 4.5 (Pure Term Transformation)**.** Let t be a closed term with a sequence of constants $\overrightarrow{\text{cons}}(t) = (c_1, \ldots, c_m)$. We can construct a corresponding pure term $\hat{t}[x_1, \ldots, x_m]$ by substituting each constant c_i with a variable x_i, such that:

1. The free variable sequence of $\hat{t}$ is $\overrightarrow{\text{FV}}(\hat{t}) = (x_1, \ldots, x_m)$;
2. Substituting back the variables restores the original term: $\hat{t}[x_1 := c_1, \ldots, x_m := c_m] = t$.

When working with λ-terms, we will often require them to be in *η-long form*, meaning every subterm of implicative type is fully η-expanded. A term is in *η-long β-normal form* when it is both in β-normal form and η-expanded according to its type. By enforcing this uniform structure, we can factor out the η aspect of conversion and define term equality solely through the lens of β-equivalence.

4.1.2 Types

Definition 4.6 (Atomic Type Sequence)**.** Let A be a finite set of atomic types. Given a linear implicative type $\alpha \in \mathcal{T}(A)$, *the sequence of atomic types* in α, denoted by $\overline{\alpha}$, is inductively defined as follows:

1. If $\alpha = a$, then $\overline{\alpha} = (a)$;
2. If $\alpha = \beta \multimap \gamma$ then $\overline{\alpha} = \overline{\beta} \frown \overline{\gamma}$.

Definition 4.7 (Type Relabeling)**.** A *type relabeling*, denoted by σ, is a type substitution that maps atomic types to atomic types only.

Definition 4.8 (Relabeling-Equivalence)**.** Let R be the relation on types defined by

$$\alpha R \beta \quad \Longleftrightarrow \quad \beta = \alpha\sigma \text{ for some type relabeling } \sigma.$$

We define $\sim$ to be the smallest symmetric and transitive relation containing R.

Concretely, two types α and β are *relabeling-equivalent* (written $\alpha \sim \beta$) if there is a finite sequence of types starting at α and ending at β such that any two consecutive types in the sequence are related by R or its inverse.

Definition 4.9 (Type Skeleton)**.** The *type skeleton* of a type α, denoted $[\alpha]$, is the equivalence class of α under the relabeling relation $\sim$. Specifically, $[\alpha]$ contains all types obtained from α by relabeling its atomic types.

Definition 4.10 (Skeleton Labeling Function)**.** Let α be a type, and let its sequence of atomic types, $\overline{\alpha}$, have length k. For any sequence of k atomic types, $b_1, \ldots, b_k$, we define $[\alpha](b_1, \ldots, b_k)$ to be the unique type β that satisfies two conditions:

1. β has the same structure as α (i.e., $\beta \in [\alpha]$).
2. The sequence of atomic types in β is $\overline{\beta} = (b_1, \ldots, b_k)$.

By construction, for every type β, it holds that

$$[\beta](\overline{\beta}) = \beta.$$

4.2 Kanazawa's Reduction

Warren's work (F. C. N. Pereira and D. H. D. Warren 1983) demonstrated how parsing could be reframed as a deductive database problem within logic programming. Building on this foundation, Ullman made advances (Ullman 1988) by implementing CFG recognition using polynomial-time Datalog programs. Kanazawa further expanded this theoretical framework in 2007, extending Datalog parsers to handle second-order ACGs while reconceptualizing grammatical recognition as a type checking problem rather than general logical deduction. This reconceptualization is enabled by key theoretical properties: balanced type assignments have unique inhabitants up to $\beta\eta$-equivalence (by Theorem 4.1), while pure linear λ-terms have balanced principal types (by Theorem 4.2). Together with subject reduction and expansion properties of linear λ-calculus, this allows

us to reduce ACG parsing to type inference and inhabitation checking. Instead of searching for β-expansions of the interpreted term directly, we can construct a Datalog program that effectively computes type assignments and verifies their inhabitation. The following section details this reduction, showing how to encode the typing constraints as Datalog rules and facts.

Having established the theoretical foundations that enable this reduction, we now turn to the algorithmic details. We now present the algorithm for transforming second-order ACG parsing into Datalog program evaluation, using the ACG encoding of the CFG grammar from Example 3.3 (which generates the language $a^n b^n$) to illustrate each step of the reduction.

4.2.1 Program Construction

First, we present the construction of a Datalog program $P_{\mathcal{G}}$ derived from a second-order ACG, $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$. We begin by creating the schemas for $P_{\mathcal{G}}$. Instead of using a for atomic types, we use p to emphasize that they become predicate symbols in Datalog.

Definition 4.11 (Schema of $P_{\mathcal{G}}$)**.** The schema of $P_{\mathcal{G}}$, $\mathsf{SCH}(P_{\mathcal{G}})$, is defined exactly as follows:

1. For each constant in the object signature $c \in C_2$, define an extensional predicate $c^k \in \mathsf{EDB}(P_{\mathcal{G}})$, where $k = |\overline{\tau_2(c)}|$;
2. For each abstract atomic type $p \in A_1$, define an intensional predicate $p^k \in \mathsf{IDB}(P_{\mathcal{G}})$, where $k = |\overline{\mathcal{L}(p)}|$.

The second point implies that the maximum arity of the intensional predicates is determined by the complexity of the ACG. As an example, if an ACG has a complexity of 1, the resulting Datalog program will be *monadic*, meaning it will consist of unary intensional predicates.

Applying Definition 4.11 to $\mathcal{G}_{a^n b^n}$ yields the schema in Example 4.1. In $\mathcal{G}_{a^n b^n}$, $\mathcal{L}(\mathsf{S})$ maps to the type $\mathsf{String} = o \multimap o$. Since String contains two atomic types (one o in the domain and one in the codomain), the predicate S has arity 2. Similarly, the constants a and b are both typed by String, so their corresponding predicates also have arity 2.

Example 4.1 (Schema of $P_{\mathcal{G}_{a^n b^n}}$).

$$\mathsf{EDB}(P_{\mathcal{G}_{a^n b^n}}) = \{a^2, b^2\}$$
$$\mathsf{IDB}(P_{\mathcal{G}_{a^n b^n}}) = \{\mathsf{S}^2\}$$

The second step in this construction is to derive the Datalog rules by applying the following definition to each constant in the abstract higher-order signature, Σ_1.

Definition 4.12 (Rule Creation)**.** Let $r \in C_1$ with $\tau_1(r) = p_1 \multimap \cdots \multimap p_n \multimap p_0$ where $n \geq 0$. Let $\mathcal{L}(r)$ be a term in η-long β-normal form relative to $\mathcal{L}(\tau_1(r))$ with $\overrightarrow{\text{cons}}(\mathcal{L}(r)) = (c_1, \ldots, c_m)$.

1. First, obtain a pure term $\widehat{\mathcal{L}(r)}[y_1, \ldots, y_m]$ from $\mathcal{L}(r)$;
2. Then, compute the principal type of $\widehat{\mathcal{L}(r)}[y_1, \ldots, y_m]$:

$$y_1 : \beta_1, \ldots, y_m : \beta_m \vdash_{PT} \widehat{\mathcal{L}(r)}[y_1, \ldots, y_m] : \alpha_1 \multimap \cdots \multimap \alpha_n \multimap \alpha_0;$$

3. Finally, construct the Datalog rule r as follows:

$$[r] \quad p_0(\overline{\alpha_0}) \leftarrow p_1(\overline{\alpha_1}), \ldots, p_n(\overline{\alpha_n}), c_1(\overline{\beta_1}), \ldots, c_m(\overline{\beta_m}).$$

   In r, for each $i \in \{0, \ldots, n\}$, every element of $\overline{\alpha_i}$ is treated as a Datalog variable; similarly, for each $j \in \{1, \ldots, m\}$, every element of $\overline{\beta_j}$ is also treated as a Datalog variable.

The resulting Datalog rules from this transformation have two key properties: they are free of any Datalog constants, and they are guaranteed to conform to the schema from Definition 4.11. The latter property is established by the following lemma.

Lemma 4.1 (Skeleton Equalities)**.** *Suppose that:*

- $t \in T(\Sigma_2)$ *is* η*-long relative to* $\mathcal{L}(p_1 \multimap \cdots \multimap p_n \multimap p_0)$*;*
- $\overrightarrow{\text{cons}}(t) = (c_1, \ldots, c_m)$ *is the sequence of constants used in* t*;*
- $y_1 : \beta_1, \ldots, y_m : \beta_m \vdash_{PT} \hat{t}[y_1, \ldots, y_m] : \alpha_1 \multimap \cdots \multimap \alpha_n \multimap \alpha_0$.

Then, the following equalities for the skeletons of the types hold:

$$[\beta_i] = [\tau_2(c_i)] \quad \textit{for } i = 1, \ldots, m,$$
$$[\alpha_i] = [\mathcal{L}(p_i)] \quad \textit{for } i = 0, \ldots, n.$$

Applying Definition 4.12 to $\mathcal{G}_{a^n b^n}$ yields the program $P_{\mathcal{G}_{a^n b^n}}$ defined in Example 4.2. $P_{\mathcal{G}_{a^n b^n}}$ is known as a *single-rule program*, or *sirup* for short (Gottlob and Papadimitriou 2003). Sirups are defined as consisting of a single rule along with several facts, which can be either grounded or ungrounded. In this particular case, the rule r_2 generates an ungrounded intensional fact.

Example 4.2 ($P_{\mathcal{G}_{a^n b^n}}$).

$$[r_1] \quad \mathsf{S}(x_1, x_4) \leftarrow \mathsf{S}(x_2, x_3), a(x_1, x_2), b(x_3, x_4).$$
$$[r_2] \quad \mathsf{S}(x_1, x_1).$$

Having seen how rules are transformed in Example 4.2, we now formalize the connection between rule instantiation and type derivation with the following lemma. This relationship is crucial for establishing the correctness of the encoding.

Lemma 4.2 (Rule Instantiation and Typing Derivation)**.** *For any valid instantiation of a rule c in $P_{\mathcal{G}}$ of the form*

$$[c] \quad p_0(\vec{s}_0) \leftarrow p_1(\vec{s}_1), \ldots, p_n(\vec{s}_n), c_1(\vec{t}_1), \ldots, c_m(\vec{t}_m).$$

where $\vec{s}_0, \vec{s}_1, \ldots, \vec{s}_n, \vec{t}_1, \ldots, \vec{t}_m$ are sequences of Datalog constants. The following holds:

$$y_1 : [\tau_2(c_1)](\vec{t}_1), \ldots, y_m : [\tau_2(c_m)](\vec{t}_m) \vdash \widehat{\mathcal{L}(c)}[y_1, \ldots, y_m] : [\mathcal{L}(p_1)](\vec{s}_1) \multimap \cdots \multimap [\mathcal{L}(p_n)](\vec{s}_n) \multimap [\mathcal{L}(p_0)](\vec{s}_0).$$

4.2.2 Instance Generation

Having established how to construct a Datalog program from a second-order ACG, we now address parsing object terms using Datalog evaluation over a query. This requires transforming the object term into an extensional database of facts that the Datalog program can process. The following definition formalizes this transformation, showing how to construct both the input instance and its associated query.

Definition 4.13 (Instance Generation)**.** Let $t \in T(\Sigma_2)$ be a term in η-long form such that $\vdash_{\Sigma_2} t : \mathcal{L}(p)$ for some $p \in A_1$. Let $(c_1, \ldots, c_m) = \overrightarrow{\mathrm{cons}}(t)$ be the sequence of constants occurring in t.

Let $\widehat{t}$ be the pure term obtained by replacing each constant c_i in t with a fresh variable y_i. The principal type of $\widehat{t}$ is given by the judgment:

$$y_1 : \beta_1, \ldots, y_m : \beta_m \vdash_{PT} \widehat{t} : \alpha$$

The instance $\mathbf{I}_t$ and query $query(t)$ associated with t are defined as:

$$\begin{aligned} \mathbf{I}_t &\coloneqq \{c_i(\overline{\beta_i}) \mid 1 \leq i \leq m\} \\ query(t) &\coloneqq p(\overline{\alpha}) \end{aligned}$$

Let us examine two properties of the instance construction. First, multiple occurrences of the same constant in the object term are treated as distinct in the pure term $\widehat{t}[y_1, \ldots, y_m]$. Second, the type variables from the principal typing become Datalog constants in the generated instance.

The application of Definition 4.13 to the η-long normal form of $t = a + a + b + b$ yields the instance $\mathbf{I}_{a^2b^2}$ shown in Example 4.3. The query defined over t is equal to $\mathsf{S}(0, 4)$.

Example 4.3 ($\mathbf{I}_{a^2b^2}$).

$$\begin{aligned} &a(0, 1). \\ &a(1, 2). \\ &b(2, 3). \\ &b(3, 4). \end{aligned}$$

Theorem 4.3 (Reduction (Kanazawa 2007, 2017))**.** *Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be a second-order linear ACG and let $p \in A_1$, If $v \in T(\Sigma_2)$ is a linear term in η-long β-normal form relative to the interpretation of p then the following are equivalent:*

1. *it exists a linear λ-term, u, defined over Σ_1 such that $\vdash_{\Sigma_1} u : p$ and $\mathcal{L}(u) \rhd_\beta v$;*
2. *$query(v)$ belongs to $P_{\mathcal{G}}(\mathbf{I})$.*

4.2.3 Datalog Proof Trees and Antecedents

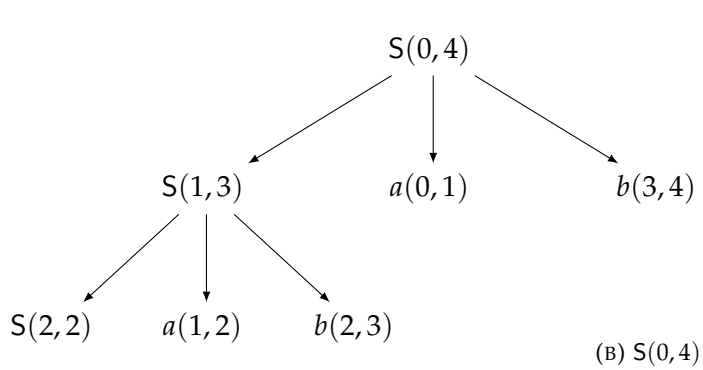
The seminaive evaluation of $P_{\mathcal{G}_{a^nb^n}}$ applied to $\mathbf{I}_{a^2b^2}$ produces the output $P_{\mathcal{G}_{a^nb^n}}(\mathbf{I}_{a^2b^2})$ shown in Example 4.4. By Theorem 4.3, the derivability of query $\mathsf{S}(0, 4)$? confirms that $t = a + a + b + b$ is in the object language of $\mathcal{G}_{a^nb^n}$.

Example 4.4 ($P_{\mathcal{G}_{a^n b^n}}(\mathbf{I}_{a^2b^2})$).

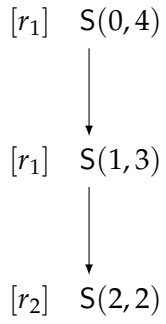
$\mathsf{S}(0,0).$ $\mathsf{S}(3,3).$
$\mathsf{S}(1,1).$ $\mathsf{S}(4,4).$
$\mathsf{S}(2,2).$ $\mathsf{S}(1,3).$
$\mathsf{S}(0,4).$

Given a term t in the object language, finding its abstract language antecedents requires analyzing proof trees where the query is the root node. Let's demonstrate this process by constructing a proof tree for $\mathsf{S}(0,4)$, illustrated in Figure 4.1a. The annotated proof tree representation (Figure 4.1b) shows the rule identifiers while omitting extensional facts. From this tree, we can derive the abstract λ-term $r_1\ (r_1\ r_2)$ by following the its structure. Each non-leaf node labeled with its abstract constant r_i creates an application of r_i to the λ-terms derived from its children proof trees.

In the general case, there is a possibility of generating an infinite number of proof trees, making it essential to use a structure, such as a shared forest (Billot and Lang 1989), to store them efficiently in a compressed, finite form. Currently, state-of-the-art Datalog engines like *soufflé* (Jordan, Scholz, and Subotić 2016) primarily function as solvers rather than provers, and therefore, they lack the capability to generate the complete set of proof trees. The toolkit to manipulate ACGs, *ACGtk* (Pogodalla 2016), includes a Datalog prover, allowing for the comprehensive construction and examination of proof trees.



(A) $\mathsf{S}(0,4)$ proof tree



(B) $\mathsf{S}(0,4)$ annotated proof tree without extensional facts

FIGURE 4.1: Proof trees for $\mathsf{S}(0,4)$

This reduction enables the application of established Datalog optimization techniques. The *generalized supplementary magic-sets rewriting method* (Beeri and Ramakrishnan 1987) can transform the program to achieve Earley-style (Earley 1970) parsing behavior. While this approach works efficiently for compact grammars like $\mathcal{G}_{a^n b^n}$ (Guillaume, Pogodalla, and Tourneur 2024), it is worth noting that for larger grammars it can generate a large number of rules, which may in turn impact parsing efficiency. Additionally, when using program rewriting optimizations, reconstructing the original proof tree from the rewritten program's proof tree becomes necessary for antecedent identification.

4.3 Extensions of the Reduction

To extend Kanazawa's reduction to more general cases, two main directions have been pursued:

1. Relaxing the linearity constraints;
2. Developing parsing methods for ACGs beyond second-order.

4.3.1 Beyond Linear λ-terms

Focusing on the first direction, extensions of this construction (Bourreau 2012; Kanazawa 2007) have been developed for two key classes of λ-terms.

Definition 4.14 (Almost Linear λ-terms (Kanazawa 2007))**.** A λ-term t is said to be *almost linear* if t is a $\lambda\mathbf{I}$-term and has a principal deduction $\mathcal{D}$ such that the following condition holds: For every subterm u of the λ-term t, if a variable x appears free in u more than once, then x must be assigned an atomic type in $\mathcal{D}$.

Definition 4.15 (Almost Affine λ-terms (Kanazawa 2012))**.** A λ-term t is said to be *almost affine* if it has a principal deduction $\mathcal{D}$ that satisfies the following conditions:

1. For every subterm u of the λ-term t, if a variable x appears free in u more than once, then x must be assigned an atomic type in $\mathcal{D}$.
2. For every subterm $\lambda x.\, u$ of the λ-term t, if the variable x does not occur in u, then x must also be assigned an atomic type in $\mathcal{D}$.

Linear λ-terms are a stricter subset of almost linear λ-terms, which in turn are a stricter subset of almost affine λ-terms. Their typings for both linear and almost linear classes are *negatively non-duplicating*, a property satisfied if condition 2 of Definition 4.2 holds. Consequently, the following extension of coherence theorem applies to these classes.

Theorem 4.4 (Coherence (Aoto and Ono 1994))**.** *All inhabitants of a negatively non-duplicated type assignment are equal under $\beta\eta$-equivalence.*

Extending the reduction to these two classes of terms is a non-trivial endeavor as challenges arise with both restricted duplication and deletion:

- When handling restricted duplication, we cannot treat occurrences of the same constant as distinct free variables during instance construction, since constants may be duplicated through β-reduction.
- Under restricted deletion, introducing a restricted form of intersection types becomes necessary, as it is impossible to find a negatively non-duplicating typing of an almost affine term t in β-normal form that applies to all almost affine λ-terms that are $\beta\eta$-convertible to t.

4.3.2 Beyond Second-Order

For the second direction, De Groote (de Groote 2015) introduced the first correct and semi-complete algorithm for parsing beyond second-order. This approach is formally captured as a proof-search problem within the MELL fragment, a formulation that generalizes Kanazawa's reduction. The computational complexity of this approach is thus contingent on the decidability of MELL, which remains an open question. As a result, the general complexity of parsing beyond second-order is still unknown, and current research is focused on developing parsing algorithms for specific, more tractable fragments.

Part III

Feature Structure Integration

Chapter 5

A λ-Calculus for Modelling Feature Structures

Having established the theoretical foundations of ACGs and parsing algorithms for second-order ACGs, we now turn our attention to the topic of integrating feature structures. To do this, we will first examine how a grammar formalism augmented with feature structures can be encoded into a core ACG.

Building on the insights gained from this examination, we will then introduce a new λ-calculus, which we call $\lambda^{\multimap\mathcal{F}\Pi}$. This calculus is designed to provide a precise way of handling feature structures. We will develop $\lambda^{\multimap\mathcal{F}\Pi}$ step-by-step, starting from the simply typed linear calculus and gradually extending it to incorporate the necessary mechanisms for modeling and manipulating feature structures.

5.1 Motivating Feature Structures in ACGs

5.1.1 A Motivating Example: CFG Augmented with Feature Structures

We begin by examining the encoding of a simple yet illustrative example: a CFG augmented with feature structures that models English subject-verb agreement.

Example 5.1 (CFG).

$$\begin{aligned} \mathsf{S} &\to \mathsf{NP}[NUM =?n]\ \mathsf{VP}[NUM =?n] \\ \mathsf{NP}[NUM =?n] &\to \mathsf{Det}[NUM =?n]\ \mathsf{N}[NUM =?n] \\ \mathsf{VP}[NUM =?n] &\to \mathsf{V}[NUM =?n]\ \mathsf{NP} \end{aligned}$$

$$
\begin{aligned}
\mathsf{Det} &\rightarrow \textit{the} \\
\mathsf{N}[NUM = sg] &\rightarrow \textit{stakeholder} \mid \textit{stock} \\
\mathsf{N}[NUM = pl] &\rightarrow \textit{stakeholders} \mid \textit{stocks} \\
\mathsf{V}[NUM = sg] &\rightarrow \textit{sells} \\
\mathsf{V}[NUM = pl] &\rightarrow \textit{sell}
\end{aligned}
$$

The grammar in Example 5.1 enforces number agreement between subjects and verbs while allowing objects to vary independently in number. This enables the generation of grammatical sentences such as *the stakeholder sells the stock* and *the stakeholders sell the stocks* and prevents ungrammatical combinations.

In this augmented CFG notation, non-terminal symbols are extended with feature structures enclosed in square brackets. These structures consist of keys (e.g., NUM), which represent linguistic attributes, and their corresponding values. Values can be atomic (like *sg* for singular or *pl* for plural) or variables (indicated by a question mark, e.g., $?n$).

The primary function of these variables is to enforce agreement: when the same variable appears across different constituents in a rule, it signifies that their respective feature values must match. Conversely, the omission of a feature indicates underspecification, meaning that constituent is neutral with respect to that particular attribute.

5.1.2 Limitations of Direct ACG Encoding

To encode a feature-augmented CFG into an ACG, we must first transform the grammar into a traditional CFG without feature structures through a process called *feature structure expansion*. This creates specialized non-terminals for each feature structure combination, allowing us to apply the standard ACG encoding from Section 3.3, which defines an abstract signature for production rules, an object signature for strings, and a lexicon for interpreting rules into strings. Example 5.2 demonstrates this expansion process on our example grammar.

Example 5.2 (Feature Structure Expansion of Example 5.1).

$$\begin{aligned} \mathsf{S} &\to \mathsf{NP}_{sg}\ \mathsf{VP}_{sg} \\ \mathsf{NP}_{sg} &\to \mathsf{Det}_{sg}\ \mathsf{N}_{sg} \\ \mathsf{VP}_{sg} &\to \mathsf{V}_{sg}\ \mathsf{NP}_{sg} \\ \mathsf{VP}_{pl} &\to \mathsf{V}_{pl}\ \mathsf{NP}_{sg} \end{aligned} \qquad \begin{aligned} \mathsf{S} &\to \mathsf{NP}_{pl}\ \mathsf{VP}_{pl} \\ \mathsf{NP}_{pl} &\to \mathsf{Det}_{pl}\ \mathsf{N}_{pl} \\ \mathsf{VP}_{sg} &\to \mathsf{V}_{sg}\ \mathsf{NP}_{pl} \\ \mathsf{VP}_{pl} &\to \mathsf{V}_{pl}\ \mathsf{NP}_{pl} \end{aligned}$$

$$\begin{aligned} \mathsf{Det}_{sg} &\to \textit{the} \\ \mathsf{N}_{sg} &\to \textit{stakeholder} \\ \mathsf{N}_{pl} &\to \textit{stakeholders} \\ \mathsf{V}_{sg} &\to \textit{sells} \end{aligned} \qquad \begin{aligned} \mathsf{Det}_{pl} &\to \textit{the} \\ \mathsf{N}_{sg} &\to \textit{stock} \\ \mathsf{N}_{pl} &\to \textit{stocks} \\ \mathsf{V}_{pl} &\to \textit{sell} \end{aligned}$$

The ACG encoding of Example 5.2 yields the ACG $\mathcal{G}_{\mathsf{CFG}} = \langle \Sigma_{\mathsf{CFG}}, \Sigma_{\mathsf{String}}, \mathcal{L}_{\mathsf{CFG}}, \mathsf{S} \rangle$, where Σ_{CFG} is defined in Example 5.3, Σ_{String} in Example 5.4, and $\mathcal{L}_{\mathsf{CFG}}$ in Example 5.5.

Example 5.3 (Σ_{CFG}).

$$\begin{aligned} \mathsf{NP}_{sg} &:: \mathsf{Type} \\ \mathsf{VP}_{sg} &:: \mathsf{Type} \\ \mathsf{Det}_{sg} &:: \mathsf{Type} \\ \mathsf{N}_{sg} &:: \mathsf{Type} \\ \mathsf{V}_{sg} &:: \mathsf{Type} \\ \mathsf{S} &:: \mathsf{Type} \end{aligned} \qquad \begin{aligned} \mathsf{NP}_{pl} &:: \mathsf{Type} \\ \mathsf{VP}_{pl} &:: \mathsf{Type} \\ \mathsf{Det}_{pl} &:: \mathsf{Type} \\ \mathsf{N}_{pl} &:: \mathsf{Type} \\ \mathsf{V}_{pl} &:: \mathsf{Type} \end{aligned}$$

$$\begin{aligned} \textsc{sentence}_{sg} &: \mathsf{NP}_{sg} \multimap \mathsf{VP}_{sg} \multimap \mathsf{S} \\ \textsc{noun_phrase}_{sg} &: \mathsf{Det}_{sg} \multimap \mathsf{N}_{sg} \multimap \mathsf{NP}_{sg} \\ \textsc{verb_phrase}_{sg} &: \mathsf{V}_{sg} \multimap \mathsf{NP}_{sg} \multimap \mathsf{VP}_{sg} \\ \textsc{the}_{sg} &: \mathsf{Det}_{sg} \\ \textsc{stakeholder} &: \mathsf{N}_{sg} \\ \textsc{stock} &: \mathsf{N}_{sg} \end{aligned} \qquad \begin{aligned} \textsc{sentence}_{pl} &: \mathsf{NP}_{pl} \multimap \mathsf{VP}_{pl} \multimap \mathsf{S} \\ \textsc{noun_phrase}_{pl} &: \mathsf{Det}_{pl} \multimap \mathsf{N}_{pl} \multimap \mathsf{NP}_{pl} \\ \textsc{verb_phrase}_{pl} &: \mathsf{V}_{pl} \multimap \mathsf{NP}_{pl} \multimap \mathsf{VP}_{pl} \\ \textsc{the}_{pl} &: \mathsf{Det}_{pl} \\ \textsc{stakeholders} &: \mathsf{N}_{pl} \\ \textsc{stocks} &: \mathsf{N}_{pl} \end{aligned}$$

$$\text{SELLS} : \mathsf{V}_{sg} \qquad \text{SELL} : \mathsf{V}_{pl}$$

Example 5.4 (Σ_{String}).

$$\begin{aligned} o &:: \mathsf{Type} \\ \mathsf{String} &= o \multimap o \\ + &= \lambda^{\circ} x\ y\ z.\ x\ (y\ z) \\ \textit{the} &: \mathsf{String} \\ \textit{stakeholder} &: \mathsf{String} \qquad \textit{stakeholders} : \mathsf{String} \\ \textit{stock} &: \mathsf{String} \qquad \textit{stocks} : \mathsf{String} \\ \textit{sells} &: \mathsf{String} \qquad \textit{sell} : \mathsf{String} \end{aligned}$$

Example 5.5 ($\mathcal{L}_{\mathsf{CFG}} : \Sigma_{\mathsf{rules}} \to \Sigma_{\mathsf{String}}$).

$$\begin{aligned} \mathsf{NP}_{sg} &\mapsto \mathsf{String} & \mathsf{NP}_{pl} &\mapsto \mathsf{String} \\ \mathsf{VP}_{sg} &\mapsto \mathsf{String} & \mathsf{VP}_{pl} &\mapsto \mathsf{String} \\ \mathsf{Det}_{sg} &\mapsto \mathsf{String} & \mathsf{Det}_{pl} &\mapsto \mathsf{String} \\ \mathsf{N}_{sg} &\mapsto \mathsf{String} & \mathsf{N}_{pl} &\mapsto \mathsf{String} \\ \mathsf{V}_{sg} &\mapsto \mathsf{String} & \mathsf{V}_{pl} &\mapsto \mathsf{String} \end{aligned}$$

$$\begin{aligned} \text{SENTENCE}_{sg} &\mapsto \lambda^{\circ} np\ vp.\ np + vp & \text{SENTENCE}_{pl} &\mapsto \lambda^{\circ} np\ vp.\ np + vp \\ \text{NOUN_PHRASE}_{sg} &\mapsto \lambda^{\circ} det\ n.\ det + n & \text{NOUN_PHRASE}_{pl} &\mapsto \lambda^{\circ} det\ n.\ det + n \\ \text{VERB_PHRASE}_{sg} &\mapsto \lambda^{\circ} v\ np.\ v + np & \text{VERB_PHRASE}_{pl} &\mapsto \lambda^{\circ} v\ np.\ v + np \end{aligned}$$

$$\begin{aligned} \text{THE}_{sg} &\mapsto \textit{the} & \text{THE}_{pl} &\mapsto \textit{the} \\ \text{STAKEHOLDER} &\mapsto \textit{stakeholder} & \text{STAKEHOLDERS} &\mapsto \textit{stakeholders} \\ \text{STOCK} &\mapsto \textit{stock} & \text{STOCKS} &\mapsto \textit{stocks} \\ \text{SELLS} &\mapsto \textit{sells} & \text{SELL} &\mapsto \textit{sell} \end{aligned}$$

The encoding of our feature-augmented CFG into Σ_{CFG} reveals fundamental limitations in ACGs' handling of feature structures. First, we see a proliferation of types, requiring separate type definitions for each feature variation of a syntactic category (e.g.,

NP_{sg} and NP_{pl} for singular and plural noun phrases). Second, this leads to a multiplication of constants, as we need distinct rules for each feature combination (e.g., separate SENTENCE constants for singular and plural agreement). These limitations stem from a core issue: ACGs lack direct support for feature structures within types. Since ACGs model syntactic categories as types (de Groote 2001, 2002; Pogodalla 2017a), the natural solution is to extend the type system itself with a modelling of feature structures.

5.2 Modelling Feature Structures

5.2.1 Atomic And Compound Feature Structures

In contrast to logic-based approaches to feature structures (Carpenter 1992), we adopt a type-theoretic perspective using standard type constructors. The first challenge in representing feature structures is finding an appropriate way to model atomic feature values. For this purpose, we propose using *enumerations* as our atomic building blocks.

Definition 5.1 (Enumeration)**.** Let E be a finite set of symbols called *enumerals*. An *enumeration* is a finite subset of E, written as:

$$\{e_1 \mid \ldots \mid e_n\},$$

where $e_1, \ldots, e_n \in E$ are distinct enumerals.

Enumerations are a degenerate form of sum types where each variant stands as a distinct value without carrying additional data. They provide a way to define atomic feature structures through explicit enumeration of their possible values.

Example 5.6 illustrates the use of enumerations to model basic morphological features in French. For readability, we will use shorthand conventions like GENDER and the equals sign $(=)$. While convenient for discussion, this notation will not be part of the formal system.

Example 5.6 (Atomic Feature Structures for French Agreement)*.*

$$\begin{aligned} \text{GENDER} &= \{masc \mid fem\} \\ \text{NUMBER} &= \{sg \mid pl\} \end{aligned}$$

While enumerations provide a mechanism for representing atomic features, linguistic descriptions often require more complex structures for bundling related features together. In linguistics, these bundled structures are typically represented as Attribute-Value Matrices (AVMs). To model AVMs in our system, we introduce *records*, which allow multiple features to be grouped into cohesive units. The use of records in linguistic type theory is not new (Cooper 2005, 2008); our approach is more restricted, as we aim to integrate them within a linear λ-calculus for ACGs while preserving tractability.

A *record type* specifies the structure of these feature bundles by declaring a set of labeled fields, each associated with a *feature structure type*. For example, using the atomic features from Example 5.6, we can define the record type $\text{AGREEMENT} = [\texttt{gend} : \text{GENDER}, \texttt{num} : \text{NUMBER}]$ which captures the core agreement properties in French morphosyntax.

To define the vocabulary for constructing these records, we introduce feature structure signatures. A feature structure signature lays out three essential components: the atomic feature values, how these enumerals are grouped into meaningful sets, and the field labels used to construct records.

Definition 5.2 (Feature Structure Signature)**.** A *feature structure signature* is a tuple $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ where:

1. E is a finite set of enumerals;
2. $\mathcal{E}$ is a partition of E into disjoint enumerations;
3. L is a set of labels;
4. $L \cap E = \emptyset$.

Record types can then be constructed by associating labels with feature structure types:

Definition 5.3 (Feature Structure Types)**.** Let $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ be a feature structure signature. The set of *feature structure types* defined over $\mathfrak{F}$, denoted by $\mathcal{F}(\mathfrak{F})$, is defined inductively as follows:

1. Every enumeration $\varepsilon \in \mathcal{E}$ is a feature structure type;
2. If $l_1, \ldots, l_n \in L$ are distinct labels and $\Phi_1, \ldots, \Phi_n \in \mathcal{F}(\mathfrak{F})$ are feature structure types, then the record type $[l_1 : \Phi_1, \ldots, l_n : \Phi_n]$ is a feature structure type.

Notably, this definition does not bind labels to a fixed type globally. A single label can therefore be associated with different feature structure types across various record definitions.

While feature structure types provide the schemas for linguistic descriptions, we need concrete instantiation of these schemas. This is precisely the role of feature structure terms. They offer a syntax for building complex combinations of feature structures and, importantly, provide operations for accessing and manipulating their individual components.

Definition 5.4 (Feature Structure Terms)**.** Let $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ be a feature structure signature and $\mathcal{X}_{\mathcal{F}}$ be a countable set of feature structure variables. The set of *feature structure terms*, denoted $F(\mathfrak{F})$, is the smallest set satisfying the following rules:

1. If $x \in \mathcal{X}_{\mathcal{F}}$, then $x \in F(\mathfrak{F})$;
2. If $e \in E$, then $e \in F(\mathfrak{F})$;
3. If $l_1, \ldots, l_n \in L$ are distinct labels, and $f_1, \ldots, f_n \in F(\mathfrak{F})$, then the record $[l_1 = f_1, \ldots, l_n = f_n] \in F(\mathfrak{F})$;
4. If $f \in F(\mathfrak{F})$ and $l \in L$, then the selection $f.l \in F(\mathfrak{F})$.

Building on the previous French agreement example, we can construct feature structure terms to represent specific agreement patterns:

$$[\texttt{gend} = \mathit{fem}, \texttt{num} = \mathit{sg}]$$
$$[\texttt{gend} = \mathit{fem}, \texttt{num} = \mathit{pl}]$$
$$[\texttt{gend} = \mathit{masc}, \texttt{num} = \mathit{sg}]$$
$$[\texttt{gend} = \mathit{masc}, \texttt{num} = \mathit{pl}].$$

These terms instantiate the schema defined by the agreement type. The selection operator allows us to extract individual features from a record. For instance, selecting the gender feature from $[\texttt{gend} : \mathit{masc}, \texttt{num} : \mathit{sg}].\texttt{gend}$ evaluates to *masc*.

For conciseness, we introduce the following notations:

- $[l_i = f_i]_{i=1}^n$ denotes a record with n fields, where each field has a unique label l_i for $i \in \{1, \ldots, n\}$ and an associated feature structure term F_i;

- $[l_i : \Phi_i]_{i=1}^n$ represents a record type with n type fields, where each field has a unique label l_i for $i \in \{1, \ldots, n\}$ and an associated type Φ_i.

For the remainder of this thesis, we assume record labels are sorted lexicographically. This simplification ensures each record has a unique representation and enables direct field-by-field comparison of records.

Having defined the syntax of feature structures, we now clarify their key structural properties. Our feature structures support *recursion*: record field values can themselves be records, allowing arbitrarily nested structures such as $[\text{AGREEMENT} = [\texttt{gend} = \textit{masc}, \texttt{num} = \textit{sg}]]$. However, our formalism deliberately excludes two features common in unification-based frameworks: *reentrancies* (structure sharing where multiple paths lead to the same substructure) and *cycles* (where a feature structure contains itself through a chain of field selections). These restrictions are motivated by our goal of maintaining a tractable computational model suitable for parsing algorithms. In traditional feature structure frameworks, reentrancies enable efficient representation of shared information and cycles allow certain recursive linguistic phenomena to be modeled directly. By disallowing these constructs, we obtain feature structures that are purely tree-structured, ensuring that all feature structure terms are finite and acyclic. This design choice aligns with our type-theoretic approach: feature structures in $\lambda^{\multimap\mathcal{F}\Pi}$ are simply well-founded terms built from enumerations and records, with no notion of pointer equality or graph-based unification.

5.2.2 Feature Structure Annotations as Type Families

In formalisms equipped with feature structures, there are two primary approaches to representing syntactic categories: one may either treat them directly as feature structures (as in HPSG (Pollard and Sag 1994)) or augment them with feature structure annotations (as in LFG (Kaplan and Bresnan 1982) and TAG (Vijay-Shanker and Joshi 1988)). In this work, we adopt the latter approach by developing a mechanism to annotate syntactic categories with feature structures.

To accomplish this, we adopt the concept of type families from dependent type theory (Martin-Löf and Sambin 1984), which provides a formal basis for making types dependent on λ-terms. A type family can be understood as a type-level function: it is a type parameterized by λ-terms that produces a concrete type only when supplied with specific feature structure arguments.

In the spirit of the indexed types approach developed by Zenger (1997) and Xi and Pfenning (1999), we deliberately restrict this type dependency to solely feature structure

terms. This design choice is crucial for ensuring that our system for syntactic category annotations remains both focused and tractable.

The addition of type families in our system introduces a new syntactic level, *kinds*, along with their corresponding kinding operator (::), which regulates the interaction between type families and feature structure terms. Just as types govern the well-formedness of terms, kinds stipulate the permissible combinations of feature structure terms with types, ensuring a coherent system of type parameterization.

Definition 5.5 (Kinds in $\lambda^{\multimap\mathcal{F}\Pi}$)**.** Let $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ be a feature structure signature. The set of kinds over $\mathfrak{F}$, denoted $\mathcal{K}(\mathfrak{F})$, is inductively defined as follows:

1. $\mathsf{Type} \in \mathcal{K}(\mathfrak{F})$;
2. If $k \in \mathcal{K}(\mathfrak{F})$ and $\Phi \in \mathcal{F}(\mathfrak{F})$, then $\Phi \Rrightarrow k \in \mathcal{K}(\mathfrak{F})$.

The kind Type identifies proper types. In contrast, the kind $\Phi \Rrightarrow k$ characterizes types that expect a feature structure. For example, the declaration $\mathsf{NP} :: \textsc{number} \Rrightarrow \mathsf{Type}$ states that noun phrases are a type family parameterized by a feature structure term typed by NUMBER.

Noun phrases can therefore be annotated with a number via *type applications* such as $\mathsf{NP}\,sg$ and $\mathsf{NP}\,pl$ for singular and plural forms, respectively. More generally, a type application has the form $\alpha\, f$, where α is a type and f is a feature structure.

Example 5.7 (Σ_{CFG} with type families).

$$
\begin{aligned}
\mathsf{S} &:: \mathsf{Type} \\
\mathsf{Det}, \mathsf{N}, \mathsf{NP}, \mathsf{V}, \mathsf{VP} &:: \textsc{number} \Rrightarrow \mathsf{Type} \\
\textsc{sentence}_{sg} &: \mathsf{NP}\,sg \multimap \mathsf{VP}\,sg \multimap \mathsf{S} \\
\textsc{sentence}_{pl} &: \mathsf{NP}\,pl \multimap \mathsf{VP}\,pl \multimap \mathsf{S} \\
\textsc{noun_phrase}_{sg} &: \mathsf{Det}\,sg \multimap \mathsf{N}\,sg \multimap \mathsf{NP}\,sg \\
\textsc{noun_phrase}_{pl} &: \mathsf{Det}\,pl \multimap \mathsf{N}\,pl \multimap \mathsf{NP}\,pl \\
\textsc{verb_phrase}_{sg,sg} &: \mathsf{V}\,sg \multimap \mathsf{NP}\,sg \multimap \mathsf{VP}\,sg \\
\textsc{verb_phrase}_{sg,pl} &: \mathsf{V}\,sg \multimap \mathsf{NP}\,pl \multimap \mathsf{VP}\,sg \\
\textsc{verb_phrase}_{pl,pl} &: \mathsf{V}\,pl \multimap \mathsf{NP}\,pl \multimap \mathsf{VP}\,pl \\
\textsc{verb_phrase}_{pl,sg} &: \mathsf{V}\,pl \multimap \mathsf{NP}\,sg \multimap \mathsf{VP}\,pl \\
\textsc{the}_{sg} &: \mathsf{Det}\,sg
\end{aligned}
$$

$$\begin{aligned}
\text{THE}_{pl} &: \mathsf{Det}\ pl \\
\text{STAKEHOLDER, STOCK} &: \mathsf{N}\ sg \\
\text{STAKEHOLDERS, STOCKS} &: \mathsf{N}\ pl \\
\text{SELLS} &: \mathsf{V}\ sg \\
\text{SELL} &: \mathsf{V}\ pl
\end{aligned}$$

This approach is demonstrated in Example 5.7 which revises the higher-order signature Σ_{CFG}. Here, while S remains a proper atomic type ($:: \mathsf{Type}$), all other syntactic categories are treated as families of types ($:: \text{NUMBER} \Rrightarrow \mathsf{Type}$). This approach directly and robustly captures morphological agreement within the type system. For now, its sole advantage is eliminating redundant type declarations.

5.2.3 Variable Sharing as Dependent Products

While type families enable parameterization of types by feature structure terms, they alone cannot express constraints where multiple constituents must share the same feature values. Consider the agreement rule in our introductory example where both noun phrase and verb phrase must share an identical number feature:

$$\mathsf{S} \rightarrow \mathsf{NP}[NUM = ?n]\ \mathsf{VP}[NUM = ?n]$$

Using only type families in ACGs would force us to construct separate constants for each possible number feature:

$$\begin{aligned}
\text{SENTENCE}_{sg} &: \mathsf{NP}\ sg \multimap \mathsf{VP}\ sg \multimap \mathsf{S} \\
\text{SENTENCE}_{pl} &: \mathsf{NP}\ pl \multimap \mathsf{VP}\ pl \multimap \mathsf{S}
\end{aligned}$$

To address this limitation, we introduce *dependent products* (denoted by Π), which serve as universal quantifiers over feature structures within types. This mechanism allows us to combine multiple feature-specific constants into a single, more general constant:

$$\text{SENTENCE} : \Pi x : \text{NUMBER}.\ \mathsf{NP}\ x \multimap \mathsf{VP}\ x \multimap \mathsf{S},$$

where the dependent product $\Pi x : \text{NUMBER}.\ \mathsf{NP}\ x \multimap \mathsf{VP}\ x \multimap \mathsf{S}$ binds the variable x of type NUMBER in the types $\mathsf{NP}\ x$ and $\mathsf{VP}\ x$.

By incorporating dependent products, the abstract signature for our introductory grammar becomes considerably more compact, as illustrated in Example 5.8.

Example 5.8 (Σ_{CFG} with Dependent Products).

$$
\begin{aligned}
\mathsf{S} &:: \mathsf{Type} \\
\mathsf{Det}, \mathsf{N}, \mathsf{NP}, \mathsf{V}, \mathsf{VP} &:: \textsc{number} \Rrightarrow \mathsf{Type} \\
\textsc{sentence} &: \Pi x : \textsc{number}.\ \mathsf{NP}\ x \multimap \mathsf{VP}\ x \multimap \mathsf{S} \\
\textsc{noun_phrase} &: \Pi x : \textsc{number}.\ \mathsf{Det}\ x \multimap \mathsf{N}\ x \multimap \mathsf{NP}\ x \\
\textsc{verb_phrase} &: \Pi x : \textsc{number}.\ \Pi y : \textsc{number}.\ \mathsf{V}\ x \multimap \mathsf{NP}\ y \multimap \mathsf{VP}\ x \\
\textsc{the} &: \Pi x : \textsc{number}.\ \mathsf{Det}\ x \\
\textsc{stakeholder} &: \Pi x : \textsc{number}.\ \mathsf{N}\ x \\
\textsc{stock} &: \Pi x : \textsc{number}.\ \mathsf{N}\ x \\
\textsc{sell} &: \Pi x : \textsc{number}.\ \mathsf{V}\ x
\end{aligned}
$$

5.2.4 Interpretations with Feature Structures

The introduction of dependent products in Example 5.8 fundamentally changes how morphological variations are handled by shifting them from the abstract to the object level. This shift requires new interpretation mechanisms for the lexicon, specifically concerning feature structures.

When transitioning from abstract to object representations, three interpretive situations can arise:

1. *Feature Structure Erasure*: We may erase the attached feature structure during interpretation, treating it as abstract information that doesn't carry over to the object representation.

2. *Feature Structure Transfer*: We may preserve and transfer the feature structure from the abstract level to the object level, maintaining the morphosyntactic information throughout the interpretation process.

3. *Feature Structure Dependent Interpretation*: The interpretation itself may vary based on the specific feature structures, allowing different realizations depending on morphological properties.

Dependent products introduce two term constructors to manage these interpretations at the term level: feature structure application (denoted by $\bullet$ to distinguish it from linear application) and feature structure abstraction (using λ instead of λ°). These constructors address the first two interpretive situations at the term level. For instance, the abstract constant SENTENCE can be interpreted directly using feature structure abstraction as:

$$\text{SENTENCE} \mapsto \lambda x.\, \lambda^\circ np\ vp.\, np + vp$$

.

To handle these two first interpretive situations at the type level, we introduce a new type constructor: *type abstraction*. Denoted as $\Lambda x.\, \alpha$, this constructor allows a type to abstract over a feature structure parameter, x. For example, an abstraction like $\Lambda x.\, \mathsf{String}$ models feature erasure. It indicates that when used as an interpretation, this type abstracts over a number feature (x) to produce a string (String), discarding the feature information in the resulting type.

To model the third scenario, feature-dependent interpretation, we need a branching mechanism that operates at both the term and type levels. We introduce this capability through *case analysis*, which functions as a specialized form of abstraction, allowing a function's behavior to be defined piecewise based on the value of a feature structure.

At the term level, case analysis allows us to define a polymorphic function whose interpretation varies according to a specific feature. The syntax for this is $\mathsf{case}_x\ \{e_1 \to t_1 \mid \cdots \mid e_n \to t_n\}$. For example, the abstract constant 'STAKEHOLDER' requires different string realizations for singular and plural numbers. Using case analysis, its interpretation for the string object language becomes a function that maps a feature value to a specific term:

$$\text{STAKEHOLDER} \mapsto \mathsf{case}_x\ \{sg \to \textit{stakeholder} \mid pl \to \textit{stakeholders}\}$$

This expression is a form of abstraction by cases. Consequently, it has the dependent product type $\Pi x : \text{NUMBER}.\, \mathsf{String}$, signifying that it is a function that accepts a number feature and returns a string. As shown in the revised lexicon in Example 5.9, this approach is applied to all morphologically variable terms like 'STOCK' and 'SELL'.

Example 5.9 ($\mathcal{L}_{\mathsf{CFG}} : \Sigma_{\mathsf{rules}} \to \Sigma_{\mathsf{String}}$ revised in $\lambda^{\multimap_{\mathcal{F}}\Pi}$).

$$
\begin{aligned}
\mathsf{S} &\mapsto \mathsf{String} \\
\mathsf{NP}, \mathsf{VP}, \mathsf{Det}, \mathsf{N}, \mathsf{V} &\mapsto \Lambda x.\, \mathsf{String} \\
\textsc{sentence} &\mapsto \lambda x.\, \lambda^{\circ} np\ vp.\ np \ + \ vp \\
\textsc{noun_phrase} &\mapsto \lambda x.\, \lambda^{\circ} det\ n.\ det \ + \ n \\
\textsc{verb_phrase} &\mapsto \lambda x.\, \lambda y.\, \lambda^{\circ} v\ np.\ v \ + \ np \\
\textsc{the} &\mapsto \lambda x.\, \mathit{the}
\end{aligned}
$$

$$
\begin{aligned}
\textsc{stakeholder} &\mapsto \mathsf{case}_x\ \{sg \to \mathit{stakeholder} \mid pl \to \mathit{stakeholders}\} \\
\textsc{stock} &\mapsto \mathsf{case}_x\ \{sg \to \mathit{stock} \mid pl \to \mathit{stocks}\} \\
\textsc{sell} &\mapsto \mathsf{case}_x\ \{sg \to \mathit{sells} \mid pl \to \mathit{sell}\}
\end{aligned}
$$

To mirror the term-level construct, we introduce a corresponding constructor at the type level: *type case analysis*, written as $\mathsf{Case}_x\ \{e_1 \to \alpha_1 \mid \cdots \mid e_n \to \alpha_n\}$. This addition enables a form of *type-level programming* (in the spirit of, Blanvillain et al. 2022) and proves particularly powerful for defining the syntax-semantics interface. Specifically, it allows the interpretation of a syntactic type as a semantic type to be conditioned on its morphosyntactic features.

With an informal overview of our system's constructs complete, we now turn to the formalization of $\lambda^{\multimap_{\mathcal{F}}\Pi}$'s type syntax.

Definition 5.6 (Types in $\lambda^{\multimap_{\mathcal{F}}\Pi}$)**.** Let A be a finite set of atomic types, $\mathcal{X}_{\mathcal{F}}$ be a countable set of feature structure variables and let $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ be a feature structure signature. The set $\mathcal{T}(A, \mathfrak{F})$, consisting of types built upon A and $\mathfrak{F}$, is inductively defined as follows:

1. If $a \in A$, then $a \in \mathcal{T}(A, \mathfrak{F})$;
2. If $\alpha, \beta \in \mathcal{T}(A, \mathfrak{F})$, then $\alpha \multimap \beta \in \mathcal{T}(A, \mathfrak{F})$;
3. If $x \in \mathcal{X}_{\mathcal{F}}$ and $\alpha \in \mathcal{T}(A, \mathfrak{F})$, then $\Lambda x.\, \alpha \in \mathcal{T}(A, \mathfrak{F})$;
4. If $\alpha \in \mathcal{T}(A, \mathfrak{F})$ and $f \in F(\mathfrak{F})$, then $\alpha f \in \mathcal{T}(A, \mathfrak{F})$.
5. If $x \in \mathcal{X}_{\mathcal{F}}$, $e_1, \ldots, e_n \in F(\mathfrak{F})$ and $\alpha_1, \ldots, \alpha_n \in \mathcal{T}(A, \mathfrak{F})$ then $\mathsf{Case}_x\ \{e_1 \to \alpha_1 \mid \cdots \mid e_n \to \alpha_n\} \in \mathcal{T}(A, \mathfrak{F})$.

The reader may notice that we have not included dependent products in this definition. Studies by de Groote, Maarek, and Ryo Yoshinaka (2007) and Pompigne (2013) have demonstrated that unrestricted dependent products allow ACGs to encode arbitrary phrase structure grammars, rendering the membership problem computationally undecidable. To preserve tractability while maintaining the linguistic expressiveness of dependent products, we impose two restrictions:

1. Dependent products must appear exclusively in prenex form;
2. The set of arguments of a term typed by a dependent product must be a finite set of feature structures.

The prenex form requirement confines all feature dependencies to the outermost level of type expressions, thereby preventing deeply nested quantification. The finiteness constraint guarantees that all possible feature structures can be enumerated, making feature structure expansion feasible.

To formally enforce these restrictions, we introduce a stratified system with two new syntactic levels: *affix types* and *affix kinds*. This approach ensures that dependent products are only used in a controlled, prenex manner.

Definition 5.7 (Affix Types)**.** Let $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ be a feature structure signature. The set of *affix types* over A and $\mathfrak{F}$, denoted as $\mathcal{D}(A, \mathfrak{F})$, is defined as follows:

- If $\alpha \in \mathcal{T}(A, \mathfrak{F})$, then $\alpha \in \mathcal{D}(A, \mathfrak{F})$;
- If $\alpha \in \mathcal{D}(A, \mathfrak{F})$, then $\Pi x \colon \mathcal{F}(\mathfrak{F}). \alpha \in \mathcal{D}(A, \mathfrak{F})$.

Definition 5.8 (Affix Kinds)**.** Let $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ be a feature structure signature. The set of affix kinds over $\mathfrak{F}$, denoted as $\mathcal{K}_{\mathcal{D}}(\mathfrak{F})$, is defined as follows:

- DType is the kind for proper affix types and belongs to $\mathcal{K}_{\mathcal{D}}(\mathfrak{F})$;
- every $\kappa \in \mathcal{K}(\mathfrak{F})$ belongs to $\mathcal{K}_{\mathcal{D}}(\mathfrak{F})$.

By assigning a dependent product type to the case-analysis construct and enforcing a prenex dependent products, it is guaranteed that case-analysis expressions are restricted to the top level of any term. This design choice eliminates the need for the permutative conversions (as in, Pompigne 2013), which would otherwise be required to manage interactions between the case-analysis constructor and other term constructors.

5.3 A Feature-Structure Enriched Linear Lambda Calculus

5.3.1 Definition

We have now formally defined nearly all the syntactic expressions involved in $\lambda^{\multimap\mathcal{F}\Pi}$. What remains is to extend linear higher-order signatures and to define the term syntax.

Definition 5.9 (Raw Higher-Order Signature in $\lambda^{\multimap\mathcal{F}\Pi}$)**.** A *raw higher-order signature* Σ in $\lambda^{\multimap\mathcal{F}\Pi}$ is a tuple $\langle A, \mathfrak{F}, \kappa, C, \tau\rangle$ where:

1. A is a finite set of type families;
2. $\mathfrak{F} = \langle E, \mathcal{E}, L\rangle$ is a feature structure signature;
3. $\kappa : A \to \mathcal{K}(\mathfrak{F})$ is a kind assigning function;
4. C is a finite set of constants;
5. The sets A, C, E, and L are pairwise disjoint;
6. $\tau : C \to \mathcal{D}(A, \mathfrak{F})$ is an affix type assigning function.

Definition 5.10 (λ-terms in $\lambda^{\multimap\mathcal{F}\Pi}$)**.** Given a countable set of linear variables $\mathcal{X}$ and a countable set of feature structure variables $\mathcal{X}_\mathcal{F}$, the set of linear *λ-terms* defined over the raw signature $\Sigma = \langle A, \mathfrak{F} = \langle E, \mathcal{E}, L\rangle, \kappa, C, \tau\rangle$, $T(\Sigma)$ is defined by induction as follows:

1. If $c \in C$, then $c \in T(\Sigma)$;
2. If $x \in \mathcal{X}$, then $x \in T(\Sigma)$;
3. If $x \in \mathcal{X}$, $t \in T(\Sigma)$ and x occurs exactly once in t, then $\lambda^\circ x.\, t \in T(\Sigma)$;
4. If $x \in \mathcal{X}_\mathcal{F}$, $t \in T(\Sigma)$, then $\lambda x.\, t \in T(\Sigma)$;
5. If $t, u \in T(\Sigma)$, and their sets of free variables are disjoint ($\mathrm{FV}(t) \cap \mathrm{FV}(u) = \emptyset$), then $(t\, u) \in T(\Sigma)$;
6. If $t \in T(\Sigma)$ and $f \in F(\mathfrak{F})$ then $(t \bullet f) \in T(\Sigma)$;
7. If $x \in \mathcal{X}_\mathcal{F}$, $e_1, \ldots, e_n \in F(\mathfrak{F})$ and $t_1, \ldots, t_n \in T(\Sigma)$ then $\mathsf{case}_x\ \{e_1 \to t_1 \mid \cdots \mid e_n \to t_n\} \in T(\Sigma)$.

Assigning a type to a feature structure or a kind to a type requires an additional context for feature structures variables.

Definition 5.11 (Feature Structure Typing Context)**.** Let $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ be a feature structure signature. A *feature structure typing context* Γ is defined as a finite (possibly empty) set of feature structure type assignments of the form

$$\Gamma = \{x_1 : \Phi_1, \ldots, x_n : \Phi_n\},$$

where $n \geq 0$, $x_1, \ldots, x_n$ are distinct variables and $\Phi_1, \ldots, \Phi_n$ are feature structure types in $\mathcal{F}(\mathfrak{F})$.

The type system of $\lambda^{\multimap \mathcal{F}\Pi}$ requires handling both feature structures and linear terms simultaneously. This leads to a typing system that maintains two distinct contexts: Γ for feature structures and Δ for linear terms (where each assumption must be used exactly once). This double-context approach builds on established work on mixed linear/non-linear dependent type systems by Pfenning and Cervesato (1996), which incorporates linear types into the Edinburgh Logical Framework (Harper, Honsell, and Plotkin 1993).

The type system is defined through three primary interdependent judgment forms. The first is feature structure typing, denoted as $\Gamma \vdash_\Sigma f : \Phi$, which verifies that a feature structure term f has feature structure type Φ. This judgment uses only the feature structure context Γ. The second form is kinding, written as $\Gamma \vdash_\Sigma \alpha :: k$, which ensures types are well-formed at their assigned kinds. The feature structure context Γ is used to type any feature terms appearing in types. The third form is term typing, expressed as $\Gamma; \Delta \vdash_\Sigma t : \alpha$, which assigns types to terms using both contexts. This judgment form integrates both feature structure and linear typing rules.

These judgments are complemented by a fourth, the definitional equality judgment $\Gamma \vdash_\Sigma \alpha \equiv \alpha'$ which asserts equivalence between two types. This equality is integrated into the system through the TYPE-EQ rule. Similarly to the Edinburgh Logical Framework, we take β-conversion as the definitional equality.

With these judgments established, we now introduce feature structure substitution, which is necessary for the typing judgment as well as the evaluation rules in $\lambda^{\multimap \mathcal{F}\Pi}$.

Definition 5.12 (Feature Structure Substitution)**.** Let $x \in \mathcal{X}_\mathcal{F}$ be a feature structure variable, and let f be a feature structure term. The substitution of x with f is defined by induction on the structure of feature structure terms, types, and λ-terms. Throughout this

definition, z denotes a fresh variable chosen to avoid capture.

$$
\begin{aligned}
y[x := f] &= f \quad \text{if } y = x \\
y[x := f] &= y \quad \text{if } y \in \mathcal{X}_{\mathcal{F}} \text{ and } x \neq y \\
e[x := f] &= e \\
[l_i = f_i]_{i=1}^{n}[x := f] &= [l_i = f_i[x := f]]_{i=1}^{n} \\
(f'.l)[x := f] &= f'[x := f].l
\end{aligned}
$$

$$
\begin{aligned}
a[x := f] &= a \\
(\alpha \multimap \beta)[x := f] &= \alpha[x := f] \multimap \beta[x := f] \\
(\Lambda y.\, \alpha)[x := f] &= \Lambda y.\, \alpha[x := f] \quad \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(f) \\
(\Lambda y.\, \alpha)[x := f] &= \Lambda z.\, \alpha[y := z][x := f] \quad \text{if } x \neq y \text{ and } y \in \mathrm{FV}(f) \\
(\alpha\, g)[x := f] &= \alpha[x := f]\, g[x := f] \\
(\mathsf{Case}_y\, \{e_i \to \alpha_i\}_{i=1}^{n})[x := f] &= \mathsf{Case}_y\, \{e_i \to \alpha_i[x := f]\}_{i=1}^{n} \quad \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(f) \\
(\mathsf{Case}_y\, \{e_i \to \alpha_i\}_{i=1}^{n})[x := f] &= \mathsf{Case}_z\, \{e_i \to \alpha_i[y := z][x := f]\}_{i=1}^{n} \quad \text{if } x \neq y \text{ and } y \in \mathrm{FV}(f) \\
(\Pi y\colon \Phi.\, \alpha)[x := f] &= \Pi y\colon \Phi.\, \alpha[x := f] \quad \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(f) \\
(\Pi y\colon \Phi.\, \alpha)[x := f] &= \Pi z\colon \Phi.\, \alpha[y := z][x := f] \quad \text{if } x \neq y \text{ and } y \in \mathrm{FV}(f)
\end{aligned}
$$

$$
\begin{aligned}
y[x := f] &= y \quad \text{if } y \in \mathcal{X} \\
c[x := f] &= c \\
(\lambda^{\circ} y.\, t)[x := f] &= \lambda^{\circ} y.\, t[x := f] \\
(t_1\, t_2)[x := f] &= t_1[x := f]\, t_2[x := f] \\
(\lambda y.\, t)[x := f] &= \lambda y.\, t[x := f] \quad \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(f) \\
(\lambda y.\, t)[x := f] &= \lambda z.\, t[y := z][x := f] \quad \text{if } x \neq y \text{ and } y \in \mathrm{FV}(f) \\
(t \bullet f')[x := f] &= t[x := f] \bullet f'[x := f] \\
(\mathsf{case}_y\, \{e_i \to t_i\}_{i=1}^{n})[x := f] &= \mathsf{case}_y\, \{e_i \to t_i[x := f]\}_{i=1}^{n} \quad \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(f) \\
(\mathsf{case}_y\, \{e_i \to t_i\}_{i=1}^{n})[x := f] &= \mathsf{case}_z\, \{e_i \to t_i[y := z][x := f]\}_{i=1}^{n} \quad \text{if } x \neq y \text{ and } y \in \mathrm{FV}(f)
\end{aligned}
$$

System 3 provides a comprehensive specification of the calculus, encompassing its syntax, kinding rules, typing rules, and evaluation semantics.

System 3 (Linear lambda-calculus with feature structures, $\lambda^{\multimap \mathcal{F} \Pi}$)**.**

Syntax:

$\mathcal{F}(\mathfrak{F}) :=$		feature structure type:
	$\mathcal{E}$	*enumeration*
	$\mid [L_i : \mathcal{F}_i(\mathfrak{F})]_{i=1}^{n}$	*record type*
$F(\mathfrak{F}) :=$		feature structure term:
	E	*enumeral*
	$\mid \mathcal{X}_{\mathcal{F}}$	*feature structure variable*
	$\mid [L_i = F_i(\mathfrak{F})]_{i=1}^{n}$	*record*
	$\mid F(\mathfrak{F}).L$	*selection*
$\mathcal{K}(\mathfrak{F}) :=$		kind:
	Type	*kind of proper types*
	$\mid \mathcal{F}(\mathfrak{F}) \Rrightarrow \mathcal{K}(\mathfrak{F})$	*kind of type families*
$\mathcal{K}_{\mathcal{D}}(\mathfrak{F}) :=$		affix kind:
	DType	*kind of proper affix types*
	$\mid \mathcal{K}(\mathfrak{F})$	*kinds*
$\mathcal{T}(A, \mathfrak{F}) :=$		type:
	A	*type family*
	$\mid \mathcal{T}(A, \mathfrak{F}) \multimap \mathcal{T}(A, \mathfrak{F})$	*linear implicative type*
	$\mid \Lambda\mathcal{X}_{\mathcal{F}}.\mathcal{T}(A, \mathfrak{F})$	*type abstraction*
	$\mid (\mathcal{T}(A, \mathfrak{F})\, F(\mathfrak{F}))$	*type application*
	$\mid \mathsf{Case}_{\mathcal{X}_{\mathcal{F}}}\, \{E \to \mathcal{T}(A \mid \mathfrak{F}) \mid \cdots \mid E \to \mathcal{T}(A \mid \mathfrak{F})\}$	*type case analysis*
$\mathcal{D}(A, \mathfrak{F}) :=$		affix type:
	$\mid \mathcal{T}(A, \mathfrak{F})$	*type*
	$\mid \Pi\mathcal{X}_{\mathcal{F}} \colon \mathcal{F}(\mathfrak{F}).\mathcal{D}(A, \mathfrak{F})$	*dependent product*
$T(\Sigma) :=$		term:
	C	*constant*
	$\mid \mathcal{X}$	*linear variable*
	$\mid \lambda^{\circ}\mathcal{X}.T(\Sigma)$	*linear abstraction*
	$\mid (T(\Sigma)\, T(\Sigma))$	*linear application*

$$
\begin{array}{lll}
\mid & (T(\Sigma) \bullet F) & \textit{feature structure application} \\
\mid & \lambda \mathcal{X}_{\mathcal{F}}.\, T(\Sigma) & \textit{feature structure abstraction} \\
\mid & \mathsf{case}_{\mathcal{X}_{\mathcal{F}}}\ \{E \to T(\Sigma) \mid \cdots \mid E \to T(\Sigma)\} & \textit{case analysis}
\end{array}
$$

Notations:

$$
\begin{array}{l}
e \in E \\
\varepsilon \in \mathcal{E} \\
l \in L \\
f \in F(\mathfrak{F}) \\
\Phi \in \mathcal{F}(\mathfrak{F}) \\
k \in \mathcal{K}_{\mathcal{D}}(\mathfrak{F}) \\
a \in A \\
\alpha, \beta, \gamma \in \mathcal{D}(A, \mathfrak{F}) \\
c \in C \\
x \in \mathcal{X} \\
x \in \mathcal{X}_{\mathcal{F}} \\
t, u, v \in T(\Sigma)
\end{array}
$$

$\boxed{\Gamma \vdash_{\Sigma} f : \Phi}$

$$\frac{}{\Gamma, x : \Phi \vdash_{\Sigma} x : \Phi}\ \textsc{var-feat} \qquad \frac{e \in \varepsilon}{\Gamma \vdash_{\Sigma} e : \varepsilon}\ \textsc{enumeral-feat}$$

$$\frac{\forall i \in [1, n].\, \Gamma \vdash_{\Sigma} f_i : \Phi_i}{\Gamma \vdash_{\Sigma} [l_i = f_i]_{i=1}^{n} : [l_i : \Phi_i]_{i=1}^{n}}\ \textsc{record-feat}$$

$$\frac{\Gamma \vdash_{\Sigma} f : [l_1 : \Phi_1, \ldots, l_k : \Phi_k, \ldots, l_n : \Phi_n]}{\Gamma \vdash_{\Sigma} f.l_k : \Phi_k}\ \textsc{selection-feat}$$

$\boxed{\Gamma \vdash_\Sigma \alpha :: k}$

$$\frac{}{\Gamma \vdash_\Sigma a :: \kappa(a)}\ \text{ATOMIC-TYPE} \qquad \frac{\Gamma \vdash_\Sigma \alpha :: \mathsf{Type} \qquad \Gamma \vdash_\Sigma \beta :: \mathsf{Type}}{\Gamma \vdash_\Sigma \alpha \multimap \beta :: \mathsf{Type}}\ \text{LFUNC-TYPE}$$

$$\frac{\Gamma, x : \Phi \vdash_\Sigma \alpha :: k}{\Gamma \vdash_\Sigma \Lambda x.\, \alpha :: \Phi \Rrightarrow k}\ \text{ABSTR-TYPE} \qquad \frac{\Gamma \vdash_\Sigma \alpha :: \Phi \Rrightarrow k \qquad \Gamma \vdash_\Sigma f : \Phi}{\Gamma \vdash_\Sigma \alpha\, f :: k}\ \text{APP-TYPE}$$

$$\frac{\Gamma \vdash_\Sigma e_i : \varepsilon \qquad \Gamma, x : \varepsilon \vdash_\Sigma \alpha_i :: k}{\Gamma \vdash_\Sigma \mathsf{Case}_x\ \{e_i \to \alpha_i\}_{i=1}^n :: \varepsilon \Rrightarrow k}\ \text{CASE-TYPE}$$

$$\frac{\Gamma, x : \Phi \vdash_\Sigma \alpha :: \mathsf{DType}}{\Gamma \vdash_\Sigma \Pi x\colon \Phi.\, \alpha :: \mathsf{DType}}\ \text{PI-TYPE} \qquad \frac{\Gamma \vdash_\Sigma \alpha :: \mathsf{Type}}{\Gamma \vdash_\Sigma \alpha :: \mathsf{DType}}\ \text{PROMOTION-TYPE}$$

$\boxed{\Gamma; \Delta \vdash_\Sigma t : \alpha}$

$$\frac{\Gamma \vdash_\Sigma \alpha :: \mathsf{Type}}{\Gamma; x : \alpha \vdash_\Sigma x : \alpha}\ \text{LVAR-TERM} \qquad \frac{\vdash_\Sigma \tau(c) :: \mathsf{DType}}{\Gamma; \vdash_\Sigma c : \tau(c)}\ \text{CONST-TERM}$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda^\circ x.\, t : \alpha \multimap \beta}\ \text{LABSTR-TERM}$$

$$\frac{\Gamma; \Delta_1 \vdash_\Sigma t : \alpha \multimap \beta \qquad \Gamma; \Delta_2 \vdash_\Sigma u : \alpha \qquad \Delta_1 \cap \Delta_2 = \emptyset}{\Gamma; \Delta_1 \cup \Delta_2 \vdash_\Sigma t\, u : \beta}\ \text{LAPP-TERM}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma t : \Pi x\colon \Phi.\, \alpha \qquad \Gamma \vdash_\Sigma f : \Phi}{\Gamma; \Delta \vdash_\Sigma t \bullet f : \alpha[x := f]}\ \text{APP-TERM} \qquad \frac{\Gamma, x : \Phi; \Delta \vdash_\Sigma t : \alpha}{\Gamma; \Delta \vdash_\Sigma \lambda x.\, t : \Pi x\colon \Phi.\, \alpha}\ \text{ABSTR-TERM}$$

$$\frac{\Gamma \vdash_\Sigma e_i : \varepsilon \qquad \Gamma, x : \varepsilon; \Delta \vdash_\Sigma t_i : \alpha[x := e_i]}{\Gamma; \Delta \vdash_\Sigma \mathsf{case}_x\ \{e_i \to t_i\}_{i=1}^n : \Pi x\colon \varepsilon.\, \alpha}\ \text{CASE-TERM} \qquad \frac{\Gamma; \Delta \vdash_\Sigma t : \alpha \qquad \Gamma \vdash_\Sigma \alpha \equiv \alpha'}{\Gamma; \Delta \vdash_\Sigma t : \alpha'}\ \text{TYPE-EQ}$$

Evaluation Rules:

$$
\begin{aligned}
(\lambda^{\circ} x.\, t)\, u &\rhd_{1\beta} t[x := u] && \text{(linear } \beta\text{-contraction)} \\
(\Lambda x.\, \alpha)\, f &\rhd_{1\beta} \alpha[x := f] && \text{(type } \beta\text{-contraction)} \\
(\lambda x.\, t) \bullet f &\rhd_{1\beta} t[x := f] && (\beta\text{-contraction)} \\
(\mathsf{Case}_x\ \{e_i \to \alpha_i\}_{i=1}^{n}) \bullet e_i &\rhd_{1\beta} \alpha_i[x := e_i] && \text{(type case analysis)} \\
(\mathsf{case}_x\ \{e_i \to t_i\}_{i=1}^{n}) \bullet e_i &\rhd_{1\beta} t_i[x := e_i] && \text{(term case analysis)} \\
[l_i = f_i]_{i=1}^{n}.l_k &\rhd_{1\beta} f_k \text{ if } k \in [1, n] && \text{(record selection)}
\end{aligned}
$$

In LVAR-TERM, we require that the linear variable x is fresh with respect to Γ. Moreover, in both ABSTR-TERM and CASE-TERM, the variable x must not occur free in the linear context.

With the introduction of new terms, our system now includes additional forms of redexes. Consequently, the definitions of β-contraction, β-reduction, β-conversion, and β-expansion are extended accordingly.

5.3.2 Static Properties

This subsection examines some essential static properties of $\lambda^{\multimap_{\mathcal{F}}\Pi}$ that will be required for the theoretical developments in the subsequent chapter. We focus here on the key properties necessary for establishing our main results, building upon the comprehensive metatheoretical foundations developed by Pompigne (2013) for a generalized calculus of which $\lambda^{\multimap_{\mathcal{F}}\Pi}$ represents a degenerate form.

Structural Properties

We begin by establishing the fundamental substructural properties of $\lambda^{\multimap_{\mathcal{F}}\Pi}$. Due to the size and complexity of the system, which involves three interdependent judgment forms for feature structures, kinds, and terms, presenting the complete formal proofs for every case would be excessively lengthy. Therefore, we will only show a selection of representative cases to illustrate the proof techniques.

Lemma 5.1 (Permutation). 1. *if* $\Gamma_1, x : \Phi_1, y : \Phi_2 \vdash_{\Sigma} f : \Phi$ *and then* $\Gamma_1, y : \Phi_2, x : \Phi_1 \vdash_{\Sigma} f : \Phi$;

2. *if* $\Gamma_1, x : \Phi_1, y : \Phi_2 \vdash_{\Sigma} \alpha :: k$ *and then* $\Gamma_1, y : \Phi_2, x : \Phi_1 \vdash_{\Sigma} \alpha :: k$;

3. *if* $\Gamma_1, x : \Phi_1, y : \Phi_2; \Delta \vdash_\Sigma t : \alpha$ *and then* $\Gamma_1, y : \Phi_2, x : \Phi_1; \Delta \vdash_\Sigma t : \alpha$.

Proof. This is a direct consequence of the feature structure context, Γ, being defined as a set of assignments with distinct variables. □

Lemma 5.2 (Weakening)**.** *Let x be a variable such that $x \notin DOM(\Gamma)$. The following properties hold:*

1. *if* $\Gamma \vdash_\Sigma f : \Phi$ *then* $\Gamma, x : \Phi' \vdash_\Sigma f : \Phi$;
2. *if* $\Gamma \vdash_\Sigma \alpha :: k$ *then* $\Gamma, x : \Phi' \vdash_\Sigma \alpha :: k$;
3. *if* $\Gamma; \Delta \vdash_\Sigma t : \alpha$ *and x does not occurs free in* Δ *then* $\Gamma, x : \Phi'; \Delta \vdash_\Sigma t : \alpha$.

Proof. By simultaneous induction on the derivation of the first judgment.

Case. $\dfrac{\vdash_\Sigma \tau(c) :: \mathsf{DType}}{\Gamma; \vdash_\Sigma c : \tau(c)}$ CONST-TERM

1.	$\Gamma; \vdash_\Sigma c : \tau(c)$	by assumption
2.	$\vdash_\Sigma \tau(c) :: \mathsf{DType}$	by inversion
3.	$\Gamma \vdash_\Sigma \tau(c) :: \mathsf{DType}$	by repeated application of the ind. hyp.
4.	$\Gamma, x : \Phi' \vdash_\Sigma \tau(c) :: \mathsf{DType}$	by ind. hyp.
5.	$\Gamma, x : \Phi'; \vdash_\Sigma c : \tau(c)$	by CONST-TERM

Case. $\dfrac{\Gamma; \Delta \vdash_\Sigma t : \Pi y\colon \Phi.\, \alpha \qquad \Gamma \vdash_\Sigma f : \Phi}{\Gamma; \Delta \vdash_\Sigma t \bullet f : \alpha[y := f]}$ APP-TERM

1.	$\Gamma; \Delta \vdash_\Sigma t \bullet f : \alpha[y := f]$	by assumption
2.	$\Gamma; \Delta \vdash_\Sigma t : \Pi y\colon \Phi.\, \alpha$ $\Gamma \vdash_\Sigma f : \Phi$	by inversion
3.	$\Gamma, x : \Phi' \vdash_\Sigma t : \Pi y\colon \Phi.\, \alpha$ $\Gamma, x : \Phi' \vdash_\Sigma f : \Phi$	by ind. hyp.
4.	$\Gamma, x : \Phi'; \Delta \vdash_\Sigma t \bullet f : \alpha[y := f]$	by APP-TERM

Case. $\dfrac{\Gamma \vdash_\Sigma e_i : \varepsilon \qquad \Gamma, y : \varepsilon; \Delta \vdash_\Sigma t_i : \alpha[y := e_i]}{\Gamma; \Delta \vdash_\Sigma \mathsf{case}_y\ \{e_i \to t_i\}_{i=1}^n : \Pi y\colon \varepsilon.\, \alpha}$ CASE-TERM

1.	$\Gamma;\Delta \vdash_\Sigma \mathsf{case}_y\ \{e_i \to t_i\}_{i=1}^n : \Pi y\colon \varepsilon.\, \alpha$	by assumption
2.	$\Gamma \vdash_\Sigma e_i : \varepsilon$ $\Gamma, y : \varepsilon; \Delta \vdash_\Sigma t_i : \alpha[y := e_i]$	by inversion
3.	$\Gamma, x : \Phi' \vdash_\Sigma e_i : \varepsilon$ $\Gamma, y : \varepsilon, x : \Phi'; \Delta \vdash_\Sigma t_i : \alpha[y := e_i]$	by ind. hyp.
4.	$\Gamma, x : \Phi', y : \varepsilon; \Delta \vdash_\Sigma t_i : \alpha[y := e_i]$	by lemma 5.1
5.	$\Gamma, x : \Phi'; \Delta \vdash_\Sigma \mathsf{case}_y\ \{e_i \to t_i\}_{i=1}^n : \Pi y\colon \varepsilon.\, \alpha$	by CASE-TERM

The rest of the cases are similar to the above cases. □

Subject Reduction

Having established the basic structural properties, we now turn to the preservation of types under reduction, which ensures that well-typed terms remain well-typed throughout computation.

Lemma 5.3 (Substitutions)**.**

1. *If* $\Gamma, x : \Phi' \vdash f : \Phi$ *and* $\Gamma \vdash f' : \Phi'$ *then* $\Gamma \vdash f[x := f'] : \Phi$
2. *if* $\Gamma, x : \Phi \vdash \alpha :: k$ *and* $\Gamma \vdash f : \Phi$ *then* $\Gamma \vdash \alpha[x := f] :: k$
3. *If* $\Gamma, x : \Phi; \Delta \vdash t : \alpha$ *and* $\Gamma \vdash f : \Phi$ *then* $\Gamma; \Delta \vdash t[x := f] : \alpha[x := f]$

Proof. By simultaneous induction on the first judgment.

Case. $\dfrac{e \in \varepsilon}{\Gamma \vdash_\Sigma e : \varepsilon}$ ENUMERAL-FEAT

1.	$x : \Phi' \vdash_\Sigma e : \varepsilon$	by assumption
2.	$e[x := f'] = e$	by definition of substitution
3.	$\Gamma \vdash_\Sigma e[x := f'] : \varepsilon$	by ENUMERAL-FEAT

Case. $\dfrac{\Gamma;\Delta \vdash_\Sigma t : \Pi y\colon \Phi.\, \alpha \qquad \Gamma \vdash_\Sigma f : \Phi}{\Gamma;\Delta \vdash_\Sigma t \bullet f : \alpha[y := f]}$ APP-TERM

1.	$\Gamma, x : \Phi'; \Delta \vdash_\Sigma t \bullet f : \alpha[y := f]$	by assumption
2.	$\Gamma, x : \Phi'; \Delta \vdash_\Sigma t : \Pi y\colon \Phi.\, \alpha$ $\Gamma, x : \Phi' \vdash_\Sigma f : \Phi$	by inversion
3.	$\Gamma \vdash_\Sigma t[x := f'] : (\Pi y\colon \Phi.\, \alpha)[x := f']$ $\Gamma \vdash_\Sigma f[x := f'] : \Phi$	by ind. hyp.
4.	$(\Pi y\colon \Phi.\, \alpha)[x := f'] = \Pi y\colon \Phi.\, \alpha[x := f']$	by definition of substitution
5.	$\Gamma; \Delta \vdash_\Sigma t[x := f'] \bullet f[x := f'] : \alpha[x := f'][y := f]$	by APP-TERM
6.	$t[x := f'] \bullet f[x := f'] = t \bullet f[x := f']$	by definition of substitution
7.	$\alpha[y := f][x := f'] = \alpha[x := f'][y := f]$	by commutativity of substitution
8.	$\Gamma; \Delta \vdash_\Sigma t \bullet f[x := f'] : \alpha[x := f'][y := f]$	by equalities

Case. $$\frac{\Gamma \vdash_\Sigma e_i : \varepsilon \qquad \Gamma, y : \varepsilon; \Delta \vdash_\Sigma t_i : \alpha[y := e_i]}{\Gamma; \Delta \vdash_\Sigma \mathsf{case}_y\ \{e_i \to t_i\}_{i=1}^n : \Pi y\colon \varepsilon.\, \alpha}\ \text{CASE-TERM}$$

1.	$\Gamma, x : \Phi'; \Delta \vdash_\Sigma \mathsf{case}_y\ \{e_i \to t_i\}_{i=1}^n : \Pi y\colon \varepsilon.\, \alpha$	by assumption
2.	$\Gamma, x : \Phi' \vdash_\Sigma e_i : \varepsilon$ $\Gamma, x : \Phi', y : \varepsilon; \Delta \vdash_\Sigma t_i : \alpha[y := e_i]$	by inversion
3.	$\Gamma \vdash_\Sigma e_i[x := f'] : \varepsilon$ $\Gamma, y : \varepsilon; \Delta \vdash_\Sigma t_i[x := f'] : \alpha[y := e_i][x := f']$	by ind. hyp.
4.	$\alpha[y := e_i][x := f'] = \alpha[x := f'][y := e_i]$	By commutativity of substitution
5.	$e_i[x := f'] = e_i$	by definition of substitution
6.	$\Gamma; \Delta \vdash_\Sigma \mathsf{case}_y\ \{e_i \to t_i[x := f']\}_{i=1}^n :$ $\Pi y\colon \varepsilon.\, \alpha[x := f']$	by CASE-TERM
7.	$\mathsf{case}_y\ \{e_i \to t_i[x := f']\}_{i=1}^n = \mathsf{case}_y\ \{e_i \to t_i\}_{i=1}^n[x := f']$	by definition of substitution
8.	$\Gamma; \Delta \vdash_\Sigma \mathsf{case}_y\ \{e_i \to t_i\}_{i=1}^n[x := f'] :$ $\Pi y\colon \varepsilon.\, \alpha[x := f']$	by equalities

The rest of the cases are similar to the above cases. □

Theorem 5.1 (Subject Reduction)**.** *Let Γ be a feature structure context and Δ be a linear context. The following properties hold for any reduction $t \rhd_{1\beta} t'$:*

1. *If* $\Gamma \vdash_\Sigma f : \Phi$ *and* $f \rhd_{1\beta} f'$, *then* $\Gamma \vdash_\Sigma f' : \Phi$.
2. *If* $\Gamma \vdash_\Sigma \alpha :: k$ *and* $\alpha \rhd_{1\beta} \alpha'$, *then* $\Gamma \vdash_\Sigma \alpha' :: k$.
3. *If* $\Gamma; \Delta \vdash_\Sigma t : \alpha$ *and* $t \rhd_{1\beta} t'$, *then* $\Gamma; \Delta \vdash_\Sigma t' : \alpha$.

Proof. By simultaneous induction on the derivation of the three judgment forms. We present the most complex case: term case analysis β-contraction.
Case. $t = (\mathsf{case}_x \{e_i \to t_i\}_{i=1}^n) \bullet e_j$ and $t' = t_j[x := e_j]$ for some $j \in [1, n]$

1.	$\Gamma; \Delta \vdash_\Sigma (\mathsf{case}_x \{e_i \to t_i\}_{i=1}^n) \bullet e_j : \alpha$	by assumption
2.	$\Gamma; \Delta \vdash_\Sigma \mathsf{case}_x \{e_i \to t_i\}_{i=1}^n : \Pi x\colon \varepsilon.\, \beta$ $\Gamma \vdash_\Sigma e_j : \varepsilon$ $\alpha = \beta[x := e_j]$	by inversion on APP-TERM
3.	$\Gamma \vdash_\Sigma e_i : \varepsilon$ for all $i \in [1, n]$ $\Gamma, x : \varepsilon; \Delta \vdash_\Sigma t_i : \beta[x := e_i]$ for all $i \in [1, n]$	by inversion on CASE-TERM
4.	$\Gamma \vdash_\Sigma e_j : \varepsilon$	from second premise above
5.	$\Gamma, x : \varepsilon; \Delta \vdash_\Sigma t_j : \beta[x := e_j]$	from case-term inversion for index j
6.	$\Gamma; \Delta \vdash_\Sigma t_j[x := e_j] : \beta[x := e_j][x := e_j]$	by lemma 5.3
7.	$\Gamma; \Delta \vdash_\Sigma t_j[x := e_j] : \beta[x := e_j]$	since substitution is idempotent
8.	$\Gamma; \Delta \vdash_\Sigma t_j[x := e_j] : \alpha$	by equality

□

Canonical Forms

Having established subject reduction, we now turn to formally characterizing the structure of expressions in β-normal form within $\lambda^{\multimap\mathcal{F}\Pi}$. This characterization employs two mutually recursive predicates for neutral normal forms (NFN) and normal forms (NF), defined across all three syntactic categories of feature structures, types, and terms.

Definition 5.13 (Feature Structure β-normal Forms)**.** A feature structure term f is in β-normal form, written $\mathsf{NF}(f)$, if it can be derived from the following rules:

$$\frac{}{\mathsf{NFN}(e)}\ \text{ENUMERAL-NEUTRAL} \qquad \frac{}{\mathsf{NFN}(x)}\ \text{VAR-NEUTRAL}$$

$$\frac{\mathsf{NFN}(f)}{\mathsf{NFN}(f.l)}\ \text{SELECTION-NEUTRAL} \qquad \frac{\forall i \in [1,n].\, \mathsf{NF}(f_i)}{\mathsf{NF}([l_i = f_i]_{i=1}^{n})}\ \text{RECORD-NF}$$

$$\frac{\mathsf{NFN}(f)}{\mathsf{NF}(f)}\ \text{NEUTRAL-NF}$$

Definition 5.14 (Type β-normal Forms)**.** A type α is in β-normal form, written $\mathsf{NF}(\alpha)$, if it can be derived from the following rules:

$$\frac{}{\mathsf{NFN}(a)}\ \text{ATOMIC-TYPE-NEUTRAL}$$

$$\frac{\mathsf{NFN}(\alpha) \qquad \mathsf{NF}(f)}{\mathsf{NFN}(\alpha\, f)}\ \text{APP-TYPE-NEUTRAL} \qquad \frac{\mathsf{NFN}(\alpha)}{\mathsf{NF}(\alpha)}\ \text{NEUTRAL-TYPE-NF}$$

$$\frac{\mathsf{NF}(\alpha) \qquad \mathsf{NF}(\beta)}{\mathsf{NF}(\alpha \multimap \beta)}\ \text{LFUNC-TYPE-NF} \qquad \frac{\mathsf{NF}(\alpha)}{\mathsf{NF}(\Lambda x.\, \alpha)}\ \text{ABSTR-TYPE-NF}$$

$$\frac{\forall i \in [1,n].\, \mathsf{NF}(\alpha_i)}{\mathsf{NF}(\mathsf{Case}_x\ \{e_i \to \alpha_i\}_{i=1}^{n})}\ \text{CASE-TYPE-NF} \qquad \frac{\mathsf{NF}(\alpha)}{\mathsf{NF}(\Pi x\colon \Phi.\, \alpha)}\ \text{PI-TYPE-NF}$$

Definition 5.15 (Terms β-normal Forms)**.** A term t is in β-normal form, written $\mathsf{NF}(t)$, if it can be derived from the following rules:

$$\frac{}{\mathsf{NFN}(c)}\ \textsc{const-neutral} \qquad \frac{}{\mathsf{NFN}(x)}\ \textsc{var-neutral} \qquad \frac{\mathsf{NFN}(t) \qquad \mathsf{NF}(u)}{\mathsf{NFN}(t\,u)}\ \textsc{lapp-neutral}$$

$$\frac{\mathsf{NFN}(t) \qquad \mathsf{NF}(f)}{\mathsf{NFN}(t \bullet f)}\ \textsc{app-neutral} \qquad \frac{\mathsf{NFN}(t)}{\mathsf{NF}(t)}\ \textsc{neutral-nf} \qquad \frac{\mathsf{NF}(t)}{\mathsf{NF}(\lambda^{\circ} x.\, t)}\ \textsc{labstr-nf}$$

$$\frac{\mathsf{NF}(t)}{\mathsf{NF}(\lambda x.\, t)}\ \textsc{abstr-nf} \qquad \frac{\forall i \in [1, n].\, \mathsf{NF}(t_i)}{\mathsf{NF}(\mathsf{case}_x\ \{e_i \to t_i\}_{i=1}^{n})}\ \textsc{case-nf}$$

We characterize closed canonical feature structures and closed canonical expressions, which will be required in the next chapter. This characterization is not exhaustive for all expressions in the calculus, but focuses on those with types of kind Type.

Lemma 5.4 (Canonical Forms for Closed Feature Structures)**.** *Let f be a closed feature structure term such that $\mathsf{NF}(f)$ and $\vdash_\Sigma f : \Phi$. Then f has one of the following forms:*

1. *$f = e$ for some enumeral e;*
2. *$f = [l_i = f_i]_{i=1}^{n}$ where each $\mathsf{NF}(f_i)$ and $\vdash_\Sigma f_i : \Phi_i$.*

Proof. Since f is closed and in normal form, it contains no variables or redexes and selections of form $x.l$. By Definition 5.13, f is either an enumeral or a record of normal forms. □

Lemma 5.5 (Canonical Forms for Closed Types of Kind Type)**.** *Let α be a closed type such that $\mathsf{NF}(\alpha)$ and $\vdash_\Sigma \alpha :: \mathsf{Type}$. Then α has one of the following forms:*

1. *$\alpha = a$ for some atomic type a where $\vdash_\Sigma a :: \mathsf{Type}$;*
2. *$\alpha = \beta \multimap \gamma$ where $\mathsf{NF}(\beta)$ and $\mathsf{NF}(\gamma)$;*
3. *$\alpha = a\, f_1 \ldots f_n$ for some atomic type a and feature structure arguments $f_1, \ldots, f_n$ where each f_i is closed and $\mathsf{NF}(f_i)$.*

Proof. By Definition 5.14, a type α in β-normal form is either neutral or has one of the following forms: $\beta \multimap \gamma$, $\Lambda x.\, \beta$, $\mathsf{Case}_x\ \{e_i \to \alpha_i\}_{i=1}^{n}$, or $\Pi x\colon \Phi.\, \beta$.

We first eliminate the impossible cases given that $\vdash_\Sigma \alpha :: \mathsf{Type}$:

- $\Lambda x.\, \beta$ has kind $\Phi \Rrightarrow k$ by ABSTR-TYPE, not Type.
- $\mathsf{Case}_x\ \{e_i \to \alpha_i\}_{i=1}^n$ has kind $\varepsilon \Rrightarrow k$ by CASE-TYPE, not Type.
- $\Pi x\colon \Phi.\, \beta$ has kind DType by PI-TYPE, not Type.

Therefore, α is either neutral or has the form $\beta \multimap \gamma$.

If $\alpha = \beta \multimap \gamma$, then by LFUNC-TYPE, both β and γ have kind Type, and since α is in normal form, both β and γ are in normal form. This gives us case (2).

If α is neutral, then by Definition 5.14, neutral types are either atomic types or type applications $\beta\, f$ where β is neutral and f is in normal form.

If $\alpha = a$ for some atomic type a, this gives us case (1).

Otherwise, $\alpha = \beta\, f$ where β is neutral and f is in normal form. Since α is closed, f must be a closed feature structure term in normal form.

Since β is neutral, by Definition 5.14, β is either an atomic type or another type application of the same form. In both cases, α has the form $a\, f_1\, \ldots\, f_n$ for some atomic type a at the head and closed feature structure arguments $f_1, \ldots, f_n$. This gives us case (3).

Therefore, every closed type α in normal form with $\vdash_\Sigma \alpha :: \mathsf{Type}$ falls into one of the three stated cases. □

Lemma 5.6 (Canonical Forms of Closed Terms Typed by Proper Types)**.** *Let t be a closed term such that $\mathsf{NF}(t)$ and $\vdash_\Sigma t : \alpha$ where $\vdash_\Sigma \alpha :: \mathsf{Type}$. Then t has one of the following forms:*

1. *$t = c$ for some constant c where $\vdash_\Sigma \tau(c) :: \mathsf{Type}$;*
2. *$t = \lambda^\circ x.\, u$ for some linear abstraction where $\mathsf{NF}(u)$;*
3. *$t = u_1\, u_2$ for some linear application where $\mathsf{NFN}(u_1)$ and $\mathsf{NF}(u_2)$;*
4. *$t = c \bullet f_1 \bullet \cdots \bullet f_n$ for some constant c and feature structure arguments $f_1, \ldots, f_n$ where each $\mathsf{NF}(f_i)$.*

Proof. By Definition 5.15, t is either neutral or has the form $\lambda^\circ x.\, u$, $\lambda x.\, u$, or case analysis.

Feature abstraction $\lambda x.\, u$ is ruled out since by ABSTR-TERM it has type $\Pi x\colon \Phi.\, \beta$ with kind DType, contradicting $\alpha :: \mathsf{Type}$.

Case analysis $\mathsf{case}_x\ \{e_i \to t_i\}_{i=1}^n$ is ruled out since by CASE-TERM it has type $\Pi x\colon \varepsilon.\, \beta$ with kind DType, contradicting $\alpha :: \mathsf{Type}$.

If $t = \lambda^\circ x.\, u$, then $\mathsf{NF}(u)$ by Definition 5.15, giving us case 2.

If t is neutral, it can be a constant, variable, linear application, or feature structure application. Variables are impossible for closed terms.

If $t = c$ for some constant c, then by CONST-TERM, c has type $\tau(c)$ where $\vdash_\Sigma \tau(c) :: \mathsf{DType}$. Since $\vdash_\Sigma \alpha :: \mathsf{Type}$ and $t : \alpha$, we must have $\vdash_\Sigma \tau(c) :: \mathsf{Type}$ by PROMOTION-TYPE. This gives us case 1.

If $t = u_1\, u_2$ where $\mathsf{NFN}(u_1)$ and $\mathsf{NF}(u_2)$, this gives us case 3.

If $t = u \bullet f$ where $\mathsf{NFN}(u)$ and $\mathsf{NF}(f)$, then by APP-TERM, u has type $\Pi x\colon \Phi.\, \beta$ with kind DType. Since u is neutral and closed, u must be either a constant with dependent product type or a chain of applications starting with such a constant. This gives the form $t = c \bullet f_1 \bullet \cdots \bullet f_n$ for case 4. □

5.3.3 Strong Normalization

Strong normalization and the Church-Rosser property are essential for $\lambda^{-\circ\mathcal{F}\Pi}$ to serve as a foundation for extending ACGs, as they ensure termination of all computations and deterministic reduction in the applicative paradigm.

The Church-Rosser property follows by the standard parallel reduction technique, as all our new constructs (feature structure terms, case analysis, dependent products) preserve the diamond property under parallel reduction.

Intuitively, $\lambda^{-\circ\mathcal{F}\Pi}$ enjoys strong normalization because the constructs we introduce (feature structure terms, dependent products, case analysis, and type abstractions) preserve termination guarantees. Feature structures represent finite data structures with deterministic access operations, case analysis performs exhaustive branching over finite enumerations, and dependent products quantify exclusively over these bounded structures. Crucially, none of these extensions introduce self-referential mechanisms or unbounded recursion that could lead to infinite reduction sequences.

A complete formal proof of strong normalization for $\lambda^{-\circ\mathcal{F}\Pi}$ would require substantial development due to the complexity of our three-tiered system with mutually interdependent judgments for feature structures, types, and terms. Instead, we sketch how this fundamental result can be established by adapting existing proof techniques.

The strong normalization of $\lambda^{-\circ\mathcal{F}\Pi}$ can be proven by adapting the dual transformation strategy developed for the Edinburgh Logical Framework (Harper, Honsell, and Plotkin 1993). Harper's key insight was to circumvent the direct analysis of dependent type systems by translating them into the simply-typed λ-calculus, whose normalization properties are well-established.

Harper's translation systematically eliminates dependencies by collapsing the three-tiered hierarchy of terms, types, and kinds into the standard two-level system of terms and types. The translation employs two key functions: an erasure function $|\cdot|$ that maps both object-level and type-level expressions to Curry-style λ-terms, and a function $\tau(\cdot)$ that translates types and kinds to simple types. The fundamental insight is that both primary judgments of dependent type theory can be uniformly represented as typing judgments in the target simply-typed system. This uniform treatment enables a powerful reducibility argument: since the translation preserves both typability and reduction behavior, any infinite reduction sequence in the source system would induce an infinite sequence in the target, contradicting the strong normalization of simply-typed λ-calculus.

Pompigne (2013) successfully adapted Harper's technique to a linear λ-calculus enriched with variants, records, and dependent products. Pompigne's work demonstrates that λ-calculus extended with products and sums provides sufficient expressive power as a target system for such enriched calculi.

Chapter 6

Affix Abstract Categorial Grammars

Equipped with the calculus $\lambda^{-\circ\mathcal{F}\Pi}$, we now extend abstract categorial grammars by integrating this modelling of feature structures, yielding what we call *Affix Abstract Categorial Grammars* (AACG). We then establish their equivalence to core ACG through a series of transformations, examine practical parsing considerations, and conclude with an analysis of feature structure omission.

6.1 Grammars

6.1.1 Definitions

Type abstractions, type case analysis and type β-redexes are fundamental to lexicon interpretation. However, allowing these constructs in constant typing would introduce excessive expressivity in the grammar system. Therefore, we impose additional constraints on the raw higher-order signatures to maintain theoretical control.

Definition 6.1 (Affix Higher-Order Signature)**.** An *affix higher-order signature* is a raw higher-order signature $\Sigma = \langle A, \mathfrak{F}, \kappa, C, \tau \rangle$ that satisfies the following conditions:

1. For all constants $c \in C$, the judgment $\vdash_\Sigma \tau(c) :: \mathsf{DType}$ holds;
2. For all constants $c \in C$, $\mathsf{NF}(\tau(c))$;
3. For all constants $c \in C$, the type $\tau(c)$ contains no occurrences of type case analysis or type abstraction.

We assume throughout the remainder of this chapter that Σ refers to an affix higher-order signature.

It remains to define the lexicon for AACGs and specify how feature structures are interpreted by the lexicon. The calculus already provides mechanisms to erase feature structures, transfer them to the object level, or create interpretations that vary based on their value. A key principle of this design is the separation of concerns: the calculus is responsible for the complex manipulation of feature structures, which simplifies the lexicon's role to that of a direct bridge between signatures via an identity mapping on feature structures.

This design choice necessitates that the feature structure signature of the abstract affix higher-order signature be included in that of the object affix higher-order signature. Consequently, all enumerals, enumerations, and labels defined in the abstract signature must also be present in the object signature.

Definition 6.2 (Feature Structure Signature Inclusion)**.** Let $\mathfrak{F}_1 = \langle E_1, \mathcal{E}_1, L_1 \rangle$ and $\mathfrak{F}_2 = \langle E_2, \mathcal{E}_2, L_2 \rangle$ be two feature structure signatures. We say that $\mathfrak{F}_1$ is included in $\mathfrak{F}_2$, denoted $\mathfrak{F}_1 \sqsubseteq_{\mathcal{F}} \mathfrak{F}_2$, if the following conditions hold:

1. $E_1 \subseteq E_2$;
2. $\mathcal{E}_1 \subseteq \mathcal{E}_2$;
3. $L_1 \subseteq L_2$.

Definition 6.3 (Type and Term Homomorphisms)**.** Let $\Sigma_1 = \langle A_1, \mathfrak{F}_1, \kappa_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, \mathfrak{F}_2, \kappa_2, C_2, \tau_2 \rangle$ be two affix higher-order signatures where $\mathfrak{F}_1 \sqsubseteq_{\mathcal{F}} \mathfrak{F}_2$. Let $F : A_1 \to \mathcal{T}(A_2, \mathfrak{F}_2)$ and $G : C_1 \to T(\Sigma_2)$ be functions. The unique homomorphic extensions are defined as follows:

1. The extension for implicative types, $\hat{F} : \mathcal{T}(A_1, \mathfrak{F}_1) \to \mathcal{T}(A_2, \mathfrak{F}_2)$, is defined inductively:

$$\begin{aligned}
\hat{F}(a) &= F(a) \text{ for } a \in A_1 \\
\hat{F}(\alpha \multimap \beta) &= \hat{F}(\alpha) \multimap \hat{F}(\beta) \\
\hat{F}(\Lambda x.\, \alpha) &= \Lambda x.\, \hat{F}(\alpha) \\
\hat{F}(\alpha\, f) &= \hat{F}(\alpha)\, f \\
\hat{F}(\mathsf{Case}_x\ \{e_i \to \alpha_i\}_{i=1}^n) &= \mathsf{Case}_x\ \{e_i \to \hat{F}(\alpha_i)\}_{i=1}^n
\end{aligned}$$

2. The extension for affix types, $\widehat{F_{\mathcal{D}}} : \mathcal{D}(A_1, \mathfrak{F}_1) \to \mathcal{D}(A_2, \mathfrak{F}_2)$, is defined based on $\hat{F}$:

$$\begin{aligned} \widehat{F_{\mathcal{D}}}(\alpha) &= \hat{F}(\alpha) \quad \text{if } \alpha \in \mathcal{T}(A_1, \mathfrak{F}_1) \\ \widehat{F_{\mathcal{D}}}(\Pi x\colon \Phi.\, \alpha) &= \Pi x\colon \Phi.\, \widehat{F_{\mathcal{D}}}(\alpha) \end{aligned}$$

3. The extension for terms, $\hat{G} : T(\Sigma_1) \to T(\Sigma_2)$, is defined inductively:

$$\begin{aligned} \hat{G}(c) &= G(c) \text{ for } c \in C_1 \\ \hat{G}(x) &= x \text{ for } x \in \mathcal{X} \\ \hat{G}(\lambda^{\circ} x.\, t) &= \lambda^{\circ} x.\, \hat{G}(t) \\ \hat{G}(t\, u) &= \hat{G}(t)\, \hat{G}(u) \\ \hat{G}(\lambda x.\, t) &= \lambda x.\, \hat{G}(t) \\ \hat{G}(t \bullet f) &= \hat{G}(t) \bullet f \\ \hat{G}(\mathsf{case}_x\ \{e_i \to t_i\}_{i=1}^n) &= \mathsf{case}_x\ \{e_i \to \hat{G}(t_i)\}_{i=1}^n \end{aligned}$$

Definition 6.4 (Affix Lexicon)**.** Let $\Sigma_1 = \langle A_1, \mathfrak{F}_1, \kappa_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, \mathfrak{F}_2, \kappa_2, C_2, \tau_2 \rangle$ be two affix signatures where $\mathfrak{F}_1 \sqsubseteq_{\mathcal{F}} \mathfrak{F}_2$. An *affix lexicon* $\mathcal{L} : \Sigma_1 \to \Sigma_2$ is a pair $\langle F, G \rangle$ where:

1. $F : A_1 \to \mathcal{T}(A_2, \mathfrak{F}_2)$ is a function interpreting types families from the abstract signature as types defined over the object signature;
2. $G : C_1 \to T(\Sigma_2)$ is a function interpreting abstract constants as terms defined over the object signature;
3. For each $a \in A_1$, the following kinding judgment must be derivable:

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

4. For each $c \in C_1$, the following typing judgment must be derivable:

$$\vdash_{\Sigma_2} G(c) : \widehat{F_{\mathcal{D}}}(\tau_1(c)).$$

Definition 6.5 (AACG)**.** Let $\Sigma_1 = \langle A_1, \mathfrak{F}_1, \kappa_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, \mathfrak{F}_2, \kappa_2, C_2, \tau_2 \rangle$ be two affix higher-order signatures. An *AACG*, $\mathcal{G}$ is a quadruplet $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ where:

1. Σ_1 is the *abstract affix higher-order signature*;

2. Σ_2 is the *object affix higher-order signature*;
3. $\mathcal{L} : \Sigma_1 \to \Sigma_2$ is an affix lexicon from Σ_1 to Σ_2;
4. $\mathbf{S} \in \mathcal{T}(A_1, \mathfrak{F}_1)$ is the *distinguished type* that satisfies the following conditions:
   - $\vdash_{\Sigma_1} \mathbf{S} :: \mathsf{Type}$;
   - $\mathsf{NF}(\mathbf{S})$.

The languages generated by an AACG are defined analogously to those of core ACGs.

Definition 6.6 (AACG Languages)**.** Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be an AACG. The *abstract language* $\mathcal{A}(\mathcal{G})$ and the *object language* $\mathcal{O}(\mathcal{G})$ of $\mathcal{G}$ are defined as follows:

$$\mathcal{A}(\mathcal{G}) = \{t \in T(\Sigma_1) \mid \vdash_{\Sigma_1} t : \mathbf{S}\}$$
$$\mathcal{O}(\mathcal{G}) = \{t \in T(\Sigma_2) \mid \exists u \in \mathcal{A}(\mathcal{G}).t = \mathcal{L}(u)\}$$

6.1.2 Defining Equivalence Between Grammars

To demonstrate the equivalence between AACGs and core ACGs, we must first establish a formal definition of what it means for two grammars to be equivalent. Two AACGs are considered equivalent if they describe the same set of linguistic objects in the same way. This requires a one-to-one correspondence between their abstract and object languages, a relationship that must be consistently preserved by the grammars' lexicons.

Definition 6.7 (Affix ACG Language Isomorphism)**.** Let $\mathcal{G}_1 = \langle \Sigma_1, \Sigma_2, \mathcal{L}_1, \mathbf{S}_1 \rangle$ and $\mathcal{G}_2 = \langle \Sigma_1', \Sigma_2', \mathcal{L}_2, \mathbf{S}_2 \rangle$ be two AACGs. A *language isomorphism* Ψ from $\mathcal{G}_1$ to $\mathcal{G}_2$ is a pair of bijective functions $\Psi = (\psi_{\mathcal{A}}, \psi_{\mathcal{O}})$ where:

1. $\psi_{\mathcal{A}} : \mathcal{A}(\mathcal{G}_1) \to \mathcal{A}(\mathcal{G}_2)$ is a bijection between the abstract languages;
2. $\psi_{\mathcal{O}} : \mathcal{O}(\mathcal{G}_1) \to \mathcal{O}(\mathcal{G}_2)$ is a bijection between the object languages;
3. for every abstract term $t \in \mathcal{A}(\mathcal{G}_1) : \psi_{\mathcal{O}}(\mathcal{L}_1(t)) = \mathcal{L}_2(\psi_{\mathcal{A}}(t))$.

A pair of bijections alone would be insufficient to guarantee a meaningful structural correspondence. The third requirement ensures that interpreting an abstract term from $\mathcal{G}_1$ and then mapping the result to $\mathcal{G}_2$'s object language yields the same outcome as first mapping the abstract term to $\mathcal{G}_2$'s language and then interpreting it. This can be visualized in Figure 6.1.

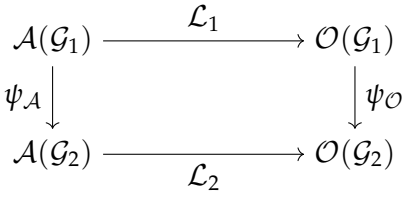


FIGURE 6.1: Language isomorphism commutativity diagram

Definition 6.8 (AACG Equivalence)**.** Two AACGs, $\mathcal{G}_1$ and $\mathcal{G}_2$, are *equivalent* or *isomorphic*, written $\mathcal{G}_1 \cong \mathcal{G}_2$, if there exists a language isomorphism Ψ from $\mathcal{G}_1$ to $\mathcal{G}_2$.

6.1.3 A Conservative Extension

Having defined equivalence, we now show that our extension is conservative. The following theorem establishes that any core ACG can be viewed as an AACG with empty feature structures, proving that the new formalism subsumes the original.

Theorem 6.1 (Core ACG Embedding)**.** *Any core ACG* $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ *can be embedded into an equivalent Affix ACG ,* $\mathcal{G}' = \langle \Sigma_1', \Sigma_2', \mathcal{L}, \mathbf{S} \rangle$*.*

Proof. A core ACG can be viewed as a degenerate case of an AACG where the feature structure signatures are empty. To construct the equivalent AACG $\mathcal{G}'$, we embed $\mathcal{G}$ by setting the feature structure signatures to $\mathfrak{F} = \langle \emptyset, \emptyset, \emptyset \rangle$ in both abstract and object signatures. The lexicon satisfies the kinding requirements trivially, as all atomic types in a core ACG are implicitly proper types of kind Type.

Since the feature structure signatures are empty, no feature structure constructs are available, and the resulting languages of terms and types are identical to those of the original core ACG. Therefore, $\mathcal{A}(\mathcal{G}) = \mathcal{A}(\mathcal{G}')$ and $\mathcal{O}(\mathcal{G}) = \mathcal{O}(\mathcal{G}')$. □

6.2 Expressivity of AACGs

We now prove that AACGs are not more expressive than core ACGs. More precisely, we show that any AACG can be transformed into a core ACG that generates equivalent languages. Figure 6.2 illustrates the hierarchical relationships between the constructors we've introduced, culminating in $\lambda^{-\circ\mathcal{F}\Pi}$. This hierarchy suggests a natural decomposition of the equivalence proof into three key steps:

1. First, we must show that any AACG with dependent products can be transformed into an equivalent AACG using only type families (reduction from $\lambda^{-\circ\mathcal{F}\Pi}$ to $\lambda^{-\circ\mathcal{F}}$).

2. Next, we need to demonstrate that any AACG with record types can be transformed into an equivalent AACG without records (reduction from $\lambda^{-\circ\mathcal{F}}$ to $\lambda^{-\circ\mathcal{E}}$).

3. Finally, we must prove that any AACG using only type families can be transformed into an equivalent core ACG (reduction from $\lambda^{-\circ\mathcal{E}}$ to $\lambda^{-\circ}$).

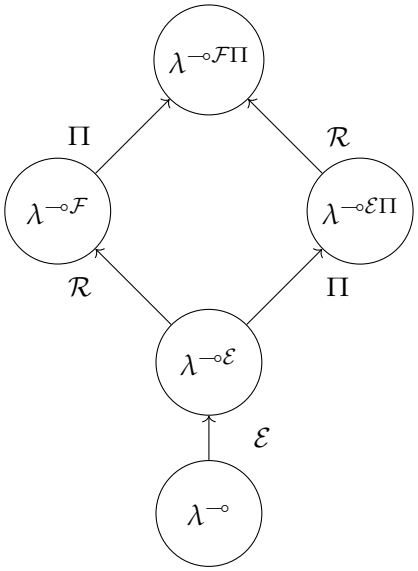


FIGURE 6.2: Hierarchical containment of $\lambda^{-\circ\mathcal{F}\Pi}$

For each reduction step, the method is to construct a transformation that maps an AACG from the source calculus to an equivalent one in the target calculus. This transformation is designed to produce a new grammar $\mathcal{G}'$, by eliminating the targeted constructor from the signatures and lexicon of the original grammar, $\mathcal{G}$. The proof of equivalence ($\mathcal{G} \cong \mathcal{G}'$) is constructive: the transformation is designed specifically to yield a language isomorphism Ψ and to guarantee that the lexicon commutativity property is satisfied.

The order of these transformations is chosen to minimize complexity. While an alternative sequence is possible (e.g., eliminating records before dependent products), this approach is more direct. Deferring the elimination of dependent products would require

the record transformation to handle a richer set of binders thereby complicating the proof unnecessarily.

6.2.1 Π-Elimination

The first step in reducing AACGs to core ACGs is the elimination of dependent products. The key insight of this elimination is that since feature structures are finite, we can expand constants typed by dependent products into a finite set of non-dependent constants.

We introduce some notation about the reduction of types and terms. For types, we write $\alpha \Downarrow \alpha'$ to denote that α reduces to α' with α' in normal form. Similarly, for terms we write $t \Downarrow u$ to signify that t reduces to u with u in normal form. Moreover, we denote the normal form of t by $t \Downarrow$ and the normal form of α by $\alpha \Downarrow$.

Definition 6.9 (Π-Elimination for Affix Higher-Order Signatures)**.** Let $\Sigma = \langle A, \mathfrak{F}, \kappa, C, \tau \rangle$ be an affix higher-order signature. The Π-elimination of Σ, written $\Pi^{\ominus}[\![\Sigma]\!] = \langle A, \mathfrak{F}, \kappa, \Pi^{\ominus}[\![C]\!], \Pi^{\ominus}[\![\tau]\!] \rangle$, is defined as follows: $\Pi^{\ominus}[\![C]\!]$ is the union of two sets:

- The set of constants $c \in C$ such that $\vdash_{\Sigma} \tau(c) :: \mathsf{Type}$;
- The set of constants $c_{f_1,\ldots,f_n}$ for each $c \in C$ where $\tau(c) = \Pi x_1 : \Phi_1 . \Pi \ldots . \Pi x_n : \Phi_n . \alpha$ and $(f_1, \ldots, f_n) \in \Phi_1 \times \cdots \times \Phi_n$ is a sequence of closed feature structure arguments in β-normal form.

For any $c' \in \Pi^{\ominus}[\![C]\!]$, $\Pi^{\ominus}[\![\tau]\!]$ is defined by:

- If $c' = c$ where $c \in C$ and $\vdash_{\Sigma} \tau(c) :: \mathsf{Type}$, then $\Pi^{\ominus}[\![\tau]\!](c') = \tau(c)$;
- If $c' = c_{f_1,\ldots,f_n}$ where $c \in C$ and $\tau(c) = \Pi x_1 : \Phi_1 . \Pi \ldots . \Pi x_n : \Phi_n . \alpha$, then $\Pi^{\ominus}[\![\tau]\!](c') = \alpha[x_1 := f_1, \ldots, x_n := f_n] \Downarrow$.

Definition 6.10 (Π-Elimination for Terms)**.** The Π-less translation of terms, written $\Pi^{\ominus}[\![t]\!]$, is defined only for terms t such that $\mathsf{NF}(t)$, $\vdash_{\Sigma} t : \alpha$ and $\vdash_{\Sigma} \alpha :: \mathsf{Type}$, inductively by:

$$
\begin{aligned}
\Pi^{\ominus}[\![x]\!] &= x \\
\Pi^{\ominus}[\![c]\!] &= c \\
\Pi^{\ominus}[\![t_1\, t_2]\!] &= \Pi^{\ominus}[\![t_1]\!]\, \Pi^{\ominus}[\![t_2]\!] \\
\Pi^{\ominus}[\![c \bullet f_1 \bullet \ldots \bullet f_n]\!] &= c_{f_1,\ldots,f_n} \\
\Pi^{\ominus}[\![\lambda^{\circ} x . t]\!] &= \lambda^{\circ} x . \Pi^{\ominus}[\![t]\!]
\end{aligned}
$$

The Π-elimination only affects the term level; types remain unchanged.

Lemma 6.1 (Typing Preservation under Π-Elimination)**.** *Let Σ be an affix higher-order signature, and let $\Pi^{\ominus}[\![\Sigma]\!]$ be its Π-elimination. Let α be a type such that $\mathsf{NF}(\alpha)$ and $\vdash_{\Sigma} \alpha :: \mathsf{Type}$. Then for any term t such that $\mathsf{NF}(t)$, if $\vdash_{\Sigma} t : \alpha$, then $\vdash_{\Pi^{\ominus}[\![\Sigma]\!]} \Pi^{\ominus}[\![t]\!] : \alpha$.*

Proof. By induction on the typing derivation $\vdash_{\Sigma} t : \alpha$. The critical case is $t = c \bullet f_1 \bullet \cdots \bullet f_n$ where $\tau(c) = \Pi x_1 : \Phi_1. \Pi \ldots . \Pi x_n : \Phi_n. \beta$ and $\alpha = \beta[x_1 := f_1, \ldots, x_n := f_n]$. By construction, $\Pi^{\ominus}[\![\tau]\!](c_{f_1,\ldots,f_n}) = \beta[x_1 := f_1, \ldots, x_n := f_n] \Downarrow$. Since $\vdash_{\Pi^{\ominus}[\![\Sigma]\!]} \alpha \equiv \alpha \Downarrow$ by definitional equality, we have $\vdash_{\Pi^{\ominus}[\![\Sigma]\!]} c_{f_1,\ldots,f_n} : \alpha$ by the TYPE-EQ rule. □

The Π-elimination for lexicons mirrors the signature transformation by creating specialized interpretations for each expanded constant. When a constant c with dependent product type is eliminated into multiple specialized constants $c_{f_1,\ldots,f_n}$, each specialized constant receives an interpretation computed by applying the original lexicon to the fully instantiated term $c \bullet f_1 \bullet \ldots \bullet f_n$ and then applying the Π-elimination transformation to the normalized result.

Definition 6.11 (Lexicon Π-Elimination)**.** Let $\mathcal{L} : \Sigma_1 \to \Sigma_2 = \langle F, G \rangle$ be an affix lexicon between two affix higher-order signatures $\Sigma_1 = \langle A_1, \mathfrak{F}_1, \kappa_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, \mathfrak{F}_2, \kappa_2, C_2, \tau_2 \rangle$. Let $\Pi^{\ominus}[\![\Sigma_1]\!]$ and $\Pi^{\ominus}[\![\Sigma_2]\!]$ be their respective Π-eliminations. The Π-elimination of $\mathcal{L}$, written $\Pi^{\ominus}[\![\mathcal{L}]\!] : \Pi^{\ominus}[\![\Sigma_1]\!] \to \Pi^{\ominus}[\![\Sigma_2]\!] = \langle F, \Pi^{\ominus}[\![G]\!] \rangle$, is defined for any $c' \in \Pi^{\ominus}[\![C_1]\!]$ by:

- If $c' = c$ where $c \in C_1$ and $\vdash_{\Sigma_1} \tau_1(c) :: \mathsf{Type}$, then $\Pi^{\ominus}[\![G]\!](c') = \Pi^{\ominus}[\![\mathcal{L}(c) \Downarrow]\!]$;
- If $c' = c_{f_1,\ldots,f_n}$ where $c \in C_1$ and $\tau_1(c) = \Pi x_1 : \Phi_1. \Pi \ldots . \Pi x_n : \Phi_n. \alpha$, then $\Pi^{\ominus}[\![G]\!](c') = \Pi^{\ominus}[\![\mathcal{L}(c \bullet f_1 \bullet \ldots \bullet f_n) \Downarrow]\!]$.

Definition 6.12 (AACG Π-Elimination)**.** Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be an Affix ACG . The Π-elimination of $\mathcal{G}$, written $\Pi^{\ominus}[\![\mathcal{G}]\!]$, is defined as:

$$\Pi^{\ominus}[\![\mathcal{G}]\!] = \langle \Pi^{\ominus}[\![\Sigma_1]\!], \Pi^{\ominus}[\![\Sigma_2]\!], \Pi^{\ominus}[\![\mathcal{L}]\!], \mathbf{S} \rangle.$$

The Π-elimination transformation requires enumerating all possible feature structure instantiations to create specialized constants. This necessitates working with terms in β-normal form to ensure feature values are concrete and fully evaluated, which is why the following results restrict attention to the normal form sublanguages $\mathcal{A}(\mathcal{G})^{\mathsf{NF}}$ and $\mathcal{O}(\mathcal{G})^{\mathsf{NF}}$, consisting respectively of terms in β-normal form from $\mathcal{A}(\mathcal{G})$ and $\mathcal{O}(\mathcal{G})$.

Lemma 6.2 (Bijectivity of Π-Elimination)**.** *Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be an AACG and $\Pi^{\ominus}[\![\mathcal{G}]\!]$ its Π-elimination. The transformation $t \mapsto \Pi^{\ominus}[\![t]\!]$ establishes bijections between $\mathcal{A}(\mathcal{G})^{NF}$ and $\mathcal{A}(\Pi^{\ominus}[\![\mathcal{G}]\!])$, and between $\mathcal{O}(\mathcal{G})^{NF}$ and $\mathcal{O}(\Pi^{\ominus}[\![\mathcal{G}]\!])$.*

Proof. Each specialized constant $c_{f_1,\ldots,f_n}$ uniquely encodes the original constant c and feature arguments $(f_1, \ldots, f_n)$. Since terms in $\mathcal{A}(\mathcal{G})^{\mathsf{NF}}$ are already in β-normal form, the transformation $t \mapsto \Pi^{\ominus}[\![t]\!]$ is well-defined and invertible on the normal form sublanguages. □

Lemma 6.3 (Lexicon Commutativity for Π-Elimination)**.** *Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be an AACG and $\Pi^{\ominus}[\![\mathcal{G}]\!]$ its Π-elimination. For any term $t \in \mathcal{A}(\mathcal{G})^{NF}$, the following holds:*

$$\Pi^{\ominus}[\![\mathcal{L}(t) \Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![t]\!])$$

Proof. We proceed by structural induction on $t \in \mathcal{A}(\mathcal{G})^{\mathsf{NF}}$. By Lemma 5.6, since t is closed and in normal form with $\vdash_{\Sigma} t : \mathbf{S}$ where $\vdash_{\Sigma} \mathbf{S} :: \mathsf{Type}$, we have the following cases:

Case. $t = c$ for some constant c with $\vdash_{\Sigma} \tau(c) :: \mathsf{Type}$.

1.	$\Pi^{\ominus}[\![t]\!] = \Pi^{\ominus}[\![c]\!] = c$	by definition 6.10
2.	$\Pi^{\ominus}[\![\mathcal{L}]\!](c) = \Pi^{\ominus}[\![\mathcal{L}(c) \Downarrow]\!]$	by definition 6.11
3.	$\Pi^{\ominus}[\![\mathcal{L}(t) \Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}(c) \Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}]\!](c) = \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![t]\!])$	by steps 1 and 2

Case. $t = c \bullet f_1 \bullet \ldots \bullet f_n$ for some constant c and feature structure arguments.

1.	$\Pi^{\ominus}[\![t]\!] = c_{f_1,\ldots,f_n}$	by definition 6.10
2.	$\mathcal{L}(t) = \mathcal{L}(c) \bullet f_1 \bullet \ldots \bullet f_n$	by lexicon homomorphism
3.	$\Pi^{\ominus}[\![\mathcal{L}]\!](c_{f_1,\ldots,f_n}) = \Pi^{\ominus}[\![\mathcal{L}(c \bullet f_1 \bullet \ldots \bullet f_n) \Downarrow]\!]$	by definition 6.11
4.	$\Pi^{\ominus}[\![\mathcal{L}(t) \Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}(c) \bullet f_1 \bullet \ldots \bullet f_n \Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}]\!](c_{f_1,\ldots,f_n}) = \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![t]\!])$	by steps 1, 2, and 3

Case. $t = \lambda^{\circ} x.\, u$ for some linear abstraction.

1. $\Pi^{\ominus}[\![t]\!] = \lambda^{\circ}x.\,\Pi^{\ominus}[\![u]\!]$ — by definition 6.10
2. $\mathcal{L}(t) = \lambda^{\circ}x.\,\mathcal{L}(u)$ — by lexicon homomorphism
3. $\Pi^{\ominus}[\![\mathcal{L}(u)\Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![u]\!])$ — by ind. hyp.
4. $\Pi^{\ominus}[\![\mathcal{L}(t)\Downarrow]\!] = \Pi^{\ominus}[\![\lambda^{\circ}x.\,\mathcal{L}(u)\Downarrow]\!]$ — by steps 1, 2, and 3
   $= \lambda^{\circ}x.\,\Pi^{\ominus}[\![\mathcal{L}(u)\Downarrow]\!]$
   $= \lambda^{\circ}x.\,\Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![u]\!])$
   $= \Pi^{\ominus}[\![\mathcal{L}]\!](\lambda^{\circ}x.\,\Pi^{\ominus}[\![u]\!])$
   $= \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![t]\!])$

Case. $t = u_1\,u_2$ for some linear application.

1. $\Pi^{\ominus}[\![t]\!] = \Pi^{\ominus}[\![u_1]\!]\,\Pi^{\ominus}[\![u_2]\!]$ — by definition 6.10
2. $\mathcal{L}(t) = \mathcal{L}(u_1)\,\mathcal{L}(u_2)$ — by lexicon homomorphism
3. $\Pi^{\ominus}[\![\mathcal{L}(u_1)\Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![u_1]\!])$ — by ind. hyp.
4. $\Pi^{\ominus}[\![\mathcal{L}(u_2)\Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![u_2]\!])$ — by ind. hyp.
5. $\Pi^{\ominus}[\![\mathcal{L}(t)\Downarrow]\!] = \Pi^{\ominus}[\![\mathcal{L}(u_1)\Downarrow\ \mathcal{L}(u_2)\Downarrow]\!]$ — by steps 1, 2, 3, and 4
   $= \Pi^{\ominus}[\![\mathcal{L}(u_1)\Downarrow]\!]\,\Pi^{\ominus}[\![\mathcal{L}(u_2)\Downarrow]\!]$
   $= \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![u_1]\!])\,\Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![u_2]\!])$
   $= \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![u_1]\!]\,\Pi^{\ominus}[\![u_2]\!])$
   $= \Pi^{\ominus}[\![\mathcal{L}]\!](\Pi^{\ominus}[\![t]\!])$

□

Theorem 6.2 (Π-less Equivalence)**.** *Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S}\rangle$ be an Affix ACG , and let $\Pi^{\ominus}[\![\mathcal{G}]\!]$ be its Π-elimination. Considering only the β-normal form sublanguages $\mathcal{A}(\mathcal{G})^{NF}$ and $\mathcal{O}(\mathcal{G})^{NF}$, $\mathcal{G}$ and $\Pi^{\ominus}[\![\mathcal{G}]\!]$ are equivalent.*

Proof. We construct a language isomorphism $\Psi = (\psi_{\mathcal{A}}, \psi_{\mathcal{O}})$ between the normal form sublanguages, where $\psi_{\mathcal{A}}(t) = \Pi^{\ominus}[\![t]\!]$ for $t \in \mathcal{A}(\mathcal{G})^{\mathsf{NF}}$ and $\psi_{\mathcal{O}}(u) = \Pi^{\ominus}[\![u\Downarrow]\!]$ for $u \in \mathcal{O}(\mathcal{G})^{\mathsf{NF}}$.

By Lemma 6.2, both $\psi_{\mathcal{A}}$ and $\psi_{\mathcal{O}}$ are bijective functions. Note that since we are restricting to the β-normal form sublanguage $\mathcal{O}(\mathcal{G})^{\mathsf{NF}}$, we have $\psi_{\mathcal{O}}(u) = \Pi^{\ominus}[\![u\Downarrow]\!] = \Pi^{\ominus}[\![u]\!]$ for all $u \in \mathcal{O}(\mathcal{G})^{\mathsf{NF}}$.

For lexicon commutativity, we need $\psi_{\mathcal{O}}(\mathcal{L}(t)) = \Pi^{\ominus}[\![\mathcal{L}]\!](\psi_{\mathcal{A}}(t))$ for any $t \in \mathcal{A}(\mathcal{G})^{\mathsf{NF}}$. Since $\psi_{\mathcal{A}}(t) = \Pi^{\ominus}[\![t]\!]$ and $\psi_{\mathcal{O}}(\mathcal{L}(t)) = \Pi^{\ominus}[\![\mathcal{L}(t)\Downarrow]\!]$, this follows directly from Lemma 6.3.

Therefore, Ψ is a language isomorphism, and $\mathcal{G} \cong \Pi^{\ominus}[\![\mathcal{G}]\!]$. □

6.2.2 Record Elimination

Having eliminated dependent products from AACGs, we now turn to the second transformation: record elimination. The key insight is that records can be systematically flattened into sequences of their components. Record types are decomposed into sequences of their constituent feature structure types, while record terms are flattened into sequences of their field values. We begin by defining the transformation for kinds and canonical proper types in normal form.

Definition 6.13 (Recordless Kinds)**.** Let $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ be a feature structure signature and $k, k' \in \mathcal{K}_{\mathcal{D}}(\mathfrak{F})$ be kinds. The *recordless translation of a kind*, denoted $\mathcal{R}^{\ominus}[\![k]\!]$, is defined inductively as follows:

$$\begin{aligned} \mathcal{R}^{\ominus}[\![\mathsf{Type}]\!] &= \mathsf{Type} \\ \mathcal{R}^{\ominus}[\![\varepsilon \Rrightarrow k']\!] &= \varepsilon \Rrightarrow \mathcal{R}^{\ominus}[\![k']\!] \\ \mathcal{R}^{\ominus}[\![[l_i : \Phi_i]_{i=1}^{n} \Rrightarrow k']\!] &= \mathcal{R}^{\ominus}[\![\Phi_1 \Rrightarrow \ldots \Rrightarrow \Phi_n \Rrightarrow k']\!]. \end{aligned}$$

Definition 6.14 (Recordless Canonical Proper Types)**.** Let α be a canonical proper type in β-normal form such that $\vdash_{\Sigma} \alpha :: \mathsf{Type}$. The recordless translation of α, denoted $\mathcal{R}^{\ominus}[\![\alpha]\!]$, is defined by cases according to Lemma 5.5:

1. If $\alpha = \beta \multimap \gamma$ where $\mathsf{NF}(\beta)$ and $\mathsf{NF}(\gamma)$, then $\mathcal{R}^{\ominus}[\![\alpha]\!] = \mathcal{R}^{\ominus}[\![\beta]\!] \multimap \mathcal{R}^{\ominus}[\![\gamma]\!]$;
2. If $\alpha = a\, f_1 \ldots f_n$ for some atomic type a and closed feature structure arguments $f_1, \ldots, f_n$ (where $n \geq 0$) where each $\mathsf{NF}(f_i)$, then:
   - Base case: If $n = 0$, then $\mathcal{R}^{\ominus}[\![\alpha]\!] = a$;
   - Recursive case: If $n > 0$, then:
     - If f_n is an enumeral, then $\mathcal{R}^{\ominus}[\![\alpha]\!] = \mathcal{R}^{\ominus}[\![a\, f_1 \ldots f_{n-1}]\!]\, f_n$;
     - If f_n is a record $[l_{n,1} = g_{n,1}, \ldots, l_{n,k_n} = g_{n,k_n}]$, then $\mathcal{R}^{\ominus}[\![\alpha]\!] = \mathcal{R}^{\ominus}[\![a\, f_1 \ldots f_{n-1}\, g_{n,1} \ldots g_{n,k_n}]\!]$.

Definition 6.15 (Recordless Affix HOS)**.** Let $\Sigma = \langle A, \mathfrak{F}, \kappa, C, \tau \rangle$ be an affix higher-order signature where $\mathfrak{F} = \langle E, \mathcal{E}, L \rangle$ is a feature structure signature. The recordless translation of Σ is defined as:

$$\mathcal{R}^{\ominus}[\![\Sigma]\!] = \langle A, \mathcal{R}^{\ominus}[\![\mathfrak{F}]\!], \mathcal{R}^{\ominus}[\![\kappa]\!], C, \mathcal{R}^{\ominus}[\![\tau]\!] \rangle$$

where:

1. $\mathcal{R}^{\ominus}[\![\mathfrak{F}]\!] = \langle E, \mathcal{E}, \varnothing \rangle$ (records are eliminated from the feature structure signature);

2. $\mathcal{R}^{\ominus}[\![\kappa]\!] : A \to \mathcal{K}(\mathcal{R}^{\ominus}[\![\mathfrak{F}]\!])$ is defined as $\mathcal{R}^{\ominus}[\![\kappa]\!](a) = \mathcal{R}^{\ominus}[\![\kappa(a)]\!]$ for all $a \in A$;
3. $\mathcal{R}^{\ominus}[\![\tau]\!] : C \to \mathcal{T}(A, \mathcal{R}^{\ominus}[\![\mathfrak{F}]\!])$ is defined as $\mathcal{R}^{\ominus}[\![\tau]\!](c) = \mathcal{R}^{\ominus}[\![\tau(c)]\!]$ for all $c \in C$, where the recordless translation of types is defined as in Definition 6.14.

As the dependent product is no longer present, the feature application and feature abstraction at the term level have been eliminated. Hence, terms are left unchanged by the recordless translation.

It remains to define the recordless lexicon, which presents the main difficulty of this transformation. The challenge lies in handling binders over feature structures: when a type abstraction binds a record-typed variable, we must transform it into a sequence of binders for the individual field components. We address this through a transformation inspired by currying in the λ-calculus with products, where functions accepting tuple arguments are restructured into sequences of nested single-argument functions.

The following lemma formalizes this transformation.

Lemma 6.4 (Currying Type Abstractions)**.** *Let $\mathcal{R} = [l_1 : \Phi_1, \ldots, l_n : \Phi_n]$ be a record type, and let $r = [l_1 = x_1, \ldots, l_n = x_n]$ be a record term where each x_i is a fresh variable. Then: If $\Gamma \vdash_\Sigma \Lambda x.\, \alpha :: \mathcal{R} \Rrightarrow k$ then $\Gamma \vdash_\Sigma \Lambda x_1 \ \ldots \ x_n.\alpha[x := r] :: \Phi_1 \Rrightarrow \ldots \Rrightarrow \Phi_n \Rrightarrow k$;*

Proof. Direct.

1.	$\Gamma \vdash_\Sigma \Lambda x.\, \alpha :: \mathcal{R} \Rrightarrow k$	by assumption
2.	$\Gamma' = \{x_i : \Phi_i\}_{i=1}^n$ $\Gamma, \Gamma' \vdash_\Sigma \Lambda x.\, \alpha :: \mathcal{R} \Rrightarrow k$	by lemma 5.2
3.	$\Gamma, \Gamma', x : \mathcal{R} \vdash_\Sigma \alpha :: k$	by inversion
4.	$\Gamma, \Gamma' \vdash_\Sigma x_i : \Phi_i$	by VAR-FEAT
5.	$\Gamma, \Gamma' \vdash_\Sigma [l_i = x_i]_{i=1}^n : [l_i : \Phi_i]_{i=1}^n$	by RECORD-FEAT
6.	$\Gamma, \Gamma' \vdash_\Sigma \alpha[x := [l_i = x_i]_{i=1}^n] :: k$	by lemma 5.3
7.	$\Gamma \vdash_\Sigma \Lambda x_1 \ \ldots \ x_n.\alpha[x := r] :: \Phi_1 \Rrightarrow \ldots \Rrightarrow \Phi_n \Rrightarrow k$	by ABSTR-TYPE

□

Definition 6.16 (Recordless Translation of Normal Form Types)**.** Let α be a type in β-normal form such that $\Gamma \vdash_\Sigma \alpha :: k$ for some kind k. Assume that there is no variable in the context Γ that is typed by a record type. The recordless translation of α, denoted $\mathcal{R}^{\ominus}[\![\alpha]\!]^k$, is defined inductively as follows:

1. If $\alpha = a$ for some atomic type a and $k = \kappa(a)$, then $\mathcal{R}^{\ominus}[\![\alpha]\!]^k = a$;
2. If $\alpha = \beta \multimap \gamma$ where $\mathsf{NF}(\beta)$, $\mathsf{NF}(\gamma)$, and $k = \mathsf{Type}$, then $\mathcal{R}^{\ominus}[\![\alpha]\!]^k = \mathcal{R}^{\ominus}[\![\beta]\!]^{\mathsf{Type}} \multimap \mathcal{R}^{\ominus}[\![\gamma]\!]^{\mathsf{Type}}$;
3. If $\alpha = \Lambda x.\, \beta$ where $\mathsf{NF}(\beta)$ and $k = \varepsilon \Rrightarrow k'$, then $\mathcal{R}^{\ominus}[\![\alpha]\!]^k = \Lambda x.\, \mathcal{R}^{\ominus}[\![\beta]\!]^{k'}$;
4. If $\alpha = \Lambda x.\, \beta$ where $\mathsf{NF}(\beta)$ and $k = [l_1 : \Phi_1, \ldots, l_n : \Phi_n] \Rrightarrow k'$, then:

   $$\mathcal{R}^{\ominus}[\![\alpha]\!]^k = \Lambda x_1. \ldots \Lambda x_n.\, \mathcal{R}^{\ominus}[\![\beta[x := [l_1 = x_1, \ldots, l_n = x_n]] \Downarrow]\!]^{k'}$$

   where each x_i is fresh;
5. If $\alpha = \beta\, f$ where β is in normal form and f is a feature structure term in normal form, then:
   - If f is an enumeral or variable, then $\mathcal{R}^{\ominus}[\![\alpha]\!]^k = \mathcal{R}^{\ominus}[\![\beta]\!]^{\Phi \Rrightarrow k}\, f$;
   - If $f = [l_1 = f_1, \ldots, l_n = f_n]$, then $\mathcal{R}^{\ominus}[\![\alpha]\!]^k = \mathcal{R}^{\ominus}[\![\cdots \beta\, f_1\, f_2 \cdots f_n]\!]^k$;
   - If $f = x.l$ is a field selection, this case is ruled out by our assumption;
6. If $\alpha = \mathsf{Case}_x\ \{e_i \to \alpha_i\}_{i=1}^n$ where each $\mathsf{NF}(\alpha_i)$ and $k = \varepsilon \Rrightarrow k'$, then:

   $$\mathcal{R}^{\ominus}[\![\alpha]\!]^k = \mathsf{Case}_x\ \{e_i \to \mathcal{R}^{\ominus}[\![\alpha_i]\!]^{k'}\}_{i=1}^n$$

Definition 6.17 (Recordless Lexicon)**.** Let $\mathcal{L} : \Sigma_1 \to \Sigma_2 = \langle F, G \rangle$ be an affix lexicon between two affix higher-order signatures $\Sigma_1 = \langle A_1, \mathfrak{F}_1, \kappa_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, \mathfrak{F}_2, \kappa_2, C_2, \tau_2 \rangle$. Let $\mathcal{R}^{\ominus}[\![\Sigma_1]\!]$ and $\mathcal{R}^{\ominus}[\![\Sigma_2]\!]$ be their recordless translations. The recordless translation of $\mathcal{L}$, written $\mathcal{R}^{\ominus}[\![\mathcal{L}]\!] : \mathcal{R}^{\ominus}[\![\Sigma_1]\!] \to \mathcal{R}^{\ominus}[\![\Sigma_2]\!] = \langle \mathcal{R}^{\ominus}[\![F]\!], \mathcal{R}^{\ominus}[\![G]\!] \rangle$, is defined as follows:

1. For each atomic type family $a \in A_1$, $\mathcal{R}^{\ominus}[\![F]\!](a) = \mathcal{R}^{\ominus}[\![F(a) \Downarrow]\!]^{\kappa_1(a)}$;
2. For each constant $c \in C_1$, $\mathcal{R}^{\ominus}[\![G]\!](c) = G(c)$.

Definition 6.18 (Recordless AACG)**.** Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ be an AACG. The recordless translation of $\mathcal{G}$, written $\mathcal{R}^{\ominus}[\![\mathcal{G}]\!]$, is defined as:

$$\mathcal{R}^{\ominus}[\![\mathcal{G}]\!] = \langle \mathcal{R}^{\ominus}[\![\Sigma_1]\!], \mathcal{R}^{\ominus}[\![\Sigma_2]\!], \mathcal{R}^{\ominus}[\![\mathcal{L}]\!], \mathcal{R}^{\ominus}[\![\mathbf{S}]\!] \rangle$$

where $\mathcal{R}^{\ominus}[\![\Sigma_1]\!]$ and $\mathcal{R}^{\ominus}[\![\Sigma_2]\!]$ are recordless translations of the respective signatures, $\mathcal{R}^{\ominus}[\![\mathcal{L}]\!]$ is the recordless translation of the lexicon, and $\mathcal{R}^{\ominus}[\![\mathbf{S}]\!]$ is the recordless translation of the distinguished type.

Lemma 6.5 (Kinding Preservation Under Recordless Transformation)**.** *Let* $\Sigma = \langle A, \mathfrak{F}, \kappa, C, \tau \rangle$ *be an affix higher-order signature, and let* $\mathcal{R}^{\ominus}[\![\Sigma]\!]$ *be its recordless translation. If* $\vdash_{\Sigma} \alpha :: k$ *where* $\mathsf{NF}(\alpha)$*, then* $\vdash_{\mathcal{R}^{\ominus}[\![\Sigma]\!]} \mathcal{R}^{\ominus}[\![\alpha]\!] :: \mathcal{R}^{\ominus}[\![k]\!]$*.*

Proof. By induction on the kinding derivation $\vdash_{\Sigma} \alpha :: k$. □

Lemma 6.6 (Typing Preservation Under Recordless Transformation)**.** *Let* $\Sigma = \langle A, \mathfrak{F}, \kappa, C, \tau \rangle$ *be an affix higher-order signature, and let* $\mathcal{R}^{\ominus}[\![\Sigma]\!]$ *be its recordless translation. If* $\vdash_{\Sigma} t : \alpha$ *where* $\mathsf{NF}(\alpha)$ *and* $\vdash_{\Sigma} \alpha :: \mathsf{Type}$*, then* $\vdash_{\mathcal{R}^{\ominus}[\![\Sigma]\!]} t : \mathcal{R}^{\ominus}[\![\alpha]\!]$*.*

Proof. By induction on the typing derivation $\vdash_{\Sigma} t : \alpha$. □

Theorem 6.3 (Recordless Equivalence)**.** *Let* $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ *be a* Π*-less Affix ACG , and let* $\mathcal{R}^{\ominus}[\![\mathcal{G}]\!]$ *be its recordless translation. Then* $\mathcal{G}$ *and* $\mathcal{R}^{\ominus}[\![\mathcal{G}]\!]$ *are equivalent.*

Proof. The recordless transformation affects only types and kinds while leaving all λ-terms syntactically unchanged. Since we are working with a Π-less AACG, no feature structure applications or abstractions remain at the term level, and record elimination operates solely on the type structure.

We construct the language isomorphism $\Psi = (\psi_{\mathcal{A}}, \psi_{\mathcal{O}})$ where $\psi_{\mathcal{A}}(t) = t$ for $t \in \mathcal{A}(\mathcal{G})$ and $\psi_{\mathcal{O}}(u) = u$ for $u \in \mathcal{O}(\mathcal{G})$.

For well-definedness, if $t \in \mathcal{A}(\mathcal{G})$, then $\vdash_{\Sigma_1} t : \mathbf{S}$. By Lemma 6.6, we have $\vdash_{\mathcal{R}^{\ominus}[\![\Sigma_1]\!]} t : \mathcal{R}^{\ominus}[\![\mathbf{S}]\!]$, so $t \in \mathcal{A}(\mathcal{R}^{\ominus}[\![\mathcal{G}]\!])$. The same argument applies to $\psi_{\mathcal{O}}$ using the homomorphic extension of the recordless lexicon.

Bijectivity follows immediately since both functions are identity maps on syntactically identical term sets. For lexicon commutativity, we need $\psi_{\mathcal{O}}(\mathcal{L}(t)) = \mathcal{R}^{\ominus}[\![\mathcal{L}]\!](\psi_{\mathcal{A}}(t))$ for any $t \in \mathcal{A}(\mathcal{G})$. Since $\psi_{\mathcal{A}}(t) = t$ and terms are preserved by the recordless transformation, we have $\mathcal{L}(t) = \mathcal{R}^{\ominus}[\![\mathcal{L}]\!](t)$, giving us $\psi_{\mathcal{O}}(\mathcal{L}(t)) = \mathcal{L}(t) = \mathcal{R}^{\ominus}[\![\mathcal{L}]\!](t) = \mathcal{R}^{\ominus}[\![\mathcal{L}]\!](\psi_{\mathcal{A}}(t))$.

Therefore, $\mathcal{G}$ and $\mathcal{R}^{\ominus}[\![\mathcal{G}]\!]$ are equivalent. □

It remains to show that any AACG using only type families can be transformed into an equivalent core ACG.

Theorem 6.4 (Feature Structure Elimination)**.** *Let* $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$ *be an AACG where all types in* Σ_1 *and* Σ_2 *are either atomic types, linear implicative types, or type families indexed by enumerations. There exists a transformation* $\mathcal{F}^{\ominus}$ *that produces an equivalent embedded core ACG* $\mathcal{F}^{\ominus}[\![\mathcal{G}]\!]$*.*

Proof. The transformation follows the classical technique of encoding finite feature structures through atomic types:

1. For each type family a indexed by enumerations $\varepsilon_1, \ldots, \varepsilon_n$, we construct a bijection $\mathcal{F}_a^{\ominus}$ that maps each tuple $(e_1, \ldots, e_n) \in \varepsilon_1 \times \cdots \times \varepsilon_n$ to an atomic type $a_{e_1,\ldots e_n}$;

2. We define a type translation function, $\mathcal{F}^{\ominus}$, that maps each β-normal indexed type family member $a\ e_1\ \ldots\ e_n$ to $\mathcal{F}_a^{\ominus}(e_1, \ldots, e_n)$ and extends homomorphically to implicative types;

3. The abstract and object signatures are transformed by applying the type translation $\mathcal{F}^{\ominus}$ to the type of each constant;

4. For the new lexicon, $\mathcal{F}^{\ominus}(\mathcal{L})$, the entry for each constant c remains the term $\mathcal{L}(c)$. For each type family a indexed by enumerations $\varepsilon_1, \ldots, \varepsilon_n$, we map $\mathcal{F}^{\ominus}[\![\mathcal{L}]\!](a_{e_1 \ldots e_n})$ to $\mathcal{F}^{\ominus}(\mathcal{L}(a\ e_1\ \ldots\ e_n) \Downarrow)$.

Since only type annotations change through the systematic replacement of type family applications with atomic types, typing is easily preserved and the terms remain syntactically identical. We therefore define the language isomorphism $\Psi = (\psi_{\mathcal{A}}, \psi_{\mathcal{O}})$ as identity morphisms where $\psi_{\mathcal{A}}(t) = t$ and $\psi_{\mathcal{O}}(u) = u$, making bijectivity immediate and lexicon commutativity trivial.

Therefore, $\mathcal{G}$ and $\mathcal{F}^{\ominus}[\![\mathcal{G}]\!]$ are equivalent. □

The three-step transformation sequence from Affix ACG to Π-less Affix ACG to recordless Affix ACG to core ACG establishes the main theoretical result: AACGs are equivalent to core ACGs. Each transformation preserves language isomorphisms, providing a constructive proof of this equivalence.

This result has several important consequences. First, it establishes a conservative extension: every core ACG can be viewed as an AACG, while every AACG can be systematically reduced to a core ACG, proving that feature structure integration adds no fundamental expressive power beyond the original formalism. Second, AACGs inherit all desirable theoretical properties of core ACGs, including decidability and polynomial-time parsing for second-order grammars via Kanazawa's reduction. Finally, despite equivalent expressive power, AACGs enable more compact and maintainable grammar representations through feature structure factorization.

6.3 AACG Parsing

The parsing complexity of an AACG is determined by both its order and the complexity of its translation into a core ACG. In particular, for second-order AACGs, polynomial-time parsing is achieved by converting them into a core ACG and applying Kanazawa's reduction.

Consider the introductory grammar G_{CFG}. Applying Kanazawa's reduction to parse the string *the stakeholder sells stocks* yields the Example 6.1. The input string has been directly integrated into the program as facts, making this a complete executable example that combines both the grammar rules and the specific input to be parsed. Note how singular and plural forms are distinguished in the predicate names, allowing the grammar to enforce number agreement between subjects and verbs.

Example 6.1 ($P_{\mathcal{G}_{\mathsf{CFG}}}$).

$$
\begin{aligned}
[\text{SENTENCE}_{sg}] \quad & \mathsf{S}(p_0, p_2) \leftarrow \mathsf{NP}_{sg}(p_0, p_1), \mathsf{VP}_{sg}(p_1, p_2).\\
[\text{SENTENCE}_{pl}] \quad & \mathsf{S}(p_0, p_2) \leftarrow \mathsf{NP}_{pl}(p_0, p_1), \mathsf{VP}_{pl}(p_1, p_2).\\
[\text{NOUN_PHRASE}_{sg}] \quad & \mathsf{NP}_{sg}(p_0, p_2) \leftarrow \mathsf{Det}_{sg}(p_0, p_1), \mathsf{N}_{sg}(p_1, p_2).\\
[\text{NOUN_PHRASE}_{pl}] \quad & \mathsf{NP}_{pl}(p_0, p_2) \leftarrow \mathsf{Det}_{pl}(p_0, p_1), \mathsf{N}_{pl}(p_1, p_2).\\
[\text{VERB_PHRASE}_{sg_sg}] \quad & \mathsf{VP}_{sg}(p_0, p_2) \leftarrow \mathsf{V}_{sg}(p_0, p_1), \mathsf{NP}_{sg}(p_1, p_2).\\
[\text{VERB_PHRASE}_{sg_pl}] \quad & \mathsf{VP}_{sg}(p_0, p_2) \leftarrow \mathsf{V}_{sg}(p_0, p_1), \mathsf{NP}_{pl}(p_1, p_2).\\
[\text{VERB_PHRASE}_{pl_sg}] \quad & \mathsf{VP}_{pl}(p_0, p_2) \leftarrow \mathsf{V}_{pl}(p_0, p_1), \mathsf{NP}_{sg}(p_1, p_2).\\
[\text{VERB_PHRASE}_{pl_pl}] \quad & \mathsf{VP}_{pl}(p_0, p_2) \leftarrow \mathsf{V}_{pl}(p_0, p_1), \mathsf{NP}_{pl}(p_1, p_2).\\
[\text{THE}_{sg}] \quad & \mathsf{Det}_{sg}(p_0, p_1) \leftarrow \mathit{the}(p_0, p_1).\\
[\text{THE}_{pl}] \quad & \mathsf{Det}_{pl}(p_0, p_1) \leftarrow \mathit{the}(p_0, p_1).\\
[\text{STAKEHOLDER}] \quad & \mathsf{N}_{sg}(p_0, p_1) \leftarrow \mathit{stakeholder}(p_0, p_1).\\
[\text{STAKEHOLDERS}] \quad & \mathsf{N}_{pl}(p_0, p_1) \leftarrow \mathit{stakeholders}(p_0, p_1).\\
[\text{STOCK}] \quad & \mathsf{N}_{sg}(p_0, p_1) \leftarrow \mathit{stock}(p_0, p_1).\\
[\text{STOCKS}] \quad & \mathsf{N}_{pl}(p_0, p_1) \leftarrow \mathit{stocks}(p_0, p_1).\\
[\text{SELL}] \quad & \mathsf{V}_{sg}(p_0, p_1) \leftarrow \mathit{sells}(p_0, p_1).\\
[\text{SELLS}] \quad & \mathsf{V}_{pl}(p_0, p_1) \leftarrow \mathit{sell}(p_0, p_1).\\
& \mathit{the}(0, 1).
\end{aligned}
$$

$$stakeholder(1,2).$$
$$sells(2,3).$$
$$stocks(3,4).$$

While this direct translation demonstrates polynomial-time parsing, it sacrifices the factorization advantages that feature structures provide in AACGs. The flattened representation expands all feature variations explicitly, resulting in redundant rules as seen with singular/plural distinctions. The explicit enumeration approach becomes increasingly inefficient as grammatical features multiply. To develop more efficient parsing strategies that preserve Affix ACG 's factorization benefits, we must first understand how ACG lexicons handle variations in interpretation. Constants whose interpretations involve case analysis create multiple paths, similar to the alternation operator | in CFGs. We formalize this concept:

Definition 6.19 (Branching Grammars)**.** An Affix ACG is *branching* if its lexicon contains at least one branching term or type.

We make two assumptions about the AACGs discussed in this section:

1. They are recordless;
2. They are non-branching.

The recordless assumption stems from standard Datalog's limitations: it lacks native record support. While Datalog extensions exist that can handle records (Ceri, Gottlob, and Tanca 1990a), maintaining this restriction allows us to focus on the core parsing techniques without additional complexity. The non-branching assumption is not restrictive in practice: any branching grammar can be transformed into a non-branching equivalent through techniques analogous to dependent product elimination.

We propose two alternative parsing strategies that leverage feature structures. The first approach, *Context-Free Backbone with Feature Filtering*, separates parsing into two phases: parse using a simplified context-free backbone of the grammar, then filter the results through feature structure constraints. The second approach, *Feature-Aware Datalog Program*, models directly feature structure in Datalog, drawing inspiration from Definite Clause Grammars (DCGs, F. C. Pereira and D. H. Warren 1980).

6.3.1 Context-Free Backbone with Feature Filtering

The first approach to parsing AACGs begins by extracting a context-free backbone through a *dependency-less translation* that produces a non-linear ACG. This translation removes feature structures from type families resulting in atomic types but keeps them in λ-terms. Enumerals are then modeled as constants typed by atomic types, and dependent products are transformed into non-linear implications. A key property of this translation is that, since enumerations are modeled as atomic types, the non-linear terms are almost affine.

Definition 6.20 (Dependency-less Affix Higher-Order Signature)**.** Let $\mathfrak{F} = \langle E, \mathcal{E}, \emptyset \rangle$ be a feature structure signature and let $\Sigma = \langle A, \mathfrak{F}, \kappa, C, \tau \rangle$ be an affix higher-order signature. The dependency-less transformation of Σ produces a core higher-order signature $D(\Sigma) = \langle D(A), D(C), D(\tau) \rangle$ where:

1. For each enumeration $\varepsilon \in \mathcal{E}$, we introduce a fresh atomic type a_ε. The set of all such types is denoted:
$$A_{\mathcal{E}} = \{a_\varepsilon \mid \varepsilon \in \mathcal{E}\}$$

2. The set of atomic types $D(A)$ is the union of the original type family symbols and the fresh enumeration types:
$$D(A) = A \cup A_{\mathcal{E}};$$

3. The set of constants $D(C)$ is the union of the original constants and the enumerals:
$$D(C) = C \cup E;$$

4. The typing function $D(\tau)$ maps:
   - Each enumeral $e \in \varepsilon$ to the atomic type a_ε corresponding to its enumeration;
   - Each original constant $c \in C$ to its transformed type $D(\tau(c))$; where D is defined inductively on affix types as follows:

$$\begin{aligned} D(a) &= a \text{ for } a \in A \\ D(\alpha\, f) &= D(\alpha) \\ D(\Lambda x.\, \alpha) &= D(\alpha) \\ D(\alpha \multimap \beta) &= D(\alpha) \to D(\beta) \\ D(\Pi x\colon \varepsilon.\, \alpha) &= a_\varepsilon \to D(\alpha). \end{aligned}$$

Definition 6.21 (Dependency-less Affix Lexicon)**.** Let $\mathcal{L} = \langle F, G\rangle : \Sigma_1 \to \Sigma_2$ be an Affix lexicon between signatures $\Sigma_1 = \langle A_1, \mathfrak{F}_1, \kappa_1, C_1, \tau_1\rangle$ and $\Sigma_2 = \langle A_2, \mathfrak{F}_2, \kappa_2, C_2, \tau_2\rangle$. The dependency-less transformation $D(\mathcal{L}) : D(\Sigma_1) \to D(\Sigma_2)$ is a pair $\langle D(F), D(G)\rangle$ where:

1. If $a \in A_1$, then $D(F)(a) = D(F(a))$;
2. If $\varepsilon \in \mathcal{E}_1$, then $D(F)(a_\varepsilon) = a_\varepsilon$.
3. If $e \in E_1$ then $D(G)(e) = e$;
4. If $c \in C_1$, then $D(G)(c) = D(G(C))$; where D is defined inductively on terms as follows:

$$\begin{aligned} D(e) &= e \\ D(c) &= c \\ D(x) &= x \\ D(\lambda^\circ x.\, t) &= \lambda x.\, D(t) \\ D(\lambda x.\, t) &= \lambda x.\, D(t) \\ D(t \bullet f) &= D(t)\, D(f) \\ D(t_1\, t_2) &= D(t_1)\, D(t_2). \end{aligned}$$

Definition 6.22 (Dependency-less Affix Grammar)**.** Let $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S}\rangle$ be an AACG. The dependency-less transformation produces a core ACG defined as:

$$D(\mathcal{G}) = \langle D(\Sigma_1), D(\Sigma_2), D(\mathcal{L}), D(\mathbf{S})\rangle.$$

The almost affine nature of these terms enables us to apply Bourreau's transformation (Bourreau 2012) to construct a recognizer for the backbone grammar. Example 6.2 presents a simplified version of the program generated by this transformation.

Example 6.2 ($P_{D(\mathcal{G}_{\mathsf{CFG}})}$).

$$\begin{aligned} [sg] \quad & \mathtt{num}(x) \leftarrow sg(x). \\ [pl] \quad & \mathtt{num}(x) \leftarrow pl(x). \\ [\textsc{sentence}] \quad & \mathsf{S}(p_0, p_2) \leftarrow \mathtt{num}(x), \mathsf{NP}(p_0, p_1), \mathsf{VP}(p_1, p_2). \\ [\textsc{noun_phrase}] \quad & \mathsf{NP}(p_0, p_2) \leftarrow \mathtt{num}(x), \mathsf{Det}(p_0, p_1), \mathsf{N}(p_1, p_2). \\ [\textsc{verb_phrase}] \quad & \mathsf{VP}(p_0, p_2) \leftarrow \mathtt{num}(x), \mathtt{num}(y), \mathsf{V}(p_0, p_1), \mathsf{NP}(p_1, p_2). \end{aligned}$$

$$
\begin{aligned}
[\textsc{the}] \quad & \mathsf{Det}(p_0, p_1) \leftarrow \mathtt{num}(x), \mathit{the}(p_0, p_1).\\
[\textsc{stakeholder}] \quad & \mathsf{N}(p_0, p_1) \leftarrow \mathit{stakeholder}(p_0, p_1).\\
[\textsc{stakeholders}] \quad & \mathsf{N}(p_0, p_1) \leftarrow \mathit{stakeholders}(p_0, p_1).\\
[\textsc{stock}] \quad & \mathsf{N}(p_0, p_1) \leftarrow \mathit{stock}(p_0, p_1).\\
[\textsc{stocks}] \quad & \mathsf{N}(p_0, p_1) \leftarrow \mathit{stocks}(p_0, p_1).\\
[\textsc{sell}] \quad & \mathsf{V}(p_0, p_1) \leftarrow \mathit{sells}(p_0, p_1).\\
[\textsc{sells}] \quad & \mathsf{V}(p_0, p_1) \leftarrow \mathit{sell}(p_0, p_1).\\
& \mathit{the}(0, 1).\\
& \mathit{stakeholder}(1, 2).\\
& \mathit{sells}(2, 3).\\
& \mathit{stocks}(3, 4).
\end{aligned}
$$

However, this program may generate derivations that violate the feature structure constraints present in the original AACG. For instance, examining rule SENTENCE, we observe that number features $\mathtt{num}(x)$ permit both singular $\mathit{sg}(x)$ and plural $\mathit{pl}(x)$ to be selected independently, without enforcing agreement between subjects and verbs. Consequently, the program could derive both *the stakeholder sells stocks* and *the stakeholder sell stocks* with equal validity, despite the latter violating number agreement in English. A filtering process must therefore be introduced to ensure that only grammatically valid derivations from the transformed program are accepted.

There are two approaches for implementing this filtering process. The first leverages the closure properties of ACGs, specifically, that ACGs are closed under intersection with regular sets (Kanazawa 2006). A regular tree language can be defined that excludes applications of constants that would fail to typecheck in the original abstract Affix higher-order signature. The intersection of this regular language with the parsed results ensures that only valid feature structure combinations remain. The second filtering approach is more direct: we can use a typechecker defined over the original abstract affix higher-order signature and use it to validate each candidate abstract term.

One major concern with this approach is that it remains unclear whether the Datalog programs produced through Bourreau's transformation inherit the LOGCFL complexity properties of the programs generated by Kanazawa's reduction. Additionally, the necessity to split the process into two distinct phases: a backbone parsing step followed by a filtering phase to enforce feature structure constraints, introduces extra computational

overhead. This two-phase approach may potentially slow down the overall parsing process, thus challenging the primary objective of using feature structures to improve parsing speed.

6.3.2 Feature-Aware Datalog Program

The second approach to parsing AACGs encodes feature structures directly within Datalog programs, preserving rather than eliminating feature information from the abstract signature. Consider a recordless non-branching AACG $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, \mathbf{S} \rangle$, its Π-eliminated equivalent form $\mathcal{G}^{\mathcal{E}} = \langle \Sigma_1^{\mathcal{E}}, \Sigma_2^{\mathcal{E}}, \mathcal{L}^{\mathcal{E}}, \mathbf{S}^{\mathcal{E}} \rangle$, and its corresponding equivalent core ACG $\mathcal{G}^{\multimap} = \langle \Sigma_1^{\multimap}, \Sigma_2^{\multimap}, \mathcal{L}^{\multimap}, \mathbf{S}^{\multimap} \rangle$. We can develop parsing algorithms by transforming the Datalog program $P_{\mathcal{G}^{\multimap}}$ and its instance $\mathbf{I}_t$ to incorporate feature structure information, while maintaining semantic equivalence. For simplicity of presentation, we assume that $\mathbf{I}_t$ is incorporated directly into the program.

Predicate Despecialization

Recall Example 6.1, where feature information appears embedded in predicate names (e.g., NP_{sg} vs NP_{pl} for singular and plural noun phrases). This explicit enumeration approach leads to program bloat as feature combinations multiply. The key insight of this transformation is that feature information can be moved from predicate names into additional arguments, transforming predicates like $\mathsf{NP}_{sg}(x, y)$ into $\mathsf{NP}(sg, x, y)$. By parameterizing features rather than encoding them in predicate names, a representation closer to $\mathcal{G}^{\mathcal{E}}$ is achieved while reducing the number of distinct predicates in the program. We formalize this technique for general Datalog programs in the following lemma:

Lemma 6.7 (Predicate Despecialization)**.** *Let $\varepsilon = \{e_1, \ldots, e_k\}$ be a finite set, and let P be a Datalog program that uses a family of predicate symbols $\{p^e\}_{e \in \varepsilon}$ (possibly alongside other predicates). Define a transformation τ on atoms as follows:*

$$\tau\Big(p^e(t_1, \ldots, t_n)\Big) = p(e, t_1, \ldots, t_n) \quad \textit{for every } e \in \varepsilon$$
$$\tau\Big(p'(t_1, \ldots, t_n)\Big) = p'(t_1, \ldots, t_n) \quad \textit{with } p' \neq p.$$

Let P' be the program obtained by applying τ to every atom in P. Then P and P' are semantically equivalent; that is, for every derivation in P producing an fact A, there is a corresponding derivation in P' producing $\tau(A)$, and vice versa.

Proof. We proceed by induction on the height h of derivations in P.

1. Base Case ($h = 1$):

   Any fact A in P is either of the form $p^e(c_1, \ldots, c_n)$ (yielding $\tau(A) = p(e, c_1, \ldots, c_n)$) or not (in which case $\tau(A) = A$). Hence, the correspondence holds.

2. Inductive Step: Assume every derivation of height at most h in P corresponds to a derivation in P' yielding $\tau(A)$. Consider a derivation of height $h + 1$ that applies a rule

   $$R: \quad H \leftarrow B_1, \ldots, B_k,$$

   with substitution θ so that $\theta(H) = A$ and each $\theta(B_i)$ is derived in height at most h. Transform R as

   $$\tau(R): \quad \tau(H) \leftarrow \tau(B_1), \ldots, \tau(B_k).$$

   By the induction hypothesis, each $\theta(B_i)$ corresponds to $\tau(\theta(B_i))$ in P'. Thus, applying $\tau(R)$ with θ in P' yields $\tau(\theta(H)) = \tau(A)$. The reverse direction is analogous.

   Thus, by induction the semantics of P and P' coincide.

□

Theorem 6.5 (Program Despecialization for Indexed Predicates)**.** *Let*

$$\mathcal{I} \;=\; \left\{ p^{(e_1,\ldots,e_k)} \mid k \geq 1,\ (e_1, \ldots, e_k) \in \mathcal{E}_p \right\}$$

be a set of indexed predicate symbols, where for each base predicate p the index tuple $(e_1, \ldots, e_k)$ is drawn from the finite set $\mathcal{E}_p \subseteq \varepsilon_1^p \times \cdots \times \varepsilon_k^p$. Let P be a Datalog program whose predicate symbols are a superset of $\mathcal{I}$. Define a transformation τ on atoms by

$$\tau\Big(p^{(e_1,\ldots,e_k)}(t_1, \ldots, t_n)\Big) \;=\; p(e_1, \ldots, e_k, t_1, \ldots, t_n)$$
$$\tau\Big(p'(t_1, \ldots, t_n)\Big) = p'(t_1, \ldots, t_n) \quad \textit{with } p' \neq p.$$

Let P' be the program obtained by applying τ to every atom in P. Then P and P' are semantically equivalent; that is, for every derivation in P producing an atom

$$p^{(e_1,\ldots,e_k)}(t_1, \ldots, t_n),$$

there is a corresponding derivation in P' producing

$$p(e_1, \ldots, e_k, t_1, \ldots, t_n),$$

and vice versa.

Proof. By sucessive applications of Lemma 6.7. □

Application to AACG Programs

Let $\Sigma_1^{\mathcal{E}} = \langle A_1^{\mathcal{E}}, \mathfrak{F}_1^{\mathcal{E}}, \kappa_1^{\mathcal{E}}, C_1^{\mathcal{E}}, \tau_1^{\mathcal{E}} \rangle$ and $\Sigma_2^{\mathcal{E}} = \langle A_2^{\mathcal{E}}, \mathfrak{F}_2^{\mathcal{E}}, \kappa_2^{\mathcal{E}}, C_2^{\mathcal{E}}, \tau_2^{\mathcal{E}} \rangle$ be the affix higher-order signatures of $\mathcal{G}^{\mathcal{E}}$. We can construct an equivalent Datalog program $P(\mathcal{G}^{\mathcal{E}})$ by applying the transformation from Theorem 6.5 to $P(\mathcal{G}^{-\circ})$. This transformation uses the set of indexed predicates $\mathcal{I}$ defined as:

$$\mathcal{I} = \left\{ a^{(e_1,\ldots,e_k)} \;\middle|\; \begin{array}{c} a \in A_1^{\mathcal{E}}, \\ \kappa_1^{\mathcal{E}}(a) = \varepsilon_1 \Rrightarrow \cdots \varepsilon_k \Rrightarrow \mathsf{Type}, \\ k \geq 1 \\ e_1 \in \varepsilon_1, \ \ldots, \ e_k \in \varepsilon_k. \end{array} \right\}$$

The transformation's effect can be seen by comparing the original program in Example 6.1 with its transformed version in Example 6.3.

Example 6.3 ($P_{\mathcal{G}_{\mathsf{CFG}}}$ without number in predicate symbols).

$$\begin{aligned}
[\text{SENTENCE}_{sg}] \quad & \mathsf{S}(p_0, p_2) \leftarrow \mathsf{NP}(sg, p_0, p_1), \mathsf{VP}(sg, p_1, p_2). \\
[\text{SENTENCE}_{pl}] \quad & \mathsf{S}(p_0, p_2) \leftarrow \mathsf{NP}(pl, p_0, p_1), \mathsf{VP}(pl, p_1, p_2). \\
[\text{NOUN_PHRASE}_{sg}] \quad & \mathsf{NP}(sg, p_0, p_2) \leftarrow \mathsf{Det}(sg, p_0, p_1), \mathsf{N}(sg, p_1, p_2). \\
[\text{NOUN_PHRASE}_{pl}] \quad & \mathsf{NP}(pl, p_0, p_2) \leftarrow \mathsf{Det}(pl, p_0, p_1), \mathsf{N}(pl, p_1, p_2). \\
[\text{VERB_PHRASE}_{sg_sg}] \quad & \mathsf{VP}(sg, p_0, p_2) \leftarrow \mathsf{V}(sg, p_0, p_1), \mathsf{NP}(sg, p_1, p_2). \\
[\text{VERB_PHRASE}_{sg_pl}] \quad & \mathsf{VP}(sg, p_0, p_2) \leftarrow \mathsf{V}(sg, p_0, p_1), \mathsf{NP}(pl, p_1, p_2). \\
[\text{VERB_PHRASE}_{pl_sg}] \quad & \mathsf{VP}(pl, p_0, p_2) \leftarrow \mathsf{V}(pl, p_0, p_1), \mathsf{NP}(sg, p_1, p_2). \\
[\text{VERB_PHRASE}_{pl_pl}] \quad & \mathsf{VP}(pl, p_0, p_2) \leftarrow \mathsf{V}(pl, p_0, p_1), \mathsf{NP}(pl, p_1, p_2). \\
[\text{THE}_{sg}] \quad & \mathsf{Det}(sg, p_0, p_1) \leftarrow \mathit{the}(p_0, p_1). \\
[\text{THE}_{pl}] \quad & \mathsf{Det}(pl, p_0, p_1) \leftarrow \mathit{the}(p_0, p_1).
\end{aligned}$$

$$
\begin{aligned}
[\textsc{stakeholder}] \quad & \mathsf{N}(sg, p_0, p_1) \leftarrow \mathit{stakeholder}(p_0, p_1).\\
[\textsc{stakeholders}] \quad & \mathsf{N}(pl, p_0, p_1) \leftarrow \mathit{stakeholders}(p_0, p_1).\\
[\textsc{stock}] \quad & \mathsf{N}(sg, p_0, p_1) \leftarrow \mathit{stock}(p_0, p_1).\\
[\textsc{stocks}] \quad & \mathsf{N}(pl, p_0, p_1) \leftarrow \mathit{stocks}(p_0, p_1).\\
[\textsc{sell}] \quad & \mathsf{V}(sg, p_0, p_1) \leftarrow \mathit{sells}(p_0, p_1).\\
[\textsc{sells}] \quad & \mathsf{V}(pl, p_0, p_1) \leftarrow \mathit{sell}(p_0, p_1).
\end{aligned}
$$

$$
\begin{aligned}
\mathit{the}(0,1).\\
\mathit{stakeholder}(1,2).\\
\mathit{sells}(2,3).\\
\mathit{stocks}(3,4).
\end{aligned}
$$

Let $\Sigma_1 = \langle A_1, \mathfrak{F}_1, \kappa_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, \mathfrak{F}_2, \kappa_2, C_2, \tau_2 \rangle$ be the affix higher-order signatures of $\mathcal{G}$. The program $P(\mathcal{G}^{\mathcal{E}})$ can be further optimized by exploiting the structure of object constants resulting from dependent product elimination. These constants, indexed by feature structure combinations, function as extensional predicates in the Datalog program. Consequently, we can directly apply theorem 6.5 to $P(\mathcal{G}^{\mathcal{E}})$ to obtain the more efficient representation $P'(\mathcal{G}^{\mathcal{E}})$, using:

$$
\mathcal{I} = \left\{ c^{(e_1,\ldots,e_k)} \;\middle|\; \begin{array}{l} c \in C_2,\\ \tau_2(c) = \Pi x_1 : \mathcal{E}_1.\, \Pi \ldots . \Pi x_k : \mathcal{E}_k.\, \alpha,\\ \vdash_{\Sigma_2} \alpha :: \mathsf{Type},\\ k \geq 1,\\ e_1 \in \varepsilon_1,\ \ldots,\ e_k \in \varepsilon_k. \end{array} \right\}
$$

This transformation does not occur in this example as the object higher-order signature is a string higher-order signature.

Dependent Product Predicate Introduction

The aim is now to restore the factorization benefits that dependent products introduced in $\mathcal{G}$, using $P'(\mathcal{G}^{\mathcal{E}})$ as the starting point. Before proceeding to rule factorization, we must

complete one intermediate step. Let $c \in C_1$ be a constant with

$$\tau_1(c) = \Pi x_1 : \varepsilon_1 . \Pi \ldots . \Pi x_k : \varepsilon_k . \, p_1 \, \vec{f}_1 \multimap \cdots \multimap p_n \, \vec{f}_n \multimap p_0 \, \vec{f}_0$$

where $\vec{f}_i$ for $i \in [0, n]$ represents a sequence of feature structures.

Consider its expanded constants:

$$c_{e_1,\ldots,e_k} \in C_1^{\mathcal{E}} \quad \text{(for each } e_1 \in \varepsilon_1, \ldots, e_k \in \varepsilon_k)$$

with type

$$\tau_1^{\mathcal{E}}(c) = \tau_1(c)[x_1 := e_1, \ldots, x_k := e_k].$$

We transform each datalog rule $[c_{e_1,\ldots,e_k}]$ by prepending atoms $\Pi_{\varepsilon_i}(e_i)$ for $i \in \{1, \ldots, k\}$, where each e_i is treated as a Datalog constant. Additionally, for each enumeration $\varepsilon \in \mathcal{E}$ and each enumeral $e \in \varepsilon$, we add the corresponding degenerated rule:

$$[e] \quad \Pi_\varepsilon(e) \leftarrow .$$

We call the resulting program $P''(\mathcal{G}^{\mathcal{E}})$. For the rules associated to NOUN_PHRASE, the transformation produces:

$$[\text{NOUN_PHRASE}_{sg}] \quad \mathsf{NP}(sg, p_0, p_2) \leftarrow \Pi_{\mathtt{num}}(sg), \mathsf{Det}(sg, p_0, p_1), \mathsf{N}(sg, p_1, p_2).$$
$$[\text{NOUN_PHRASE}_{pl}] \quad \mathsf{NP}(pl, p_0, p_2) \leftarrow \Pi_{\mathtt{num}}(pl), \mathsf{Det}(pl, p_0, p_1), \mathsf{N}(pl, p_1, p_2).$$

Since the introduced Π atoms correspond to the dependent product structure and are always derivable, this transformation preserves the program's semantic equivalence.

Rule Factorization

The key insight enabling factorization is that all rules derived from a single abstract constant share a common structure. Consider an abstract constant c with dependent product type $\tau_1(c) = \Pi x_1 : \Phi_1 . \Pi \ldots . \Pi x_k : \Phi_k . \, p_1 \, \vec{f}_1 \multimap \cdots \multimap p_n \, \vec{f}_n \multimap p_0 \, \vec{f}_0$ in the original grammar $\mathcal{G}$. The expanded rules $\{c_{e_1,\ldots,e_k}\}_{e_1 \in \varepsilon_1, \ldots, e_k \in \varepsilon_k}$ in $P''(\mathcal{G}^{\mathcal{E}})$ differ only in their feature structure arguments, maintaining identical structure otherwise.

This structural consistency arises from three key properties:

1. The Π predicates remain identical across all instantiations because they reflect the type declaration $\tau_1(c)$, independent of specific feature values.

2. The intensional predicates preserve their type family structure under dependent product instantiation: if $\Pi x\colon \varepsilon.\, \alpha$ contains type families $A_1, \ldots, A_m$, then $\alpha[e/x]$ contains the same families for any $e \in \varepsilon$.

3. The extensional predicates maintain consistent structure because the original AACG is non-branching, ensuring that $\mathcal{L}(c_{e_1,\ldots,e_k})$ has identical form across all feature instantiations. More precisely, each $\mathcal{L}(c_{e_1,\ldots,e_k})$ yields the same pure term $\widehat{\mathcal{L}(c_{e_1,\ldots,e_k})}$.

To formalize the factorization process, we analyze the lexicon interpretation structure. Given fresh feature structure variables $\{z_1, \ldots, z_k\}$ relative to $\mathcal{L}(c)$, the non-branching property guarantees that $\mathcal{L}(c \bullet z_1 \bullet \ldots \bullet z_k)$ produces a term whose structure is independent of the specific variable values.

We compute the η-long β-normal form of $\mathcal{L}(c \bullet z_1 \bullet \ldots \bullet z_k)$, denoted as t, and extract its constant sequence $\overrightarrow{\text{cons}}(t) = (d_{1,\zeta_1}, \ldots, d_{m,\zeta_m})$, where each d_i is a constant and ζ_i represents its associated feature structure sequence. When d_i appears as $d_i \bullet f_1 \bullet \ldots \bullet f_n$ in the original term, we encode this as $d_{i,f_1,\ldots,f_n}$ with $\zeta_i = (f_1, \ldots, f_n)$ potentially containing variables $z_1, \ldots, z_k$.

By abstracting over the constants, we obtain the pure term $\hat{t}[d_{1,\zeta_1}, \ldots, d_{m,\zeta_m}]$ with principal type:

$$d_{1,\zeta_1} : \beta_1, \ldots, d_{m,\zeta_m} : \beta_m \vdash_{PT} \widehat{\mathcal{L}(c_{e_1,\ldots,e_k})}[d_{1,\zeta_1}, \ldots, d_{m,\zeta_m}] : \alpha_1 \multimap \cdots \multimap \alpha_n \multimap \alpha_0.$$

The factorized program $P(\mathcal{G})$ replaces each family of rules $\{c_{e_1,\ldots,e_k}\}_{e_1 \in \varepsilon_1, \ldots, e_k \in \varepsilon_k}$ in $P''(\mathcal{G}^{\mathcal{E}})$ with a single parameterized rule. Let $\sigma = [x_1 := z_1, \ldots, x_k := z_k]$ denote the substitution of bound variables with fresh variables.

Since the lexicon acts as an identity morphism on feature structures, we need to connect the abstract type level (intensional predicates) with the object term level (extensional predicates). To achieve this connection, we use the fresh variables $z_1, \ldots, z_k$ in the Π atoms instead of the variables $x_1, \ldots, x_k$ from the original dependent product type. The substitution σ applied to the feature structure sequences $\vec{f}_i$ ensures that the same feature variables z_i used in the Π predicates are consistently propagated through both the intensional predicates (from the abstract signature) and the extensional predicates (from the object signature). The factorized rule takes the form:

$$[c] \quad p_0(\vec{f}_0\sigma, \overline{\alpha_0}) \leftarrow \Pi_{\mathcal{E}_1}(z_1), \ldots, \Pi_{\mathcal{E}_k}(z_k), p_1(\vec{f}_1\sigma, \overline{\alpha_1}), \ldots, p_n(\vec{f}_n\sigma, \overline{\alpha_n}), d_1(\zeta_1, \overline{\beta_1}), \ldots, d_m(\zeta_m, \overline{\beta_m}).$$

The enumerals in the sequences $\vec{f}_i$ are treated as Datalog constants, while the feature variables are handled as Datalog variables. This transformation yields the factorized program in Example 6.4.

Example 6.4 (Factorized $P_{\mathcal{G}_{\mathsf{CFG}}}$).

$$
\begin{aligned}
[sg] \quad & \Pi_{\mathtt{num}}(sg) \leftarrow . \\
[pl] \quad & \Pi_{\mathtt{num}}(pl) \leftarrow . \\
[\textsc{sentence}] \quad & \mathsf{S}(p_0, p_2) \leftarrow \Pi_{\mathtt{num}}(x_1), \mathsf{NP}(x_1, p_0, p_1), \mathsf{VP}(x_1, p_1, p_2). \\
[\textsc{noun_phrase}] \quad & \mathsf{NP}(x_1, p_0, p_2) \leftarrow \Pi_{\mathtt{num}}(x_1), \mathsf{Det}(x_1, p_0, p_1), \mathsf{N}(x_1, p_1, p_2). \\
[\textsc{verb_phrase}] \quad & \mathsf{VP}(x_1, p_0, p_2) \leftarrow \Pi_{\mathtt{num}}(x_1), \Pi_{\mathtt{num}}(x_2), \mathsf{V}(x_1, p_0, p_1), \mathsf{NP}(x_2, p_1, p_2). \\
[\textsc{the}] \quad & \mathsf{Det}(x_1, p_0, p_1) \leftarrow \Pi_{\mathtt{num}}(x_1), \mathit{the}(p_0, p_1). \\
[\textsc{stakeholder}] \quad & \mathsf{N}(sg, p_0, p_1) \leftarrow \mathit{stakeholder}(p_0, p_1). \\
[\textsc{stakeholders}] \quad & \mathsf{N}(pl, p_0, p_1) \leftarrow \mathit{stakeholders}(p_0, p_1). \\
[\textsc{stock}] \quad & \mathsf{N}(sg, p_0, p_1) \leftarrow \mathit{stock}(p_0, p_1). \\
[\textsc{stocks}] \quad & \mathsf{N}(pl, p_0, p_1) \leftarrow \mathit{stocks}(p_0, p_1). \\
[\textsc{sell}] \quad & \mathsf{V}(sg, p_0, p_1) \leftarrow \mathit{sells}(p_0, p_1). \\
[\textsc{sells}] \quad & \mathsf{V}(pl, p_0, p_1) \leftarrow \mathit{sell}(p_0, p_1). \\
& \mathit{the}(0, 1). \\
& \mathit{stakeholder}(1, 2). \\
& \mathit{sells}(2, 3). \\
& \mathit{stocks}(3, 4).
\end{aligned}
$$

Theorem 6.6 (Semantic Preservation)**.** *The factorized program $P(\mathcal{G})$ and the enumerated program $P''(\mathcal{G}^{\mathcal{E}})$ are equivalent.*

Proof. The programs are equivalent because partial evaluation of $P(\mathcal{G})$ with only the Π facts produces exactly $P''(\mathcal{G}^{\mathcal{E}})$. When we restrict the factorized rules to specific enumeral values by applying the Π facts, each parameterized rule instantiates to the corresponding family of rules in $P''(\mathcal{G}^{\mathcal{E}})$, recovering the original expanded program structure. □

This factorized approach achieves the primary goal of feature-aware parsing: maintaining the compactness and efficiency benefits of feature structures while preserving polynomial-time complexity. The intensional treatment of Π predicates, combined with

parameterized rules, establishes a bijective correspondence between proof trees and abstract terms, ensuring that parsing results can be directly converted back to the original AACG representation.

While this transformation approach provides a concise proof of equivalence, a more direct implementation would modify Kanazawa's reduction to generate factorized programs $P(\mathcal{G})$ directly. This modification involves extending the arity of intensional and extensional schemas to accommodate feature structures, then applying this factorization methodology directly on the abstract constants with dependent products during rule generation rather than as a post-processing step.

6.4 Modelling Omissions

6.4.1 Omissions as Coercions

Having established how AACGs can be parsed, we now return to the case study (Example 5.1) to address a crucial detail we have not yet investigated: feature structure omission.

The grammar includes a rule where feature structures are entirely omitted:

$$\mathsf{Det} \to \textit{the}$$

A straightforward encoding to this feature structure system might look like this:

$$\mathsf{Det} :: \textsc{number} \Rrightarrow \mathsf{Type}$$
$$\textsc{the} : \Pi x : \textsc{number}.\, \mathsf{Det}\, x$$

With the following interpretations in the lexicon:

$$\mathsf{Det} \mapsto \Lambda x.\, \mathsf{String}$$
$$\textsc{the} \mapsto \lambda x.\, \textit{the}$$

However, this encoding introduces two problems. First, it employs dependent products for a task that deviates from their primary design, which is to enforce constraints across syntactic categories. Second, the feature parameter adds unnecessary complexity to the typing of THE since it plays no meaningful role in the lexicon's interpretation of THE. This makes the representation more complicated than necessary.

To address these issues, we introduce an alternative modelling using a dedicated type $\mathsf{Det}^{\perp}$ for determiners that omit feature specifications. This allows determiners like (THE) to be typed more naturally as $\mathsf{Det}^{\perp}$. To connect this feature-omitting type with the feature-aware type system, we employ *coercion* (Tannen, Gunter, and Scedrov 1990) through a dedicated constant. The coercion constant $!_{\mathsf{Det}^{\perp}\to\mathsf{Det}}$ (read as "coerce from $\mathsf{Det}^{\perp}$ to Det") allows terms of the feature-omitting type to be used wherever the feature-aware type is expected:

$$\begin{aligned}
\mathsf{Det} &:: \textsc{number} \Rrightarrow \mathsf{Type} \\
\mathsf{Det}^{\perp} &:: \mathsf{Type} \\
\textsc{the} &: \mathsf{Det}^{\perp} \\
!_{\mathsf{Det}^{\perp}\to\mathsf{Det}} &: \Pi x\colon \textsc{number}.\, \mathsf{Det}^{\perp} \multimap \mathsf{Det}\, x
\end{aligned}$$

The lexicon interpretations follow naturally from this type structure. The feature-omitting type $\mathsf{Det}^{\perp}$ maps directly to strings through String, while THE maps straightforwardly to *the*. The coercion operator $!_{\mathsf{Det}^{\perp}\to\mathsf{Det}}$ is implemented as an identity function that accepts an additional feature structure parameter but discards it:

$$\begin{aligned}
\mathsf{Det} &\mapsto \Lambda x.\, \mathsf{String} \\
\mathsf{Det}^{\perp} &\mapsto \mathsf{String} \\
\textsc{the} &\mapsto \mathit{the} \\
!_{\mathsf{Det}^{\perp}\to\mathsf{Det}} &\mapsto \lambda x.\, \lambda^{\circ} y.\, y
\end{aligned}$$

The lexicon mappings become more direct and transparent, better reflecting how determiners can omit feature specifications. While this approach might seem like a significant amount of formal machinery for a single lexical entry like THE, its true value emerges when applied to syntactic categories with numerous lexicalizations. For categories such as nouns or verbs, which contain a heterogeneous mix of entries (some that are morphologically variable and others that are not), this coercion mechanism offers a systematic and scalable solution. Without it, every lexical entry, including those that are feature-invariant, would need to be assigned a more complex type that accommodates unused feature parameters, leading to unnecessary clutter and reducing the grammar's clarity.

6.4.2 Type Hierarchies

This coercion approach may be generalized to create systematic type hierarchies from any type family kind. Consider a type a with kind $(\varepsilon_1 \Rrightarrow \ldots \Rrightarrow \varepsilon_m \Rrightarrow \mathsf{Type})$. We can construct a powerset lattice of its enumeration sets, as illustrated in Figure 6.3. Each node n_i in this lattice corresponds to a subset of enumerations and is associated with a type a_{n_i} whose kind reflects the enumerations present in that subset. The relationships between these types are formalized through coercion operators. For any pair of adjacent nodes n_i and n_{i+1} in the lattice where n_{i+1} extends n_i with one additional enumeration ε_{k+1}, we define a coercion operator:

$$!_{a_{n_i} \to a_{n_{i+1}}} : \Pi x_1 : \varepsilon_1 . \Pi \ldots . \Pi x_k : \varepsilon_k . \Pi x_{k+1} : \varepsilon_{k+1} . \\ a_{n_i}\ x_1\ \ldots\ x_k \multimap a_{n_{i+1}}\ x_1\ \ldots\ x_k\ x_{k+1}$$

with the following interpretation:

$$!_{a_{n_i} \to a_{n_{i+1}}} \mapsto \lambda^{\circ} x_1\ \ldots\ x_k\ x_{k+1} .\ \lambda^{\circ} y . y .$$

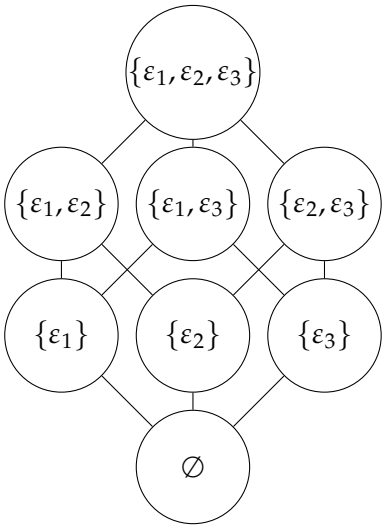


FIGURE 6.3: Example of Powerset lattice for three enumerations

While this presentation focused on enumeral omission, this construction generalizes

naturally to handle arbitrary finite lattices over type families, making it particularly suitable for modeling feature structure subtyping with finite subtypes.

However, there are theoretical limitations to consider when extending this approach. We deliberately chose not to incorporate automatic type hierarchy construction through feature structure subtyping into $\lambda^{-\circ\mathcal{F}\Pi}$. Although subtyping would offer an elegant mechanism for handling complex hierarchies, it fundamentally conflicts with a crucial requirement of this system: dependent product arguments must be drawn from finite domains. The conflict arises because *structural subtyping* on record types, where the relation $\Phi' <: \Phi$ denotes that Φ' is a subtype of Φ, introduces an infinite set of possible record terms through width and depth subtyping rules:

$$\frac{m \geq n}{[l_1 : \Phi_1, \ldots, l_m : \Phi_m] <: [l_1 : \Phi_1, \ldots, l_n : \Phi_n]} \textsc{Width}$$

$$\frac{\forall i \in \{1, \ldots, n\}, \Phi'_i <: \Phi_i}{[l_1 : \Phi'_1, \ldots, l_n : \Phi'_n] <: [l_1 : \Phi_1, \ldots, l_n : \Phi_n]} \textsc{Depth}$$

For example, let $\Phi_0 = []$ be the empty record type. We can construct an infinite sequence of types $(\Phi_n)_{n\geq 0}$ where $\Phi_{n+1} = [l : \Phi_n]$. By the depth subtyping rule, we have an infinite descending chain of subtypes:

$$\cdots <: \Phi_{n+1} <: \Phi_n <: \cdots <: \Phi_0.$$

This implies that the set of inhabitants of Φ_0 is infinite, as it includes the infinite set of terms $\{t_n \mid n \geq 0\}$, where $t_0 = []$ and $t_{n+1} = [l = t_n]$. This would violate the system's requirement for finite inhabitants of feature structure types, which is essential for maintaining the equivalence with core ACGs.

In contrast, a *nominal subtyping system* would meet this requirement. However, $\lambda^{-\circ\mathcal{F}\Pi}$, in its current form, does not include a mechanism for naming or declaring feature structure types; they are defined structurally. Introducing a nominal system would first require extending the calculus with a finite set of named feature structure type declarations. With such an extension, since subtype relations would be explicitly declared over this finite set of names, the resulting hierarchy and therefore the set of inhabitants for each type would remain finite.

The primary goal in designing $\lambda^{-\circ\mathcal{F}\Pi}$, however, was to keep its theoretical kernel as

lean and expressive as possible, without embedding complex features like a native subtyping system or a mechanism to name feature structure types. This minimalist design is not a limitation. Using a similar construction with coercions, we conjecture that an extension of $\lambda^{-\circ\mathcal{F}\Pi}$ with both named feature types and nominal subtyping would be no more expressive than one with only named feature types. This reasoning follows the construction of coercion semantics pioneered by Tannen, Gunter, and Scedrov (1990), suggesting that the calculus is sufficiently expressive to simulate these advanced features through systematic transformations rather than including them as primitives.

Chapter 7

Implementation and Evaluation

This chapter transitions from the theoretical development of Affix Abstract Categorial Grammars (AACGs) to their practical application and evaluation. The preceding chapters laid the groundwork by introducing $\lambda^{-\circ\mathcal{F}\Pi}$, a typed λ-calculus designed to model feature structures, and formally proved that AACGs, despite their enhanced expressiveness, are equivalent in generative power to core ACGs. The focus now shifts to the implementation of this system in an industrial compiler, built to address the practical challenges of grammar maintenance and performance in commercial NLG systems.

This chapter details the compiler's architecture and the syntactic enhancements that facilitate real-world grammar engineering. Recall that Yseop is the commercial NLG technology company that provided the industrial context for this CIFRE thesis. Subsequently, we present a quantitative analysis of the gains achieved, using Yseop's large-scale French grammar as a case study to compare grammar size and complexity. Finally, we report on performance benchmarks that measure text generation latency.

7.1 Compiler Overview

7.1.1 Description

Yseop specializes in generating financial and pharmaceutical reports from data in French and English, using ACGs instead of statistical approaches to ensure error-free results. However, the lack of feature structures in ACG has led to challenges such as poor parsing performance and high cognitive load for grammar maintainers. To overcome these issues, we have developed a concrete language and a proprietary compiler for AACGs.

The concrete language for Yseop's system builds upon $\lambda^{-\circ\mathcal{F}\Pi}$ and incorporates syntactic enhancements. We utilize ANTLR (T. J. Parr and Quong 1995), a parser generator that

constructs $LL(^*)$ parsers (T. Parr and Fisher 2011), to process this language. The resulting abstract syntax tree is then translated into a core language based on $\lambda^{-\circ\mathcal{E}\Pi}$.

Java (Gosling and McGilton 1995) was chosen as the development platform for the new compiler based on two compelling factors. The first is pragmatic: a significant portion of Yseop's existing backend infrastructure is implemented in Java, making it the natural choice for maintaining system compatibility. The second factor stems from Java's robust native support for parallelization and multi-threading capabilities, which simplifies the development and long-term maintenance of concurrent applications.

The compiler architecture employs a multi-stage pipeline design optimized for processing concurrent requests on AACGs. While the implementation primarily follows functional programming principles, it strategically allows controlled side effects in specific components where they provide measurable performance advantages.

$\lambda^{-\circ\mathcal{F}\Pi}$'s extensive use of binders in both terms and types necessitated careful consideration of their representation. Given that the industrial compiler must handle concurrent text generation requests with low latency requirements, the choice of representation significantly impacts system performance. We explored two potential approaches. The first was Higher-Order Abstract Syntax (HOAS, Pfenning and Elliott 1988), which elegantly represents object language bindings using meta-language variables and binders. While HOAS offers a natural and concise treatment of variable bindings and scope, performance testing revealed significant overhead in Java due to the cost of encoding anonymous functions, which would have been prohibitive for a production system handling thousands of concurrent requests. We ultimately adopted the locally nameless representation (Charguéraud 2012), which combines named variables for free variables with De Bruijn indices for bound variables (de Bruijn 1972). This hybrid approach provides both the reasoning clarity of named variables and the implementation efficiency of De Bruijn indices, making it suitable for high-throughput commercial applications.

7.1.2 Syntactic Sugar

The compiler incorporates several syntactic enhancements that simplify grammar definitions and improve readability. The first of these addresses record definitions, which can become cumbersome when managing complex hierarchies, as noted by Cardelli and Mitchell (1991). A more efficient strategy involves defining each record incrementally, based on the previous ones. To illustrate this contrast, Example 7.1 demonstrates a detailed, explicit approach where each record type is fully specified.

Example 7.1 (Explicit Agreement Hierarchy).

$$\begin{aligned}
\text{AGREEMENT} &= \begin{bmatrix}\texttt{num}: & \text{NUMBER}\end{bmatrix} \\
\text{FRAGREEMENT} &= \begin{bmatrix}\texttt{gend}: & \text{GENDER} \\ \texttt{num}: & \text{NUMBER}\end{bmatrix} \\
\text{FRARTAGREEMENT} &= \begin{bmatrix}\texttt{def}: & \text{DEFINITENESS} \\ \texttt{gend}: & \text{GENDER} \\ \texttt{num}: & \text{NUMBER}\end{bmatrix}
\end{aligned}$$

In contrast, Example 7.2 shows a more concise and modular approach, where each record type builds upon the previous one, adding only the new feature structures.

Example 7.2 (Incremental Agreement Hierarchy).

$$\begin{aligned}
\text{AGREEMENT} &= \begin{bmatrix}\texttt{num}: & \text{NUMBER}\end{bmatrix} \\
\text{FRAGREEMENT} &= \text{AGREEMENT} \leftarrow \texttt{gend} : \text{GENDER} \\
\text{FRARTAGREEMENT} &= \text{FRAGREEMENT} \leftarrow \texttt{def} : \text{DEFINITENESS}
\end{aligned}$$

In this incremental approach, the basic concept of agreement is initially defined in terms of number. The definition of French agreement (FRAGREEMENT) extends this by including gender. Finally, FRARTAGREEMENT further extends FRAGREEMENT by adding the feature structure for definiteness.

Similarly, record term alias can be created by reusing existing values, as demonstrated in Example 7.3.

Example 7.3 (Incremental Agreement Feature Structure Terms).

$$\begin{aligned}
agrSg &= \begin{bmatrix}\texttt{num} = & sg\end{bmatrix} \\
agrFSg &= agrSg \leftarrow \texttt{gend} = fem \\
agrDFSg &= agrFSg \leftarrow \texttt{def} = def
\end{aligned}$$

One other improvement we've made is the introduction of parameter declarations for top-level linear abstractions and type abstractions. This enhancement leads to a more organized representation of the parameters in interpretations. In the example below, the first line shows the lexicon definition of a NOUN_PHRASE, where the abstractions are explicitly defined. The second line, however, adds the parameter declaration and removes

the explicit linear abstractions:

$$\begin{aligned} &\text{NOUN_PHRASE} \mapsto \lambda^{\circ} det.\ \lambda^{\circ} n.\ det\ +\ n \\ &\qquad\qquad \Updownarrow \\ &\text{NOUN_PHRASE}\ det\ n \mapsto det\ +\ n \end{aligned}$$

Pattern matching naturally extends this approach when working with feature structures. This powerful programming construct allows grammar developers to define patterns that directly correspond to feature structures, eliminating the need for complex, error-prone nested case statements while enhancing both readability and maintainability.

Consider the following French example:

$$\begin{aligned} \text{CLIENT} \mapsto\ & \mathsf{case}_x\ \{ \\ & \quad sg \rightarrow \mathsf{case}_y\ \{ \\ & \qquad masc \rightarrow client\ | \\ & \qquad fem \rightarrow cliente \\ & \quad \}\ | \\ & \quad pl \rightarrow \mathsf{case}_y\ \{ \\ & \qquad masc \rightarrow clients\ | \\ & \qquad fem \rightarrow clientes \\ & \quad \} \\ & \} \end{aligned}$$

This example provides an interpretation for a lexical entry CLIENT using a nested case expression. In contrast, the example below decomposes this definition into four pattern maching equations, where each rule directly associates a specific combination of number and gender with its corresponding string representation:

$$\begin{aligned} \text{CLIENT}\ sg\ masc &\mapsto client \\ \text{CLIENT}\ sg\ fem &\mapsto cliente \\ \text{CLIENT}\ pl\ masc &\mapsto clients \\ \text{CLIENT}\ pl\ fem &\mapsto clientes \end{aligned}$$

Definition 7.1 (Pattern)**.** Let $\mathcal{V}$ be a countable set of variables. The set $\mathcal{P}$ of patterns is defined inductively by the following rules:

1. If e is an enumeral, then $e \in \mathcal{P}$.
2. If $x \in \mathcal{V}$, then $x \in \mathcal{P}$.
3. The symbol _ is a pattern (called a *wildcard*).
4. If $l_1, \ldots, l_n$ are distinct labels and $p_1, \ldots, p_n \in \mathcal{P}$ are patterns, then the record

$$\{l_1 : p_1, \ldots, l_n : p_n\}$$

   is in $\mathcal{P}$.

The wildcard _ is used when the specific value of a feature structure is irrelevant. It is syntactic sugar for a fresh unused variable.

The general form of pattern matching in lexicons can be represented as a series of equations, where each line associates a pattern with its corresponding term:

$$\begin{aligned} c\ p_{1,1}\ \ldots\ p_{1,m} &\mapsto t_1 \\ &\ldots \\ c\ p_{n,1}\ \ldots\ p_{n,m} &\mapsto t_n \end{aligned}$$

Here, c is a constant that appears in each equation, followed by a sequence of pattern $p_{i,j}$. Each pattern $c\ p_{i,1}\ \ldots\ p_{i,m}$ is mapped to its corresponding term t_i. When applying the lexicon, these patterns are matched against the input to determine which term should be produced.

Pattern matching in lexicons can be further enhanced by incorporating parameters for top-level linear abstractions. The general form for a constant c becomes:

$$\begin{aligned} c\ p_{1,1}\ \ldots\ p_{1,m}\ y_{1,m+1}\ \ldots\ y_{1,m+j} &\mapsto t_1 \\ &\ldots \\ c\ p_{n,1}\ \ldots\ p_{n,m}\ y_{n,m+1}\ \ldots\ y_{n,m+j} &\mapsto t_n \end{aligned}$$

Pattern matching in this system must satisfy two fundamental requirements: *non-overlapping patterns* (each input matches exactly one pattern) and *complete coverage* (every

possible input has a matching pattern). These requirements allows for a systematic translation in pure $\lambda^{-\circ\mathcal{F}\Pi}$.

The translation of pattern matching into these primitive constructs follows the formal approach detailed in Chapter 5 of Peyton Jones 1987. Notably, this translation process employs an intermediate representation that uses pattern matching abstractions.

This suggests an interesting theoretical possibility: $\lambda^{-\circ\mathcal{F}\Pi}$ could potentially be reformulated using pattern matching abstraction as its sole primitive construct for handling feature structures. Such a reformulation would eliminate the need for separate feature structure abstraction and case analysis constructors.

7.2 Derivation Recovery: Custom Prover vs. Standard Solver

During ACG parsing, the abstract term's structure is recovered from the corresponding Datalog proof tree. However, high-performance Datalog engines like Soufflé are solvers that return only the final set of proven facts, not the complete derivation forest needed to reconstruct proof trees. This creates an implementation choice: build a specialized Datalog engine with provenance tracking, or find a way to reconstruct derivations using standard solvers.

7.2.1 A Transformation-Based Solution for Standard Solvers

Provenance tracking requires recording not just which facts are derivable, but also the specific sequence of rule applications that led to each derivation. Standard solvers discard this information for efficiency.

We present a program transformation technique that preserves derivation history as explicit data within facts themselves. This approach can leverage existing optimized engines, integrates with standard Datalog optimizations, and avoids the synchronization overhead that would be required to maintain shared derivation structures during parallel bottom-up evaluation.

The transformation rewrites the original Datalog program P into a new program P' that explicitly tracks derivation information. The transformation creates a two-tiered representation where each intensional predicate $p \in \mathsf{IDB}(P)$ is replaced by two types of predicates in P':

1. *Evaluation predicates* $p^{\text{eval}}(\rho, \vec{t})$ that extend the original predicate $p(\vec{t})$ with a rule label ρ. These appear in rule bodies and represent facts with their derivation source.

2. *Provenance predicates* $p^{\text{prov},i,j}(\rho, \vec{t}, \vec{\rho_B}, \vec{v})$ that capture complete one-step derivations. Here:

   - i = number of intensional predicates in the original rule body
   - j = number of local variables in the original rule
   - $\vec{\rho_B}$ = vector of rule labels from antecedent facts
   - $\vec{v}$ = vector of local variable bindings

   These appear only in rule heads and store derivation trees.

Definition 7.2 (Partial Proof Tree)**.** A partial proof tree for a fact $p_0(\vec{t}_0)$ derived using rule ρ is a structure:

$$[\rho] \quad p_0(\vec{t}_0) \leftarrow \rho_1 : p_1(\vec{t}_1), \ldots, \rho_n : p_n(\vec{t}_n)$$

where each $p_i(\vec{t}_i)$ for $i \in [1, n]$ is a fact, and ρ_i is the label of the rule that was used to derive that fact.

The provenance predicates contain the necessary information to reconstruct this structure. While P' generates additional provenance facts, it contains precisely the information needed to reconstruct the original derivations.

Definition 7.3 (Predicate Schemas)**.** For each intensional predicate p of arity k in the original program, the transformed program will contain:

- An evaluation predicate: p^{eval} of arity $k + 1$.
- For each distinct structure of rules defining p (i.e., for each pair (i, j) representing i IDB body atoms and j local variables), a corresponding provenance predicate: $p^{\text{prov},i,j}$ of arity $1 + k + i + j$.

The transformation from P to P' proceeds in two stages.

1. **Rule Transformation:** For each rule ρ in P of the form:

   $$\rho : p_0(\vec{t}_0) \leftarrow p_1(\vec{t}_1), p_2(\vec{t}_2), \ldots, p_n(\vec{t}_n).$$

   Let $I = \{k : p_k \text{ is intensional}\} \subseteq \{1, \ldots, n\}$ and $E = \{k : p_k \text{ is extensional}\} \subseteq \{1, \ldots, n\}$. Let $i = |I|$ and $\vec{v}$ be the j local variables that appear in the body but not the head. We create a single transformed rule ρ' in P'. The head of ρ' is the specific

provenance predicate $p_0^{\text{prov},i,j}$. In the body, any intensional predicate p_k is replaced by its evaluation version p_k^{eval}, while any extensional predicate is left unchanged.

The transformed rule is:

$$\rho' : p_0^{\text{prov},i,j}(\rho, \vec{t}_0, \vec{\rho}_I, \vec{\nu}) \leftarrow \bigwedge_{k \in I} p_k^{\text{eval}}(\rho_k, \vec{t}_k), \bigwedge_{l \in E} p_l(\vec{t}_l).$$

In this rule, $\vec{\rho}_I = (\rho_k)_{k \in I}$ represents the vector of rule labels from the evaluation facts in the body, where the variables ρ_k for $k \in I$ bind to these labels.

2. **Projection Rules:** For each provenance predicate $p^{\text{prov},i,j}$ in the transformed program, add a single projection rule that converts its facts into evaluation facts for the corresponding predicate p.

$$\text{proj}_{p,i,j} : p^{\text{eval}}(\rho, \vec{t}) \leftarrow p^{\text{prov},i,j}(\rho, \vec{t}, \vec{\rho_B}, \vec{\nu}).$$

This rule strips the detailed provenance information (the vectors $\vec{\rho_B}$ and $\vec{\nu}$), passing only the fact's own rule label ρ and original arguments $\vec{t}$ forward.

Example 7.4 (Transformed Program with Provenance). Applying this transformation to the factorized Datalog program from Example 6.4, we obtain the following provenance-enhanced version. The original program contains intensional predicates $\Pi_{\texttt{num}}$, S, NP, VP, Det, N, V and extensional predicates *the*, *stakeholder*, *sells*, etc. Each rule is assigned a unique numerical label for clarity.

Rule Transformations

$$[1] \quad \Pi_{\texttt{num}}^{\text{prov},0,0}(1, sg) \leftarrow .$$

$$[2] \quad \Pi_{\texttt{num}}^{\text{prov},0,0}(2, pl) \leftarrow .$$

$$[3] \quad \mathsf{S}^{\text{prov},3,1}(3, p_0, p_2, \rho_\Pi, \rho_{NP}, \rho_{VP}, x_1) \leftarrow \Pi_{\texttt{num}}^{\text{eval}}(\rho_\Pi, x_1), \mathsf{NP}^{\text{eval}}(\rho_{NP}, x_1, p_0, p_1), \mathsf{VP}^{\text{eval}}(\rho_{VP}, x_1, p_1, p_2).$$

$$[4] \quad \mathsf{NP}^{\text{prov},3,0}(4, x_1, p_0, p_2, \rho_\Pi, \rho_{DET}, \rho_N) \leftarrow \Pi_{\texttt{num}}^{\text{eval}}(\rho_\Pi, x_1),$$

$$\begin{aligned}
&&&\mathsf{Det}^{\text{eval}}(\rho_{DET}, x_1, p_0, p_1),\\
&&&\mathsf{N}^{\text{eval}}(\rho_N, x_1, p_1, p_2).\\
[5]\quad &\mathsf{VP}^{\text{prov},4,1}(5, x_1, p_0, p_2, \rho_{\Pi 1}, \rho_{\Pi 2}, \rho_V, \rho_{NP}, x_2) \leftarrow &&\Pi^{\text{eval}}_{\texttt{num}}(\rho_{\Pi 1}, x_1),\\
&&&\Pi^{\text{eval}}_{\texttt{num}}(\rho_{\Pi 2}, x_2),\\
&&&\mathsf{V}^{\text{eval}}(\rho_V, x_1, p_0, p_1),\\
&&&\mathsf{NP}^{\text{eval}}(\rho_{NP}, x_2, p_1, p_2).\\
[6]\quad &\mathsf{Det}^{\text{prov},1,0}(6, x_1, p_0, p_1, \rho_\Pi) \leftarrow &&\Pi^{\text{eval}}_{\texttt{num}}(\rho_\Pi, x_1), \mathit{the}(p_0, p_1).\\
[7]\quad &\mathsf{N}^{\text{prov},0,0}(7, \mathit{sg}, p_0, p_1) \leftarrow &&\mathit{stakeholder}(p_0, p_1).\\
[8]\quad &\mathsf{N}^{\text{prov},0,0}(8, \mathit{pl}, p_0, p_1) \leftarrow &&\mathit{stakeholders}(p_0, p_1).\\
[9]\quad &\mathsf{N}^{\text{prov},0,0}(9, \mathit{sg}, p_0, p_1) \leftarrow &&\mathit{stock}(p_0, p_1).\\
[10]\quad &\mathsf{N}^{\text{prov},0,0}(10, \mathit{pl}, p_0, p_1) \leftarrow &&\mathit{stocks}(p_0, p_1).\\
[11]\quad &\mathsf{V}^{\text{prov},0,0}(11, \mathit{sg}, p_0, p_1) \leftarrow &&\mathit{sells}(p_0, p_1).\\
[12]\quad &\mathsf{V}^{\text{prov},0,0}(12, \mathit{pl}, p_0, p_1) \leftarrow &&\mathit{sell}(p_0, p_1).
\end{aligned}$$

Projection Rules

$$\begin{aligned}
[\text{projPi00}]\quad &\Pi^{\text{eval}}_{\texttt{num}}(\rho, x) \leftarrow \Pi^{\text{prov},0,0}_{\texttt{num}}(\rho, x).\\
[\text{projS31}]\quad &\mathsf{S}^{\text{eval}}(\rho, p_0, p_2) \leftarrow \mathsf{S}^{\text{prov},3,1}(\rho, p_0, p_2, \rho_{B1}, \rho_{B2}, \rho_{B3}, \nu_1).\\
[\text{projNP30}]\quad &\mathsf{NP}^{\text{eval}}(\rho, x_1, p_0, p_2) \leftarrow \mathsf{NP}^{\text{prov},3,0}(\rho, x_1, p_0, p_2, \rho_{B1}, \rho_{B2}, \rho_{B3}).\\
[\text{projVP41}]\quad &\mathsf{VP}^{\text{eval}}(\rho, x_1, p_0, p_2) \leftarrow \mathsf{VP}^{\text{prov},4,1}(\rho, x_1, p_0, p_2, \rho_{B1}, \rho_{B2}, \rho_{B3}, \rho_{B4}, \nu_1).\\
[\text{projDET10}]\quad &\mathsf{Det}^{\text{eval}}(\rho, x_1, p_0, p_1) \leftarrow \mathsf{Det}^{\text{prov},1,0}(\rho, x_1, p_0, p_1, \rho_{B1}).\\
[\text{projN00}]\quad &\mathsf{N}^{\text{eval}}(\rho, x, p_0, p_1) \leftarrow \mathsf{N}^{\text{prov},0,0}(\rho, x, p_0, p_1).\\
[\text{projV00}]\quad &\mathsf{V}^{\text{eval}}(\rho, x, p_0, p_1) \leftarrow \mathsf{V}^{\text{prov},0,0}(\rho, x, p_0, p_1).
\end{aligned}$$

7.2.2 Practical Implications and Implementation Decision

The transformed program produces provenance facts that enable reconstruction of complete derivation forests. Since provenance tracking precedes other optimizations, recovery simply requires filtering out optimization-generated facts.

However, this approach introduces significant performance trade-offs. The transformation increases predicate arities and generates larger fact sets, making seminaive evaluation more expensive. Combined with the overhead of bridging the Java compiler with C++ engines like Soufflé through native interfaces, these factors led to pursuing an alternative approach.

Instead, we have chosen to implement a prover using the parallel version of the improved seminaive algorithm. This strategy, based on the stratification method detailed in Chapter 2, first partitions intensional predicates into their respective strata based on dependencies in the precedence graph. A topological sort of these strata yields an optimal evaluation sequence where independent strata can be processed concurrently. This design is well-supported by the compiler's Java implementation, which was chosen specifically for its robust native support for parallelization and multi-threading.

Despite choosing a custom prover for production needs, the transformation-based approach remains valuable for research and experimentation. It enables direct performance comparisons across different Datalog engines and provides a cost-effective method for prototyping optimization techniques on specific grammars before committing to specialized prover development.

7.3 Experiments

7.3.1 French Grammar

To validate the practical effectiveness of AACGs, the language and compiler were tested using Yseop's French grammar, which is developed for data-to-text generation. This grammar builds upon two foundations:

1. An ACG encoding (Pogodalla 2017a) of a comprehensive French Tree-Adjoining Grammar (TAG, Abeillé 2002);

2. A syntax-semantics interface that leverages TAG derivation trees as an intermediate representation, following Pogodalla's architectural principles (Pogodalla 2017b).

The grammar's architecture implements these components through a system of three ACGs, as shown in Figure 7.1:

1. the grammar $\mathcal{G}_{\text{derived trees}}$ transforms derivation trees into derived trees;

2. the grammar $\mathcal{G}_{\text{yield}}$ converts derived trees into strings;

3. the grammar $\mathcal{G}_{\text{sem.}}$ connects derivation trees to the semantic representation

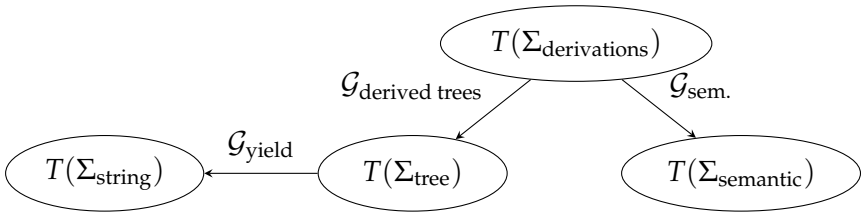


FIGURE 7.1: French Grammar Architecture

For the semantic layer, Yseop employ a minimalist representation we call *semantic graphs*. These are defined over a higher-order signature containing just a single atomic type for entities. Within this framework, relations, predicates, and modifiers are all encoded uniformly as functions over entities. This semantic design prioritizes computational efficiency in implementation over maximal expressiveness.

The French grammar implementation is not, strictly speaking, an ACG, but rather functions as a metagrammar that generates specialized ACGs based on the input semantic graph. This metagrammar selects appropriate lexicalizations that may be needed for a given input, and is modularized into several components, each handling specific grammatical constructions. This architectural choice (Salmon 2017) addresses a practical limitation: parsing with very large ACGs can be computationally intensive and slow. The metagrammar's output adheres to the architecture presented above, producing lexicalized ACGs that are optimized for the specific generation task at hand.

The original metagrammar, which we refer to as the *legacy system*, was implemented using two complementary technologies: C++ for high-performance implementation of core ACG algorithms, and YML (a proprietary object-oriented programming language) for constructing the lexicalized ACGs. To evaluate the impact of feature structure integration, we conducted a comprehensive assessment by manually rewriting Yseop's French grammar into the AACG language. This complete transformation enabled direct quantitative comparison between the implementations.

The results, shown in Figure 7.2, demonstrated large improvements in terms of grammar size. The legacy system's French grammar contained 51,246 unanchored derivation trees. After incorporating feature structures, this was reduced to 2,792 trees, an 18-fold reduction in size. The most significant reductions occurred in verbal, prepositional, and

nominal phrases, as well as in auxiliary elements like punctuation and linking words (categorized as *others*). For English, which has simpler morphology, we would expect more modest but still meaningful size reductions.

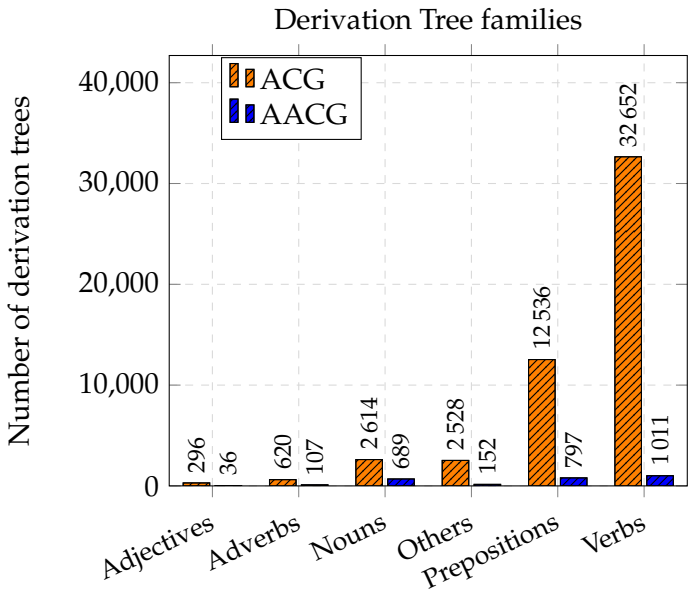


FIGURE 7.2: Comparison of the numbers of trees in the two derivation signatures

7.3.2 Text generation

The legacy system defined grammar lexicalizations through imperative programs, which made debugging challenging. To maintain a purely declarative approach, the AACG language was extended with meta constructs and modules. This extension enables direct comparison of both ACG lexicalization from semantic graphs and the performance of the resulting grammars. To this end, we designed two distinct scenarios for generating financial texts, each testing the grammars under different conditions. These scenarios were crafted to assess text generation from semantic graphs of varying complexity.

The first scenario, named *Describe Value*, involves generating textual content from a relatively simple semantic graph consisting of 6 nodes. This graph conveys a straightforward financial statement about a company named NewCo. Specifically, it captures the

information that *NewCo had recorded satisfactory revenues of $921,283.5.* An example of a sentence generated in this scenario could be: *NewCo a enregistré des revenus satisfaisants de 921 283,5 $*. The simplicity of the graph, with its minimal number of nodes and uncomplicated relationships, allows for a focused examination of the grammar's performance in generating basic financial descriptions. The aim here is to observe how quickly and accurately the system can transform this elementary semantic representation into a coherent text.

The second scenario, termed *Describe Variation*, features a more complex semantic graph with 15 nodes, providing a more detailed financial narrative. This graph includes historical data and trends, describing NewCo's gross margin in 2017 and its subsequent decline. It notes that in 2017, NewCo's gross margin was $1,463,000, which was considered unsatisfactory, and then details a 0.5% decrease in gross margin by 2018, indicating poor sales performance. An example of a sentence generated from this scenario might be: *En 2017, la marge brute de NewCo de 1 463 000 $ était insatisfaisante et a diminué de 0,5 % jusqu'en 2018, indiquant de mauvaises ventes*. Unlike the first scenario, this graph introduces temporal elements and comparative data, requiring the system to generate a more intricate text that not only presents the financial figures but also explains their significance in the context of NewCo's overall business performance. The challenge in this scenario lies in producing a narrative that captures both the quantitative data and the qualitative assessment of the company's financial health.

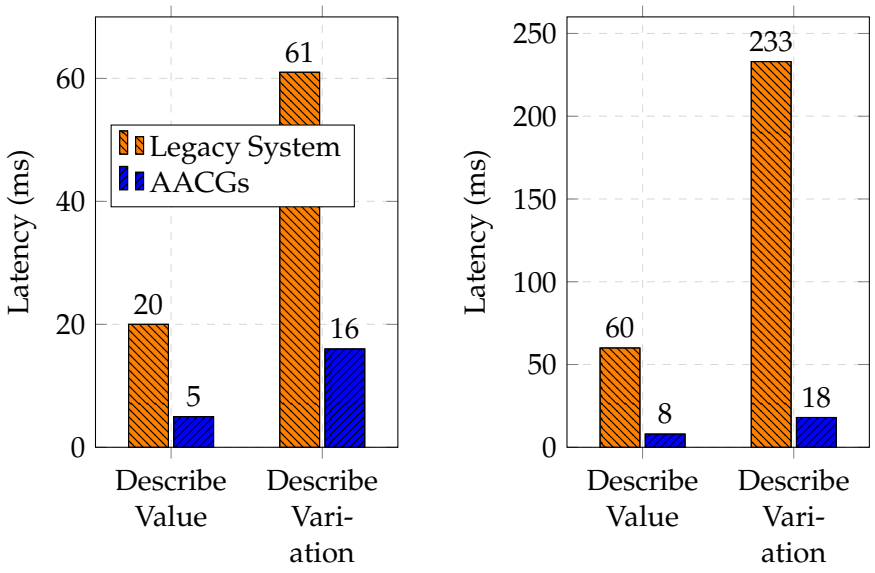
Text generation benchmarks (Figure 7.3a) demonstrated that French AACGs significantly outperformed the legacy system across both scenarios, with each scenario executed 30 times for statistical reliability. In the Describe Value scenario, AACGs reduced latency from 20 milliseconds to 5 milliseconds. The performance gains were even more dramatic in the Describe Variation scenario, where latency dropped from 61 milliseconds to 16 milliseconds.

The improved text generation performance stems from two key factors: reduced grammar size and algorithmic enhancements. Through grammar factorization, we significantly reduced complexity and computational overhead during ACG parsing. Additionally, while the legacy system used the basic seminaive algorithm for Datalog evaluation, the AACG language implements an enhanced version. These combined optimizations, compact grammar representation and improved evaluation algorithm, account for the latency reductions observed in both scenarios.

In the combined benchmark for grammar lexicalization and text generation (Figure 7.3b), similar improvements are observed. The *Describe Value* scenario shows a reduction in latency from 60 milliseconds in the legacy system to 8 milliseconds with AACGs. In the

more complex *Describe Variation* scenario, the latency is reduced from 233 milliseconds to 18 milliseconds.

The dramatic reduction in initialization latency stems from precompilation of the grammar's unanchored format, particularly through a key optimization: the construction of template Datalog rules. These templates serve as pre-built rules for each module, reducing initialization time which requires the construction of Datalog programs.



(A) Text generation

(B) Grammar lexicalization and text generation

FIGURE 7.3: Benchmarks

7.3.3 Parsing

To isolate the impact of grammar factorization on performance, we conducted an experiment using only the Datalog engine of the AACG system. We transformed the refactored French grammar into three versions: the original factorized grammar denoted as $\mathcal{G}$, a version with type families but no dependent products ($\mathcal{G}^{\mathcal{E}}$), and a core ACG representation

($\mathcal{G}^{-\circ}$). By comparing Datalog evaluation times across these configurations, we could assess the specific performance gains achieved through factorization, independent of other system optimizations.

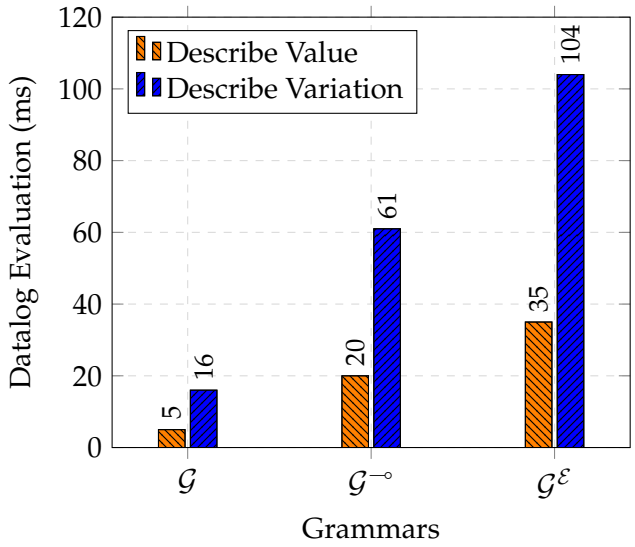


FIGURE 7.4: Performance comparison of Datalog evaluation across three equivalent grammar formulations in the Describe Value and Describe Variation scenarios.

The results in Figure 7.4 demonstrate the superior performance of the factorized grammar $\mathcal{G}$ over alternative representations across both *Describe Value* and *Describe Variation* scenarios. In contrast, the $\mathcal{G}^{\mathcal{E}}$ grammar shows significantly higher latency due to its lack of specialized predicates, which forces the Datalog engine to perform more extensive computations during bottom-up evaluation.

For programs generated by the Kanazawa reduction, the number of rules emerges as the primary performance factor. While reducing rule count improves evaluation efficiency, further optimization can be achieved through predicate specialization. This argues against approaches that rely on context-free backbones with subsequent filtering, as such approaches not only slow down program evaluation but also require additional type-checking of generated solutions. Instead, the results suggest an optimal strategy: combining rule minimization with targeted predicate specialization to maximize ACG parsing performance. Targeted predicate specialization involves creating distinct predicates for

specific feature structure combinations rather than using generic predicates with runtime filtering.

Conclusion

Contributions

This work advances both the theoretical foundations and practical applications of ACGs through three key contributions:

1. We developed $\lambda^{-\circ\mathcal{F}\Pi}$, a typed lambda calculus that uniquely combines linear typing with morphosyntactic feature structures through dependent products, enumerations for atomic features, and records for bundled attributes, while maintaining strong normalization and finite feature structure domains;

2. We introduced AACGs, which extend ACGs with feature structures while preserving their fundamental theoretical properties. Despite enhanced expressiveness for modeling morphological variation, we prove that AACGs are equivalent in generative power to core ACGs through a constructive three-stage transformation: dependent product elimination, record elimination, and feature structure expansion. This non-trivial equivalence result demonstrates that feature integration achieves exponential grammar compression without increasing computational complexity;

3. We validated our theoretical contributions through practical implementation. By integrating our approach with Yseop's infrastructure, we achieved an 18-fold reduction in French grammar size through feature structure factorization. This dramatic simplification yields two significant benefits: improved text generation performance and enhanced grammar maintainability.

Together, these contributions aim to partly address a critical challenge in symbolic text generation: reducing the substantial effort required to develop and maintain large-scale grammars.

Future Work

Future developments of AACGs can advance along three research directions: First, the expressiveness of feature structures could be enhanced by further extending the type system. A natural evolution would be generalizing enumerations into variants, enabling richer finite constructs that better capture linguistic phenomena. More ambitiously, we could develop a more abstract and flexible feature structure language by incorporating constraint systems. Drawing inspiration from Xi's work (Xi and Pfenning 1999) on refined types in ML using generic index languages with logical constraints, such an extension would enable more precise type specifications while maintaining compatibility with existing systems. This would allow grammarians to express complex linguistic constraints that are currently difficult to model with just variable sharing.

Second, while ACGs have demonstrated their ability to encode various grammar formalisms, developing formal encoding algorithms for unification grammars presents ongoing research opportunities. The key difficulty lies in handling reentrancy and cyclicity, which are common in linguistic descriptions but require careful theoretical treatment.

Third, the emergence of probabilistic extensions to ACGs (Ludmann, Pogodalla, and de Groote 2022) creates opportunities for modeling linguistic ambiguity. Integrating feature structures with probabilistic ACGs would yield a framework capable of simultaneously representing morphosyntactic constraints and language ambiguity.

Résumé Étendu

Introduction et Contexte

La génération de langage naturel (NLG) constitue un domaine spécialisé du traitement automatique des langues dont l'objectif est la production automatique de textes lisibles par l'humain. Complémentaire à la compréhension du langage naturel, la NLG représente le versant génératif du TAL, produisant des textes qui répondent aux standards humains de fluidité et de cohérence. Les systèmes de NLG se déclinent selon deux approches principales : la transformation de données structurées en récits cohérents (data-to-text), et la reformulation de textes existants sous de nouvelles formes (text-to-text).

Dans le contexte commercial, la NLG a acquis une traction significative dans divers secteurs industriels, avec des applications particulièrement marquantes en finance pour la génération automatique d'analyses de marché et de rapports financiers, ainsi qu'en santé pour la production de résumés cliniques et de documentation médicale. Les approches modernes vont des systèmes de génération par remplissage de patrons textuels aux architectures neuronales sophistiquées capables de générer des textes de plus en plus naturels et contextuellement appropriés. Néanmoins, ces systèmes font face à des défis complexes : le maintien de l'exactitude factuelle, l'assurance d'une cohérence logique à travers des passages plus longs, et l'adaptation du style linguistique à l'audience visée.

Cette thèse s'inscrit dans le contexte d'un système commercial de NLG utilisant les Grammaires Catégorielles Abstraites (ACGs, de Groote 2001), un formalisme originellement développé par l'équipe de recherche Sémagramme. Les ACGs fournissent une fondation mathématiquement rigoureuse pour la génération de textes grammaticalement corrects dans plusieurs langues.

Le formalisme ACG repose sur une fondation mathématique minimaliste, celle du λ-calculus linéaire simplement typé (Girard 1995). Cette base théorique confère au formalisme plusieurs propriétés intéressantes. Premièrement, les ACGs supportent à la fois les formes de surface et les structures de dérivation, offrant une représentation unifiée de

différents niveaux d'analyse linguistique. Deuxièmement, elles possèdent une réversibilité inhérente (Dymetman 1994), permettant leur utilisation tant pour l'analyse syntaxique que pour la génération de texte. Troisièmement, les ACGs peuvent exprimer des langages au-delà de la puissance des grammaires non-contextuelles (CFGs), tout en maintenant une tractabilité computationnelle.

À l'instar du Grammatical Framework (GF, Ranta 2004), l'ACG fonctionne comme un cadre grammatical où les grammaires sont des objets de première classe. Ce traitement permet la composition de grammaires, où des structures linguistiques complexes peuvent être construites à partir de composantes grammaticales plus simples. La compositionnalité des grammaires ACG fournit ainsi un cadre pour modéliser le langage à différents niveaux linguistiques, facilitant la structuration modulaire de systèmes grammaticaux complexes.

Défis et limitations

La gestion efficace des propriétés morphosyntaxiques (genre, nombre, accord, etc.) est cruciale pour générer des textes grammaticalement corrects. Malgré la puissance du formalisme ACG, l'absence de mécanismes intrinsèques pour gérer ces propriétés via des structures de traits constitue une limitation significative. Les structures de traits (Carpenter 1992) sont fondamentales en ingénierie grammaticale moderne, encodant ces propriétés à travers des mécanismes efficaces d'unification qui permettent une gestion systématique des contraintes dans l'analyse syntaxique et sémantique.

L'absence de support direct pour les structures de traits dans les ACGs nécessite le recours à des stratégies d'encodage alternatives qui accroissent la complexité de la modélisation linguistique. Pour illustrer cette limitation, considérons l'encodage d'une grammaire non-contextuelle augmentée de traits modélisant l'accord sujet-verbe en anglais. Une telle grammaire, exprimant naturellement les contraintes d'accord entre le sujet et le verbe via des variables de traits (par exemple, $\mathsf{S} \rightarrow \mathsf{NP}[NUM = ?n]\ \mathsf{VP}[NUM = ?n]$), doit être transformée en une grammaire non-contextuelle classique par un processus d'expansion des structures de traits.

Ce processus d'expansion crée des non-terminaux spécialisés pour chaque combinaison de traits, générant ainsi une prolifération de types et de constantes. Dans notre exemple d'accord simple avec deux valeurs de nombre (*sg*, *pl*), le non-terminal NP se dédouble en NP_{sg} et NP_{pl}, tandis que la règle de phrase se multiplie en deux variantes distinctes. Pour des systèmes linguistiques réels impliquant multiples traits à plusieurs

valeurs (genre, nombre, personne, cas, etc.), cette expansion conduit à une explosion combinatoire. L'encodage ACG résultant présente une prolifération de types, nécessitant des définitions de types séparées pour chaque variation de traits d'une catégorie syntaxique, ainsi qu'une multiplication de constantes, requérant des règles distinctes pour chaque combinaison de traits.

Dans les applications commerciales de NLG, cette limitation théorique introduit des considérations pratiques importantes pour l'implémentation et la maintenance des grammaires. L'expansion des structures de traits résulte en des grammaires de taille significativement accrue, rendant leur développement, maintenance et débogage considérablement plus difficiles. De plus, cette expansion impacte les performances des systèmes, tant en termes de temps d'analyse que d'empreinte mémoire.

L'objectif de cette recherche est par conséquent d'enrichir le formalisme ACG en y incorporant des structures de traits à travers un système de types étendu. Cette extension vise à maintenir les fondations mathématiques des ACGs tout en les rendant plus accessibles pour l'ingénierie grammaticale pratique.

Contributions

Face à la limitation identifiée, cette thèse apporte quatre contributions qui s'articulent autour de l'intégration de structures de traits dans le formalisme ACG, sur les plans théorique, algorithmique et pratique.

Le calcul $\lambda^{-\circ\mathcal{F}\Pi}$

La première contribution consiste en l'élaboration d'un λ-calculus typé, nommé $\lambda^{-\circ\mathcal{F}\Pi}$, qui combine de manière unique le typage linéaire avec les structures de traits morphosyntaxiques. Ce calcul constitue la fondation mathématique sur laquelle reposent les extensions ultérieures du formalisme ACG.

Le calcul $\lambda^{-\circ\mathcal{F}\Pi}$ étend le λ-calculus linéaire simplement typé par trois mécanismes fondamentaux destinés à modéliser les structures de traits :

Premièrement, les *énumérations* permettent de représenter les traits atomiques finis. Une énumération constitue un ensemble fini de symboles appelés énuméraux, offrant un moyen de définir des structures de traits atomiques par énumération explicite de leurs valeurs possibles. Par exemple, les traits morphologiques de base du français peuvent être modélisés comme $\text{GENDER} = \{masc \mid fem\}$ et $\text{NUMBER} = \{sg \mid pl\}$.

Deuxièmement, les *enregistrements* permettent de regrouper plusieurs traits en unités cohésives. Les descriptions linguistiques nécessitent fréquemment des structures complexes pour assembler des traits apparentés, traditionnellement représentées en linguistique comme des matrices attribut-valeur (AVMs). Un type d'enregistrement spécifie la structure de ces regroupements de traits en déclarant un ensemble de champs étiquetés, chacun associé à un type de structure de traits. Par exemple, le type $\text{AGREEMENT} = [\texttt{gend} : \text{GENDER}, \texttt{num} : \text{NUMBER}]$ capture les propriétés d'accord morphosyntaxique en français. Les termes de structure de traits peuvent alors être construits comme $[\texttt{gend} = \mathit{fem}, \texttt{num} = \mathit{sg}]$ ou $[\texttt{gend} = \mathit{masc}, \texttt{num} = \mathit{pl}]$, instanciant le schéma défini par le type d'accord.

Troisièmement, les *produits dépendants* (notés Π) servent de quantificateurs universels sur les structures de traits au sein des types. Bien que les familles de types (type families) permettent de paramétrer des types par des termes de structure de traits, elles ne peuvent à elles seules exprimer des contraintes où plusieurs constituants doivent partager les mêmes valeurs de traits. Les produits dépendants résolvent cette limitation en permettant la liaison de variables de traits à travers des types. Par exemple, la règle d'accord $\mathsf{S} \to \mathsf{NP}[NUM = ?n]\ \mathsf{VP}[NUM = ?n]$ peut être exprimée par le type $\Pi x : \text{NUMBER}.\ \mathsf{NP}\ x \multimap \mathsf{VP}\ x \multimap \mathsf{S}$, où le produit dépendant lie la variable x de type NUMBER dans les types $\mathsf{NP}\ x$ et $\mathsf{VP}\ x$, garantissant ainsi que le syntagme nominal et le syntagme verbal partagent la même valeur de nombre.

Le calcul $\lambda^{\multimap \mathcal{F} \Pi}$ préserve des propriétés théoriques cruciales du λ-calculus linéaire simplement typé, notamment la normalisation forte, assurant ainsi la terminaison de tous les calculs, et le caractère fini des domaines de structures de traits, garantissant la tractabilité computationnelle. Ces propriétés sont essentielles pour maintenir la viabilité algorithmique du formalisme lors de son application à l'analyse syntaxique et à la génération de texte.

Les Grammaires Catégorielles Abstraites à Affixes

La deuxième contribution de cette thèse consiste en l'introduction des Grammaires Catégorielles Abstraites à Affixes (AACG), une extension des ACGs qui intègre les structures de traits tout en préservant les propriétés théoriques fondamentales du formalisme. Le nom de cette extension s'inspire de la ressemblance avec une généralisation des grammaires à affixes (Koster 1991), qui étendaient originellement les grammaires non-contextuelles en introduisant des éléments paramétriques dans les non-terminaux pour imposer des contraintes d'accord à travers les règles grammaticales.

Les AACGs étendent les ACGs classiques en incorporant des structures de traits directement dans leurs signatures, permettant ainsi de définir des types paramétrés par des traits morphosyntaxiques. La conception des AACGs suit une séparation des préoccupations : le calcul $\lambda^{-\circ\mathcal{FI}\Pi}$ est responsable de la manipulation complexe des structures de traits, ce qui simplifie le rôle du lexique en celui d'un pont direct entre signatures via une correspondance identité sur les structures de traits.

Le résultat central de cette contribution établit l'équivalence forte entre AACGs et ACGs classiques. Plus précisément, nous démontrons que toute Affix ACG peut être transformée en une ACG équivalente, au sens où les deux grammaires génèrent les mêmes langages abstrait et objet via un isomorphisme de langages préservant la structure de dérivation. Cette équivalence est démontrée de manière constructive par une transformation qui élimine progressivement les constructions de structures de traits.

Cette transformation constructive démontre que, bien que les AACGs offrent une expressivité accrue pour modéliser les variations morphologiques, elles sont équivalentes en puissance générative aux ACGs classiques. Cette équivalence implique que l'intégration de structures de traits permet une compression grammaticale exponentielle sans augmentation de la complexité computationnelle.

Analyse syntaxique

La troisième contribution aborde les retombées algorithmiques de l'équivalence entre AACGs et ACGs pour l'analyse syntaxique. La complexité de l'analyse syntaxique d'une Affix ACG est déterminée par son ordre et par la complexité de sa traduction en ACG classique. En particulier, pour les AACGs du second ordre, une analyse syntaxique en temps polynomial est réalisable en les convertissant en ACG classique et en appliquant la réduction de Kanazawa (Kanazawa 2007), qui transforme l'analyse syntaxique en évaluation de programmes Datalog (un langage de requêtes logiques).

Bien que cette traduction directe démontre la tractabilité polynomiale, elle sacrifie les avantages de factorisation qu'offrent les structures de traits dans les AACGs. La représentation aplatie expanse explicitement toutes les variations de traits, résultant en des règles redondantes. Cette approche par énumération explicite devient de plus en plus inefficace à mesure que les traits grammaticaux se multiplient.

Pour préserver les bénéfices de factorisation des AACGs, nous proposons deux approches alternatives : le *squelette non-contextuel avec filtrage de traits*, qui sépare l'analyse en deux phases (analyse d'un squelette simplifié puis filtrage par unification de traits), et le *programme Datalog conscient des traits*, qui modélise directement les structures de traits

en Datalog en s'inspirant des Grammaires à Clauses Définies (DCGs, F. C. Pereira and D. H. Warren 1980), évitant ainsi l'explosion combinatoire de l'expansion des traits. Les deux approches préservent la complexité polynomiale.

Implémentation et validation

La quatrième et dernière contribution valide les développements théoriques par une implémentation pratique et une évaluation quantitative sur une grammaire à grande échelle. Un compilateur dédié aux AACGs a été conçu et développé en Java.

Le compilateur intègre plusieurs améliorations syntaxiques qui simplifient les définitions de grammaires et améliorent leur lisibilité, notamment une approche incrémentale pour la définition d'enregistrements permettant de construire des hiérarchies complexes de types de structures de traits de manière modulaire.

L'évaluation sur une grammaire française à grande échelle démontre l'efficacité de l'approche : la taille grammaticale est réduite d'un facteur 18 (de 51 246 à 2 792 arbres de dérivation), et la latence de génération de texte est divisée par un facteur d'environ 4 selon la complexité des scénarios testés. Ces résultats confirment que l'intégration de structures de traits offre des bénéfices substantiels en termes de maintenabilité grammaticale et de performances.

Conclusion et Perspectives

Cette thèse a enrichi le formalisme des Grammaires Catégorielles Abstraites en y intégrant les structures de traits morphosyntaxiques, répondant ainsi à une limitation significative pour l'ingénierie grammaticale pratique. Les contributions articulent développements théoriques, résultats formels et validation empirique : le calcul $\lambda^{\multimap,\mathcal{F}\Pi}$ fournit une fondation mathématique rigoureuse pour modéliser les structures de traits à travers énumérations, enregistrements et produits dépendants ; les Grammaires Catégorielles Abstraites à Affixes étendent les ACGs tout en préservant l'équivalence en puissance générative, démontrant qu'une compression grammaticale exponentielle est réalisable sans augmentation de complexité computationnelle ; enfin, l'implémentation industrielle et l'évaluation sur une grammaire française à grande échelle confirment les bénéfices pratiques substantiels en termes de maintenabilité grammaticale et de performances de génération.

Les développements futurs des AACGs peuvent s'articuler selon trois directions de recherche.

Premièrement, l'expressivité des structures de traits pourrait être enrichie par des extensions supplémentaires du système de types. Une évolution naturelle consisterait à généraliser les énumérations en variants, permettant des constructions finies plus riches capturant mieux certains phénomènes linguistiques. Plus ambitieusement, l'incorporation de systèmes de contraintes, inspirée des travaux de Xi (Xi and Pfenning 1999) sur les types raffinés en ML utilisant des langages d'index génériques avec contraintes logiques, permettrait des spécifications de types plus précises tout en maintenant la compatibilité avec les systèmes existants. Cette extension autoriserait l'expression de contraintes linguistiques complexes actuellement difficiles à modéliser avec le seul partage de variables.

Deuxièmement, bien que les ACGs aient démontré leur capacité à encoder divers formalismes grammaticaux, le développement d'algorithmes formels d'encodage pour les grammaires d'unification présente des opportunités de recherche continues. La difficulté principale réside dans le traitement de la réentrance et de la cyclicité, fréquentes dans les descriptions linguistiques mais nécessitant un traitement théorique attentif.

Troisièmement, l'émergence d'extensions probabilistes aux ACGs (Ludmann, Pogodalla, and de Groote 2022) crée des opportunités pour modéliser l'ambiguïté linguistique. Intégrer les structures de traits avec les ACGs probabilistes produirait un cadre capable de représenter simultanément les contraintes morphosyntaxiques et l'ambiguïté du langage, ouvrant ainsi de nouvelles perspectives pour la modélisation linguistique probabiliste dans le cadre des ACGs.

Bibliography

Abeillé, Anne (2002). *Une grammaire électronique du français*. Sciences du langage. CNRS Éditions. ISBN: 9782271058249.

Abiteboul, Serge, Richard Hull, and Victor Vianu (1995). *Foundations of databases*. Vol. 8. Addison-Wesley Reading. ISBN: 0201537710. URL: http://webdam.inria.fr/Alice/.

Aoto, Takahito and Hiroakira Ono (Jan. 1994). "Non-Uniqueness Of Normal Proofs for Minimal Formulas in Implication-Conjunction Fragment of BCK". In: *Bulletin of the Section of Logic* 3, pp. 104–112.

Babaev, A A and S V Solov'ev (Nov. 1982). "A coherence theorem for canonical morphisms in cartesian closed categories". en. In: *Journal of Soviet Mathematics* 20.4, pp. 2263–2279. DOI: 10.1007/BF01629434.

Beeri, C. and R. Ramakrishnan (1987). "On the power of magic". In: *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '87. San Diego, California, USA: Association for Computing Machinery, pp. 269–284. DOI: 10.1145/28659.28689.

Billot, Sylvie and Bernard Lang (June 1989). "The Structure of Shared Forests in Ambiguous Parsing". In: *27th Annual Meeting of the Association for Computational Linguistics*. Vancouver, British Columbia, Canada: Association for Computational Linguistics, pp. 143–151. DOI: 10.3115/981623.981641.

Blanvillain, Olivier et al. (Jan. 2022). "Type-level programming with match types". In: *Proc. ACM Program. Lang.* 6.POPL. DOI: 10.1145/3498698.

Bourreau, Pierre (June 2012). "Jeux de typage et analyse de lambda-grammaires non-contextuelles". Theses. Université Sciences et Technologies - Bordeaux I.

Cardelli, Luca and John C. Mitchell (1991). "Operations on records". In: *Mathematical Structures in Computer Science* 1.1, pp. 3–48. DOI: 10.1017/S0960129500000049.

Carpenter, Robert L. (1992). *The Logic of Typed Feature Structures: With Applications to Unification Grammars, Logic Programs and Constraint Resolution*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press. DOI: 10.1017/CBO9780511530098.

Ceri, Stefano, Georg Gottlob, and Letizia Tanca (1990a). "Extensions of Pure Datalog". In: *Logic Programming and Databases*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 208–245. DOI: 10.1007/978-3-642-83952-8_11.

— (1990b). *Logic Programming and Databases*. Surveys in computer science. Springer. ISBN: 3-540-51728-6. URL: https://www.worldcat.org/oclc/20595273.

Charguéraud, Arthur (2012). "The locally nameless representation". In: *Journal of automated reasoning* 49, pp. 363–408. DOI: 10.1007/s10817-011-9225-2.

Church, Alonzo (1936). "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58, p. 345. DOI: 10.2307/2371045.

Cooper, Robin (Apr. 2005). "Records and Record Types in Semantic Theory". In: *Journal of Logic and Computation* 15.2, pp. 99–112. DOI: 10.1093/logcom/exi004.

— (2008). "Type Theory with Records and Unification-based Grammar". In: *Logics for Linguistic Structures*. Ed. by Fritz Hamm and Stephan Kepser. Vol. 201. Trends in Linguistics. Studies and Monographs. Berlin; New York: De Gruyter Mouton, pp. 9–34. DOI: 10.1515/9783110211788.9.

Curry, H. B. (1930). "Grundlagen der Kombinatorischen Logik". In: *American Journal of Mathematics* 52.3, pp. 509–536. DOI: 10.2307/2370619.

Curry, Haskell B., Robert Feys, and William Craig (1959). *Combinatory Logic, Volume I*. Vol. 68. 4. Duke University Press, pp. 548–550. DOI: 10.2307/2182503.

de Bruijn, Nicolaas Govert (1972). "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes mathematicae (proceedings)*. Vol. 75. 5. Elsevier, pp. 381–392. URL: https://automath.win.tue.nl/archive/pdf/aut029.pdf.

de Groote, Philippe (2001). "Towards Abstract Categorial Grammars". In: *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*. ACL '01. Toulouse: Association for Computational Linguistics, pp. 252–259. DOI: 10.3115/1073012.1073045.

— (May 2002). "Tree-Adjoining Grammars as Abstract Categorial Grammars". In: *Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+6)*. Universitá di Venezia: Association for Computational Linguistics, pp. 145–150.

— (July 2015). "Abstract Categorial Parsing as Linear Logic Programming". In: *Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015)*. Ed. by Marco Kuhlmann, Makoto Kanazawa, and Gregory M. Kobele. Chicago, USA: Association for Computational Linguistics, pp. 15–25. DOI: 10.3115/v1/W15-2302.

de Groote, Philippe, Sarah Maarek, and Ryo Yoshinaka (2007). "On Two Extensions of Abstract Categorial Grammars". In: *Logic for Programming, Artificial Intelligence, and*

Reasoning. Ed. by Nachum Dershowitz and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 273–287. DOI: 10.1007/978-3-540-75560-9_21.

de Groote, Philippe and Sylvain Pogodalla (2004). "On the Expressive Power of Abstract Categorial Grammars: Representing Context-Free Formalisms". In: *Journal of Logic, Language, and Information* 13.4, pp. 421–438. DOI: 10.1007/s10849-004-2114-x.

Dymetman, Marc (1994). "Inherently Reversible Grammars". In: *Reversible Grammar in Natural Language Processing*. Ed. by Tomek Strzalkowski. Boston, MA: Springer US, pp. 33–57. DOI: 10.1007/978-1-4615-2722-0_2.

Earley, Jay (Feb. 1970). "An efficient context-free parsing algorithm". In: *Commun. ACM* 13.2, pp. 94–102. DOI: 10.1145/362007.362035.

Girard, Jean-Yves (1987). "Linear logic". In: *Theoretical Computer Science* 50.1, pp. 1–101. DOI: 10.1016/0304-3975(87)90045-4.

— (1995). "Linear Logic: its syntax and semantics". In: *Advances in Linear Logic*. Ed. by Jean-Yves Girard, Yves Lafont, and Laurent Regnier. London Mathematical Society Lecture Note Series. Cambridge University Press, pp. 1–42. DOI: 10.1017/CBO9780511629150.002.

Gosling, James and Henry McGilton (1995). *The Java Language Environment: A White Paper*. Tech. rep., p. 38. URL: https://stroustrup.com/1995_Java_whitepaper.pdf.

Gottlob, Georg and Christos Papadimitriou (2003). "On the complexity of single-rule datalog queries". In: *Information and Computation* 183.1. International Workshop on Implicit Computational Complexity (ICC'99), pp. 104–122. DOI: 10.1016/S0890-5401(03)00012-9.

Guillaume, Maxime, Sylvain Pogodalla, and Vincent Tourneur (May 2024). "ACGtk: A Toolkit for Developing and Running Abstract Categorial Grammars". In: *Functional and Logic Programming. 17th International Symposium, FLOPS 2024, Proceedings*. Ed. by Jeremy Gibbons and Dale Miller. Vol. Lecture Notes in Computer Science. Functional and Logic Programming. 17th International Symposium, FLOPS 2024 LNCS 14659. Kumamoto, Japan: Springer, pp. 13–30. DOI: 10.1007/978-981-97-2300-3_2.

Harper, Robert, Furio Honsell, and Gordon Plotkin (Jan. 1993). "A framework for defining logics". In: *Journal of the ACM* 40.1, pp. 143–184. DOI: 10.1145/138027.138060.

Hindley, J Roger (1969). "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146, pp. 29–60. DOI: 10.1090/S0002-9947-1969-0253905-6.

— (1997). *Basic simple type theory*. Cambridge Tracts in Theoretical Computer Science 42. Cambridge University Press. DOI: 10.1017/CBO9780511608865.

Hirokawa, Sachio (1991). "Principal type-schemes of BCI-lambda-terms". In: *Theoretical Aspects of Computer Software*. Ed. by Takayasu Ito and Albert R. Meyer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 633–650. DOI: 10.1007/3-540-54415-1_68.

Jordan, Herbert, Bernhard Scholz, and Pavle Subotić (2016). "Soufflé: On Synthesis of Program Analyzers". In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, pp. 422–430. DOI: 10.1007/978-3-319-41540-6_23.

Kanazawa, Makoto (2006). "Abstract families of abstract categorial languages". In: *Electronic Notes in Theoretical Computer Science* 165, pp. 65–80. DOI: 10.1016/j.entcs.2006.05.037.

— (June 2007). "Parsing and Generation as Datalog Queries". In: *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. Ed. by Annie Zaenen and Antal van den Bosch. Prague, Czech Republic: Association for Computational Linguistics, pp. 176–183.

— (2012). *Almost affine lambda terms*. Tech. rep. National Institute of Informatics. URL: https://www.nii.ac.jp/TechReports/public_html/12-003E.pdf.

— (2017). "Parsing and Generation as Datalog Query Evaluation". In: *IfCoLog Journal of Logics and their Applications* 4.4. URL: http://www.collegepublications.co.uk/downloads/ifcolog00013.pdf.

Kaplan, Ronald M. and Joan Bresnan (1982). "Lexical-Functional Grammar: A Formal System for Grammatical Representation". In: *The Mental Representation of Grammatical Relations*. Ed. by Joan Bresnan. MIT Press, pp. 173–281.

Koster, C. H. A. (1991). "Affix grammars for programming languages". In: *Attribute Grammars, Applications and Systems*. Ed. by Henk Alblas and Bořivoj Melichar. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 358–373. DOI: 10.1007/3-540-54572-7_13.

Lambek, Joachim (1958). "The Mathematics of Sentence Structure". In: *American Mathematical Monthly* 65, pp. 154–170. DOI: 10.1080/00029890.1958.11989160.

Ludmann, Pierre, Sylvain Pogodalla, and Philippe de Groote (Sept. 2022). "Multityped Abstract Categorial Grammars and Their Composition". In: *WoLLIC 2022 - 28th International Workshop on Logic, Language, Information, and Computation*. Vol. 13468. Lecture Notes in Computer Science. Iaşi, Romania: Springer International Publishing, pp. 105–122. DOI: 10.1007/978-3-031-15298-6_7.

Martin-Löf, Per and Giovanni Sambin (1984). *Intuitionistic type theory*. URL: https://oro.open.ac.uk/75650/.

Mints, GE (1979). "A coherence theorem for cartesian closed categories". In: *Journal of Symbolic Logic* 44.3, pp. 453–454.

Montague, Richard et al. (1970). "Universal grammar". In: *Theoria* 36 (3), pp. 222–46. DOI: 10.1111/j.1755-2567.1970.tb00434.x.

Nederpelt, R. P. (1973). "Strong Normalization in a Typed Lambda-calculus with Lambda-structured Types". PhD thesis. Eindhoven, Netherlands: Technische Hogeschool Eindhoven.

Parr, Terence and Kathleen Fisher (2011). "LL(*): the foundation of the ANTLR parser generator". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, pp. 425–436. DOI: 10.1145/1993498.1993548.

Parr, Terence John and Russell W. Quong (1995). "ANTLR: A Predicated- *LL(k)* Parser Generator". In: *Softw. Pract. Exp.* 25.7, pp. 789–810. DOI: 10.1002/SPE.4380250705.

Pereira, Fernando C. N. and David H. D. Warren (June 1983). "Parsing as Deduction". In: *21st Annual Meeting of the Association for Computational Linguistics*. Cambridge, Massachusetts, USA: Association for Computational Linguistics, pp. 137–144. DOI: 10.3115/981311.981338.

— (1980). "Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks". In: *Artificial intelligence* 13.3, pp. 231–278. DOI: 10.1016/0004-3702(80)90003-X.

Peyton Jones, Simon L (Mar. 1987). *The implementation of functional programming languages*. en. Prentice-Hall International series in computer science. London, England: Prentice-Hall. URL: https://simon.peytonjones.org/slpj-book-1987/.

Pfenning, Frank and Iliano Cervesato (July 1996). " A Linear Logical Framework ". In: *Logic in Computer Science, Symposium on*. Los Alamitos, CA, USA: IEEE Computer Society, p. 264. DOI: 10.1109/LICS.1996.561339.

Pfenning, Frank and Conal Elliott (1988). "Higher-order abstract syntax". In: *ACM sigplan notices*. PLDI '88 23.7, pp. 199–208. DOI: 10.1145/960116.54010.

Pogodalla, Sylvain (July 2016). "ACGTK: un outil de développement et de test pour les grammaires catégorielles abstraites". In: *Actes de la 23ème Conférence sur le Traitement Automatique des Langues Naturelles, 31ème Journées d'Études sur la Parole, 18ème Rencontre des Étudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (JEP-TALN-RECITAL 2016)*. Ed. by Laurence Danlos and Thierry Hamon. Démonstration. Paris, France: Association pour le Traitement Automatique des Langues, pp. 1–2.

— (2017a). "A syntax-semantics interface for Tree-Adjoining Grammars through Abstract Categorial Grammars". In: *Journal of Language Modelling* 5.3, pp. 527–605. DOI: 10.15398/jlm.v5i3.193.

Pogodalla, Sylvain (Sept. 2017b). "Abstract Categorial Grammars as a Model of the Syntax-Semantics Interface for TAG". In: *FSMNLP 2017 and TAG+13 conference*. Umeå, Sweden.

Pollard, C. and I.A. Sag (1994). *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. University of Chicago Press. ISBN: 9780226674476.

Pompigne, Florent (2013). "Modélisation logique de la langue et grammaires catégorielles abstraites". PhD thesis. URL: http://www.theses.fr/2013LORR0283.

Ranta, Aarne (Mar. 2004). "Grammatical Framework". In: *Journal of Functional Programming* 14.2, pp. 145–189. DOI: 10.1017/S0956796803004738.

Salmon, Raphael (July 10, 2017). "Natural Language Generation Using Abstract Categorial Grammars". PhD thesis. Sorbonne Paris Cité. URL: https://www.theses.fr/2017USPCC196.

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

— (2007). "On the membership problem for Non-linear Abstract Categorial Grammars". In: *New Directions in Type Theoretic Grammars*. Ireland: Reinhard Muskens, pp. 43–50.

Seldin, J. P. (1968). "Studies in Illative Combinatory Logic". PhD thesis. University of California.

Tait, William W (1967). "Intensional interpretations of functionals of finite type I". In: *The journal of symbolic logic* 32.2, pp. 198–212. DOI: 10.2307/2271658.

Tannen, Val, Carl A. Gunter, and Andre Scedrov (1990). "Computing with Coercions". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. Ed. by Gilles Kahn. ACM, pp. 44–60. DOI: 10.1145/91556.91590.

Ullman, Jeffrey D. (1988). *Principles of database and knowledge-base systems, Vol. I*. USA: Computer Science Press, Inc. DOI: 10.5555/42790.

— (1990). *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. USA: W. H. Freeman & Co. DOI: 10.5555/533142.

Vijay-Shanker, K. and A.K. Joshi (1988). "Feature Structures Based Tree Adjoining Grammars". In: *Coling Budapest 1988 Volume 2: International Conference on Computational Linguistics*. DOI: 10.3115/991719.991783.

Xi, Hongwei and Frank Pfenning (1999). "Dependent types in practical programming". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, pp. 214–227. DOI: 10.1145/292540.292560.

Yoshinaka, R (2006). "Linearization of affine abstract categorial grammars". In: *Proceedings of FG 2006: The 11th conference on Formal Grammar*. URL: https://web.stanford.edu/group/cslipublications/cslipublications/FG/2006/yoshinaka.pdf.

Zenger, Christoph (1997). "Indexed types". In: *Theoretical Computer Science* 187.1, pp. 147–165. DOI: 10.1016/S0304-3975(97)00062-5.