Université Sorbonne Paris Cité

Thèse de doctorat
de l'Université Sorbonne Paris Cité
Préparée à L'université Paris Diderot
Ecole Doctorale de Sciences Mathématiques de Paris Centre (ED 386)
Laboratoire de Linguistique Formelle

Natural language generation using abstract categorial grammars

Par Raphaël Salmon

Thèse de Doctorat d'Informatique

Dirigée par Laurence Danlos

Présentée et soutenue publiquement à Paris le 10/07/2017

Président du jury :	Pogodalla Sylvain, CR, LORIA
Rapporteurs :	Saint-Dizier Patrick, DR, IRIT
	Lapalme Guy, Professeur, Université de Montréal
Examinateurs :	Pogodalla Sylvain, CR, LORIA
	Meunier Frédéric, Consultant, SInequa
Directeur de thèse :	Laurence Danlos, Professeur, Université Paris Diderot

Title : **Natural language generation using abstract categorial grammars.**

Abstract : This thesis explores the usage of Abstract Categorial Grammars (ACG) for Natural Language Generation (NLG) in an industrial context. While NLG system based on linguistic theories have a long history, they are not prominent in industry, which, for the sake of simplicity and efficiency, usually prefer more "pragmatic" methods. This study shows that recent advances in computational linguistics allow to conciliate the requirements of soundness and efficiency, by using ACG to build the main elements of a production grade NLG framework (document planner and microplanner), with performance comparable to existing, less advanced methods used in industry.

Keywords : NLG, ACG, microplanning, document planning

Titre : **Génération automatique de texte avec des grammaires catégorielles abstraites.**

Résumé : Cette thèse explore l'usage des Grammaires Categorielles Abstraites (CGA) pour la Génération Automatique de Texte (GAT) dans un contexte industriel. Les systèmes GAT basés sur des théories linguistiques ont un long historique, cependant ils sont relativement peu utilisés en industrie, qui préfère les approches plus "pragmatiques", le plus souvent pour des raisons de simplicité et de performance. Cette étude montre que les avancées récentes en linguistique computationnelle permettent de conciler le besoin de rigeur théorique avec le besoin de performance, en utilisant CGA pour construire les principaux modules d'un système GAT de qualité industrielle ayant des performances comparables aux méthodes habituellement utilisées en industrie.

Mots clefs : GAT, GCA, planification de document, microplanning

Acknowledgements

I would like to thank the following people for their help, support and guidance while making this thesis:

- Laurence Danlos, for her unshakeable trust and her guidance throughout these four years.
- Patrick Saint-Dizier and Guy Lapalme for their helpful comments and suggestions for improving this work.
- Sylvain Pogodalla for being ever so kind to show interest in my work and putting so much time and effort in giving me his precious advices regarding the topic of my research.
- Alain Kaeser for his unshakeable support and trust as well as his ever present enthusiasm.
- Lorie Den Os for building the English grammar used in this work and being so kind and enthusiast while doing it.
- Yseop for financing this work and welcoming me into its team.
- Also the whole Yseop team, with all the people, too numerous to name here, with whom I've had the great pleasure to work with.
- And last but not least, the ANRT for its generous financial contribution, which made it all possible.

Contents

List of Figures

Chapter 1

Introduction

Since the beginning of the field in the 60s, artificial intelligence (AI) has known series of peaks of inflated expectations and troughs of disillusionment. The AI winter of the 90s, following the expert systems hype of the 80s, has fragmented the field into many, relatively independent subfields, which have tried since then to solve specific problems of perception, cognition or action separately. One of the main criticism against AI techniques at the time was that they didn't seem to be able to go into the real world, solving real world problems, but were confined to toy problems in the laboratory. It is only relatively recently that some of the subfields of AI born during this period have reached a level of performance which allows to solve real world problems quite convincingly and to use them in large scale industrial contexts. Probably the best example of this success is the field of deep learning techniques, which has solved several pending problems in image recognition, speech recognition and many others. But numerical techniques are not the only one which have have evolved, and more classical symbolic computation techniques also have their share of innovations. This thesis is about Natural Language Generation (NLG), one of the subfields of AI traditionally more inclined to symbolic approaches (especially in industrial contexts), and which have recently gained enough stability and maturity in order to be used in commercial applications.

1.1 What is NLG ?

1.1.1 NLG and NLU

In a perception/reflection/action cycle, natural language generation comes at the end. A NLG system takes as input an abstract representation of meaning, generally the product of some inference mechanism, and transforms it into text. NLG can be seen as the mirror

of Natural Language Understanding (NLU), which takes as input surface text and try to extract its meaning. However in practice, NLG and NLU systems often use quite different approaches. While machine learning techniques are now predominant in NLU, it is not yet the case in NLG, and many practical systems still use symbolic approaches. This difference between NLU and NLG can be explained by several factors:

- The input of NLG is not well defined[1]. We can produce text from raw numerical data, tables, databases, ontologies, etc.. Though formats for the representation of meaning do exist, we cannot always convert from one to another, and there is no definition for all possible concepts that a NLG system may receive.
- A consequence is that there is not a lot of data available and usable for training, as compared for instance to NLU.
- Moreover, pragmatic factors which are ignored in NLU play an important role in NLG. For instance, things like intention, media of communication, target of the communication, or even psychological modelling of the speaker.

All these factors make NLG a very difficult task, and while attempts at building systems which perform both text understanding and text generation have been made, the two subjects are often considered as radically different matters (McDonald, 1986).

1.1.2 Choice management

The problem of NLG can be represented as a set of interdependent *choices*. For each piece of meaning in the input, we must choose a text chunk which corresponds to this meaning (if any), and assemble it with other text chunks in a coherent manner. The choices are dependent on several external, pragmatic factors, but they are also made interdependent by the rules of language: syntax, semantics and rhetorics.

The most basic way to handle all these constraints is to use *conditional templates*, which allows to choose different textualisations for an input depending on any kind of condition. While this method may seem simplistic, it has its importance in practice. It can be used in pretty much every existing programming language and can be used by anyone with minimal knowledge in these languages. Therefore it is probably the method of choice for any NLG application which is simple enough, and this is the case of a non negligible part of real world applications. However conditional templates quickly meet their limits when the number of choices and their interdependencies grows.

[1]The input of a particular NLG system must be well defined, but there is no general consensus as to what representation of meaning should be used, and different NLG system often use different approaches.

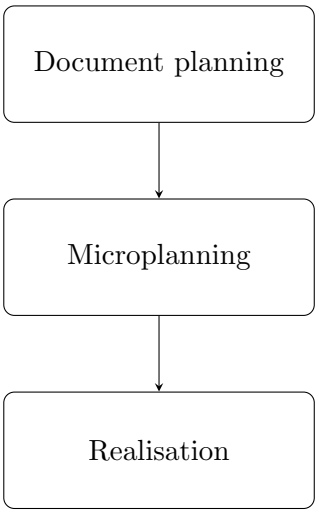


FIGURE 1.1: The three stages of natural language generation (Reiter and Dale, 2000). First the document planning phase creates a document plan from a data source. Then the microplanning phase transforms this document plan into a text structure. Finally, the realisation phase produces the surface text and embeds it into a document or a web page.

The usual strategy when the problem of NLG becomes more complex, is to limit the number of dependencies between the different choices by isolating them in independent modules. All the difficulty here is to regroup the choices in such a way that all known important dependencies are handled, while limiting the number of dependencies. There are a lot of different ways of doing this, which are not always compatible. However an usually accepted modularization is the one by (Reiter, 1994, Reiter and Dale, 2000), which distinguishes between a *document planner*, a *microplanner* and a *realiser* module, organized into a pipeline (see Figure 1.1). The document planner regroups choices about the conceptual representation of the problem, the microplanner about the syntactic representation of the output and the realiser about the format of the output text. I use these distinctions in Chapters 3 and 4 in order to present and compare different NLG techniques.

1.2 Context and goal of the thesis

1.2.1 The context

This thesis has been funded by Yseop[2], one of the few companies around the world specialized in NLG[3]. Companies specialized in natural language generation must cope

[2] www.yseop.com, the thesis was co-financed by the "Association Nationale de la Recherche et de la Technologie" (ANRT).

[3] Other companies include (but are not limited to) Narrative Science (www.narrativescience.com), Arria (www.arria.com) and Automated Insight (www.automatedinsights.com).

with very different scenarios, and build NLG systems for different clients as quickly as possible. These applications may be built in a few days by a single person with no specific training in programming or linguistics for the simplest cases, or by a small team of developers and NLG/Data Analysis experts in a few months for the more complex ones. Therefore there are strong requirements on the technology to be both simple to use in simple cases and powerful enough to handle complex scenarios.

Since a single system cannot cope efficiently with all potential scenarios, the technology at the core of a NLG company takes the form of a framework, which offers predefined behaviours, architectures, best practices, specific functions, perhaps a specific software environment to easily develop NLG applications. In fact, many things can be thought as being part of a NLG framework, and we may differentiate between purely technical tools, from the environment surrounding these technical tools. By environment, I mean for instance:

- A specialized IDE (Integrated Development Environment) with helpful functions or data for NLG, like grammars, database access and data filtering tools, document visualization, spell checkers, multilingual displays, etc..
- A deployment solution for accessing the NLG system from the internet, perhaps specialized protocols or web services, backup facilities, cloud deployment and load balancing facilities, etc..
- Documentation, teaching materials, tutorials, bootstrap applications, support service, etc..
- Project management methods, procedures for specifying the solution, extracting the knowledge from experts of the domain, etc.

The environment is heavily dependent on the core technology, and at least as important in order to successfully and efficiently build NLG applications. Therefore it is indispensable that the core technology be well designed in order to integrate smoothly in its environment and allow for powerful extensions and easy to use tools.

1.2.2 The goal

The main goal of this thesis is to *extend the core technology* of Yseop, in order to be able to handle complex problems more easily. In other words, the goal is to build a NLG framework which copes with known NLG related problems and integrates smoothly into an existing technology. Reflecting on the rich environment surrounding the Yseop

technology, the details of this goal are a mix of theoretical and pragmatic concerns, which could be summarized as follows:

- The new system should be able to perform document planning and microplanning tasks. The Yseop technology already provides several high level functionalities, like a realiser module, referring expressions generation and a few other microplanning functionalities, however it lacks dedicated functions for some tasks usually associated with microplanning or document planning (see Chapters 3 and 4).
- It should be able to use existing resources, in particular linguistic ones, like dictionaries or grammars.
- It should be integrated in the existing technology, in way which allows to access simpler functionalities when the added complexity is not needed.
- It should be fast enough to be used in a production environment. To give an order of magnitude, the time allowed for text generation for on-demand applications usually varies between a few milliseconds and a few hundreds of milliseconds.
- The system should be as simple to use and configure as possible.
- It should be flexible and easy to maintain over a long period of time.

This work has been the occasion to reflect on the architecture of the overall technology, and to propose new means of achieving the desirable properties of an ideal NLG system.

1.3 Main results

The proposition developped in this thesis is to use the formalism of Abstract Categorial Grammars (ACG, De Groote, 2001) as a kernel system for natural language generation. ACG is an abstract formalism which allows to represent different other formalisms. Using it as a kernel means that we can develop different tools for natural language generation using different formalisms, and use ACG as *a low-level, canonical way to represent and run* these tools. This allows to focus on computational issues in the kernel, and on pragmatic or ergonomic ones in the configuration of the system (i.e. the tools built on top of the kernel).

To show the soundness of this approach, a prototype system has been built directly in the core technology of Yseop. This prototype system has been evaluated on microplanning and document planning tasks, in order to confirm that it satisfies the efficiency and ease of use standards of an industrial NLG technology. This evaluation has been globally

successful and it has shown that the prototype meets most of the initial goals of this thesis, and can be turned into a production system.

From an academic perspective, the main contributions of this thesis are: an analysis of the usage of ACG in the context of a NLG framework, and more specifically for performing the document planning task (Chapter 7), an implementation of ACG in an industrial context (Chapter 8) and feedbacks on the usage of ACG as a NLG framework in an industrial context(Chapter 9).

1.4 Plan

The first part of the thesis is dedicated to the state of the art in natural language generation. As there are many different techniques, I focus in priority on methods related to the tools implemented. Chapter 2 gives an overview of the different approaches to NLG from a very high-level perspective. Chapter 3 presents the most common methods for document planning and Chapter 4 the ones for microplanning. Finally, in Chapter 5, I elaborate on the reasons to use ACG as a kernel system for NLG and compare this approach with others.

In the second part of the thesis, I give a detailed presentation of ACG, the prototype implementation and the developed tools. Chapter 6 gives the definitions for ACG. Chapter 7 details how ACG can be used in the context of a NLG framework and develops on the usage of ACG for document planning. Chapter 8 is a precise description of the implementation of ACG into the Yseop technology, and Chapter 9 presents the evaluation of the framework and concludes on the usage of ACG for natural language generation and future work.

Part I

Natural Language Generation

Chapter 2

Natural language generation system architectures

The field of NLG is at the crossways of several disciplines, which have influenced the general architecture and details of many different NLG systems. In this chapter, I survey four major influences in the design of NLG systems: software engineering, linguistics, combinatorial optimization and machine learning.

In this chapter and the following ones, I use as a running example a car selling application, which is one of the applications which have been used to test the technology in a realistic environment (see Chapter 9). The context of this application is a recommender system for an online car selling application. The user fills a form with information about its needs and situation, and the system proposes ten cars which might correspond to its needs. The text generation system writes the description of each car, using the characteristics of the car and a user model built from the input the user.

2.1 Software engineering

The general problem of NLG is very complex, and generally involves a lot of different structures and constraints. Organizing them in a coherent and efficient way is a challenge in itself. Propositions of architectures for NLG systems are often based on software engineering concepts, like extendibility, reusability, efficiency, etc., and present particular modularizations which support these properties. A popular example of such architecture is the task-based pipeline architecture from (Reiter and Dale, 2000), which focuses on pragmatic concerns and has already been successfully used in an industrial context

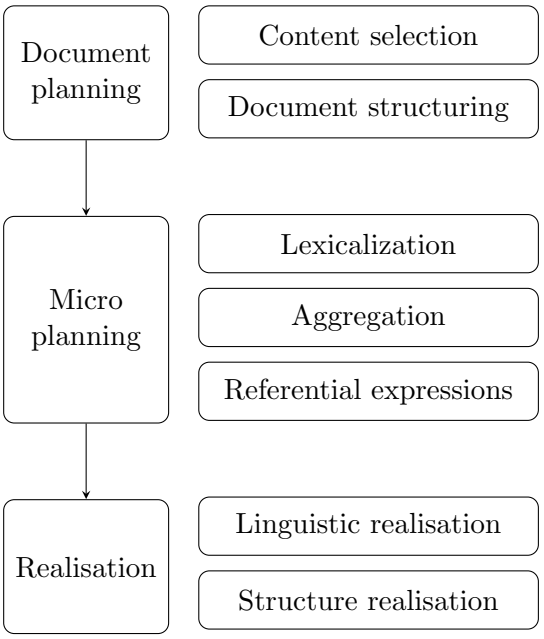


FIGURE 2.1: The detailed task-based pipeline architecture. The tasks of document planning are content selection and document structuring. The microplanning phase does lexicalisation, aggregation and referential expressions generation. Realisation can be divided into linguistic realisation and structure realisation.

(Reiter, 2015)[1]. I also present in this section the RAGS project (Mellish et al., 2004), another proposition of architecture which focuses on software quality and normalisation of the NLG problem in a somewhat different manner than (Reiter and Dale, 2000).

2.1.1 The task-based pipeline architecture

Document planning, microplanning and realisation can be refined by defining sub-modules, which encode additional independence assumptions (Reiter and Dale, 2000). Figure 2.1 shows the detailed task-based pipeline architecture. Document planning is divided into a content selection and a document structuring module. Microplanning is divided into three sub-tasks: lexicalisation, aggregation and referring expressions generation. Finally, realisation can be divided into a linguistic realisation and structure realisation module.

2.1.1.1 Content selection

The content selection module takes upon itself to build messages (i.e. representations of meanings) from an input data source. Since it needs to connect to very different

[1]The industrial version uses the JAVA programming language and a more precise description of the algorithms for each module. However most of the details of this implementation are private.

data sources, it is probably the most ill defined part of the NLG pipeline. Its main concerns are segmentation of the messages, correction and inference on the input data and filtering. Content selection is potentially a very complex problem in itself, and is sometimes treated as a whole system with its own complex architecture, either as an extension of the task-based pipeline architecture (Reiter, 2007), or as an external system.

2.1.1.2 Document structuring

The document structuring module builds a document plan from a set of messages produced by the content selection module. The simplest case of document structuring is to simply order the messages based on a communication goal. However it may involve more complex rhetorical reasoning and tree, or graph structures, as we will see in Section 3.2.

2.1.1.3 Lexicalisation

The lexicalisation module maps messages to their textualisation. The textualisation of a message is often a syntactic structure, which associate syntactic information with words and group them in syntactic constituents, like noun phrases, or prepositional phrases. The choice of syntactic construction may be dependent on external factors, as the desired level of detail, or some constraints on the vocabulary.

2.1.1.4 Aggregation

The goal of the aggregation module is to set the limits of sentences and paragraphs and to pack information in order to avoid redundancies. For instance, if we have two messages which can be textualised as:

The car has a GPS.
The car has an automatic transmission.

The aggregation module can take advantage of the fact that both messages talk about the same car to pack the information into a single sentence:

The car has a GPS and an automatic transmission.

2.1.1.5 Referring expression generation

The referring expression generation module chooses the expressions which refer to entities. Entities may be persons, objects or even abstract concepts. The problem of referring to entities comes in two parts. First the entity must be introduced, the first time it is referred to in the discourse. Then, if multiple references are made to the same entity, the system should produce short expressions which unambiguously refer to our entity. These expression may be noun phrase like "*the car*", or pronouns, like "*it*" or "*she*". To produce the referring expressions, the module uses the discourse history and some notion of salience, in order to compute what is ambiguous and what is not in the current context.

2.1.1.6 Linguistic and structure realisation

It is possible to separate the realization module between a linguistic and a structure realisation module. The linguistic realisation module makes all the choices which impact only the text. The structure realisation module makes all the choices which involve all other aspects of the document, namely formatting and specific document formats.

2.1.1.7 Advantages and limits

A good modular design is characterized by few, easily recognizable modules, connected by interfaces, and there should be as few, weakly coupling, interfaces as possible. The task-based pipeline fits this description relatively well. Each module can be described rather succinctly, by the task it performs, and pipeline architectures minimize the number of interfaces between modules. The identified tasks all take their root in practical applications and the architecture focuses on solving practical problems encountered during the creation of NLG systems.

This pragmatic approach has been both a factor of success and a source of criticisms for the task-based pipeline architecture. A common argument against this architecture is that it makes independence assumptions which are too strong, and imposes a specific order on decisions which should be done jointly (Appelt, 1985, Danlos and Namer, 1988, Mellish et al., 2004). A related observation is that some of the identified tasks may be distributed over several levels of abstraction. For instance, aggregation and sentence boundary delimitation may take place at a conceptual level or at a syntactic level. These remarks, as well as the fact that the modules of the task-based architecture are defined informally and are sometimes hard to compare with other approaches, have been the

motivation for another proposition for a reference architecture for generation systems: the RAGS project (Mellish et al., 2004).

The prototype we developed is closer in spirit to RAGS than the task-based pipeline architecture (see Chapter 5 for a comparison between the three approaches). However, since the task-based pipeline architecture has also a descriptive function ans has been used as such extensively, I still use it to introduce the different techniques in Chapters 3 and 4.

2.1.2 A Reference Architecture for Generation Systems (RAGS)

The RAGS project (Mellish et al., 2004), proposes an architecture whose purpose is to compare existing systems and serve as a basis for the development of new NLG systems within a common framework, with the possibility of sharing modules between researchers and use some predefined components or data structures. Unlike the task-based pipeline architecture, the RAGS central elements are the data structures, the processes or tasks being left undefined (although some example functions are proposed). There are two levels of data structures in RAGS:

- High level data types, describing the main linguistic abstraction levels and usual data structures found in NLG systems. The proposed high-level structures are: conceptual, rhetorical, document, semantic, syntactic and quote (string) structures. Each level is described formally by an abstract type definition.
- A low-level structure, defined as a typed directed graph, which can be used to represent the high-level data types. This low-level structure is the concrete information transmitted through the interfaces of the system. RAGS also provides an XML specification for the low-level structures which allows to persist and transmit data.

Apart from these definitions, the rest of the architecture is relatively unconstrained and many scenarios are possible. One can create its own high-level data types as long as they can be represented in the low-level structure. Modules can be arranged in any order and perform any function (the goal still being to go from a conceptual representation to a string representation). The formalisation of the different data structures and the underlying directed graph representation allows to compose an architecture using different existing systems, by only coding the glue code for the interface, and to use different programming languages for different parts of an application (again with glue code). The

RAGS project still proposes a default implementation, based on a blackboard, event-driven architecture, which can be used to simulate several other architectures (including pipeline architectures).

The architecture of the RAGS project is very flexible. In particular, the formalisation of a low-level structure shared between all modules of the system makes it particularly suited for extensions and reusable parts management. It also has a great expressive power and can be used to analyse most of the existing NLG systems. However this expressivity also makes it complex. The authors voluntarily opened the architecture as much as possible, in order to leave room for new architectures and theories. While this is precious in a research environment, leaving the architecture too open may be problematic in industrial contexts, because of the complexity it requires and the potential performance issues[2]. The approach presented in this thesis is similar to RAGS in that it also uses a low-level data type, but it makes much stronger assumptions on the architecture of the system and the underlying algorithms (see Chapter 5).

2.2 Linguistically motivated architectures

Unlike the task-based pipeline architecture, which defines task oriented modules, linguistic theories differentiate between different levels of abstraction. The description of language using levels of abstraction may be compatible with a task-based view of NLG. For instance, document planning may be viewed as solving issues at the rhetorical and semantic level. Microplanning can be seen as implementing the semantic-syntax interface, and realisation as making decisions about morphology and possibly phonology. In the details however, the comparison does not necessarily hold.

Perhaps one of the best examples of the general mindset of linguistic approaches to natural language generation is the Meaning-Text Theory (MTT, Polguère 1998, Kahane 2003), and its applications to NLG (Lavoie and Rambow, 1997, Coch, 1996). The meaning-text theory is a formalism describing a bidirectional pipeline of six modules (see Figure 2.2). Each module makes the connection between two levels of abstraction. The seven levels of abstractions are semantic, deep-syntax, surface-syntax, deep-morphology, surface-morphology, deep-phonology and surface-phonology. The architecture is generic, and can be parametrized for different languages or applications by defining a *lexicon* and *correspondence rules*. The lexicon defines the structures which compose the different

[2]It is hard to tell if an implementation of the RAGS project would really suffer from performance issues in an industrial context. The performances can vary greatly between different implementations, and the architecture is flexible enough to allow opportunistic limiting assumptions. However the proposed event-based implementation, like any event-based architecture, is quite complex (as compared for instance to a pipeline architecture).

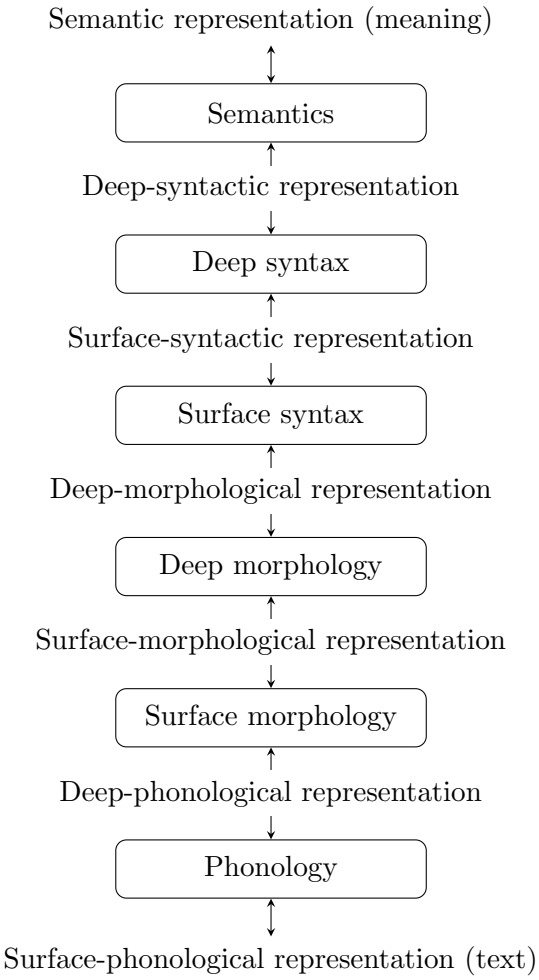


FIGURE 2.2: The meaning-text theory bidirectional pipeline architecture. Six modules connect seven levels of abstraction, from the meaning of the text to the surface text.

levels of abstraction, while the correspondence rules define the possible transformations between the levels of abstraction.

Other formalisms have been used as a basis, either for implementing particular modules of a task-based NLG pipeline or complete NLG systems. For instance FUF (Elhadad, 1993) is presented as a syntactic realiser inspired from Functional Unification Grammars (FUG, Kay 1979). It comes with SURGE (Systemic Unification Realization Grammar of English, Elhadad and Robin 1996), a wide coverage grammar of English and produces text from thematic structures. Another example using TAG (Joshi, 1985) for microplanning is the G-TAG formalism (Danlos, 2000), which describes the semantic-syntax interface. These systems also make independence assumption based on the distinction between different levels of abstraction. For instance, G-TAG distinguish between conceptual representations, semantic-syntactical representations and syntactic-morphological representations (and surface text).

Linguistically motivated systems have been mainly used as independent modules for syntactic realisation (Elhadad, 1993, Lavoie and Rambow, 1997). However, when they include a semantic representation, they may do full or partial lexicalisation, aggregation or referring expression generation. Although pipeline architectures are quite standard for linguistically motivated systems, the justifications used for the modularization are different from a task-based approach. This often results in different independence assumption and makes the comparison between task-based and linguistically motivated architectures harder (see Chapter 4 for more on this issue).

Using ACG can be considered as a linguistic approach to NLG. Like MTT, it uses grammars in order to describe transformations between different levels of abstraction. However in ACG no assumption on the number of levels of abstraction is made, and there is a unique abstract data type used for every level of abstraction. This theory neutral stance makes a big difference in practice, as it allows more flexibility.

2.3 Combinatorial optimization

The problem of choosing variants under some constraints can be formalized as a combinatorial optimization problem. Combinatorial optimization has a vast literature and many subfields and is often used to represent the NLG process or some part of it.

Combinatorial optimization is the problem of finding an optimal solution in a set (finite or infinite) of candidates. The optimality criterion is typically given by a cost function which returns a single numerical value for each candidate solution. A solution is optimal if no other solution is given a lower value by the cost function (or a higher value by a value function, depending on the viewpoint). In some cases, simply finding viable candidate solutions is a difficult task, and the optimality criterion has no importance or doesn't even exist. In these cases, the problem is rather called a *constraint satisfaction problem*, or satisfiability problem (see Rossi et al. 2006 and Boyd and Vandenberghe 2004 for detailed introductions on the subject).

2.3.1 Constraint satisfaction problems

The core concept behind all combinatorial problems is the concept of *search space*. A search space defines the set of potential solutions to the problem. Once it is defined, we may then browse it to look for actual solutions, optimal or not. A search space is defined by two things: *variables* and *constraints*. A variable represents a choice that must be made in order to characterize a solution to the problem. Any particular choice is called

a *value* and the set of all possible choices represented by a variable is called its *domain*. A constraint is a limitation on the possible values that a group of variable can take.

Different classes of problems with different complexities and methods can be identified based on the kind of variables and constraints which define the search space. Here are some of the most well known classes:

- Real valued variables along with *linear constraints* (i.e. linear equations), are solved by *linear programming* (LP, Dantzig, 1998) methods. These methods can solve problems with up to millions of variables in reasonable time. If some or all of the variables are integer valued, then the methods fall in the category of *linear integer programming* (LIP).
- If the domains of the variables are convex sets and the constraints are convex functions, then the problem can be solved using *convex optimization* techniques (Boyd and Vandenberghe, 2004). Convex optimization techniques may also solve quite sizeable problems, but offer less guaranties than linear programming techniques.
- If some of the domains of the variables are not convex sets or some of the constraints are not convex functions, then we fall in the general category of constraint satisfaction problems.

The methods for solving general constraint satisfaction problems (with or without optimization) can be divided into two broad categories: *search* and *local search* techniques. Search techniques are adaptations on the brute force search algorithm. The idea is to exploit the regularities and dependencies in the search space using inference techniques in order to avoid checking most of the potential solutions. The local search techniques on the other hand (often) use a less systematic approach. The general method is to initialize variables with random values or a heuristic potential solution, and then make small changes to the variable values in order to find better potential solutions around the first one. This allows to quickly find good solutions in huge search spaces. However in the case of optimization problems, local search techniques often give no guaranty that the best solution found is indeed the best solution (but it guaranties that it is the better one in some corner of the search space). There are many local search techniques, among which: gradient ascent (and in general all gradient based techniques), tabu search, simulated annealing, genetic algorithms, etc..

Most of the combinatorial problem solving techniques used in NLG are search techniques. Therefore I will now introduce search related concepts in more details.

Constraint programming includes many techniques for searching efficiently through a search space. It is sometimes presented as including local search techniques and some

specialized optimization techniques. Here I focus on classical search techniques only, as most implementations of general purpose constraint programming solver are limited to this case (see for instance Schulte et al. 2010 on the Gecode C++ solver).

Constraint programming techniques are based around two main concepts: *search strategies* and *inference methods* (Rossi et al., 2006). A search strategy basically decides in what order the choices should be made. The two broad categories of search strategies are *breadth-first search* and *depth-first search*. A breadth first search will explore the search space by listing all intermediary partial solutions to the problem, and build more and more precise solutions at each iteration. On the other hand, depth-search techniques will rush for complete solutions to the problem, forgetting along the way partial solutions which have already been explored. Breadth-first search is typically good at finding all possible solutions to a problem, but consumes a lot of memory. Solvers use in general depth-first search techniques. The general process for depth first search is to choose a variable, then choose a value for this variable and loop on another variable until all variables have been assigned a value. The function which decides the order in which the variables are selected is called the *variable heuristic* and the one which decides the order in which the values for each variable are tested the *value heuristic*.

The other aspect of constraint programming is inference. Inference in constraint programming is called *constraint propagation*. The basic idea is to use the choices made during search to predict that some combinations of choices cannot lead to a viable solution and remove them from the list of choices which are yet to be made. If at some point a variable has no more viable values, then the search *backtracks* to the last choice made in the search and tries another value.

Each type of constraint has its own inference procedures. Inference in itself is a method for solving constraint satisfaction problems. However it is exponential both in time and space. Constraint programming techniques have to trade-off between exhaustive search on one side and full inference on the other. By combining the right level of inference with the right level of search and good heuristics, one can check many potential solutions in the most promising parts of the search space and solve hard problems which cannot be solved by search or inference alone.

A technique similar to constraint programming, yet with a somewhat different view on the way a search space should be defined, is *logic programming* and its extensions into *constraint logic programming* techniques. This includes for instance the Prolog language and its extensions or restrictions like Datalog systems. This last method is the one used by the implementation presented in this thesis (see Chapter 8).

2.3.2 NLG and constraint satisfaction

Combinatorial optimization techniques give a natural framework for describing and solving hard combinatorial problems. Since NLG is a combinatorial problem, most of the techniques developed for NLG can be seen through the lens of combinatorial optimization (Piwek and Van Deemter, 2006), whether the techniques explicitly refer to combinatorial optimization or not. The complexity of NLG is reflected by the numerous types of constraints which can be used. For instance, the definition of a search space for NLG may include morphological, syntactic, semantic and discourse constraints, but also pragmatic and style constraints, or other domain dependent constraints like the size of the text and user/speaker model related constraints.

Although systems explicitly using combinatorial optimization techniques generally aim at solving as much dependencies as possible, the systems often focus on particular modules of a task-based pipeline architecture. For instance, (Marcu, 1997) defines a constraint satisfaction problem for document structuring, (Elhadad et al., 1997) explore "floating constraints" appearing during lexicalisation, (Callaway and Lester, 1997) use constraints to control clause aggregation in a revision-based microplanner, (Gardent, 2002) defines a constraint satisfaction problem for referring expressions generation and the linguistic realizer FUF (Elhadad, 1993) uses a breadth-first search algorithm. This illustrates the fact that combinatorial optimization techniques are not incompatible with a modularized architecture.

Different problems in NLG have been modelled using different kinds of structures, variables and constraints, giving search spaces with different properties and different associated solving methods. For instance, (Marciniak and Strube, 2005, Lampouras and Androutsopoulos, 2013) use linear integer programming techniques to jointly solve several sub-tasks of the NLG pipeline. Revision-based architectures, like (Inui et al., 1992), could be seen as local search methods, with an initial guess (or draft) and then local optimization by revision of the initial guess. Finally, classical search methods are the default implementation for most linguistic formalisms, like unification-based grammars (Elhadad, 1993) or tree-adjoining grammars (Meunier, 1997).

To conclude, combinatorial optimization techniques fit rather naturally in implementations of NLG systems, whether the systems makes strong independence assumptions or not. They tend to be used when the problem is sufficiently complex to justify the use of elaborate techniques and they are still limited by combinatorial explosion issues. Approaches based on formal grammars are particularly compatible with combinatorial optimization techniques, since they are defined precisely and can be described easily as a search space. This is why it is used in the implementation presented in Chapter 8.

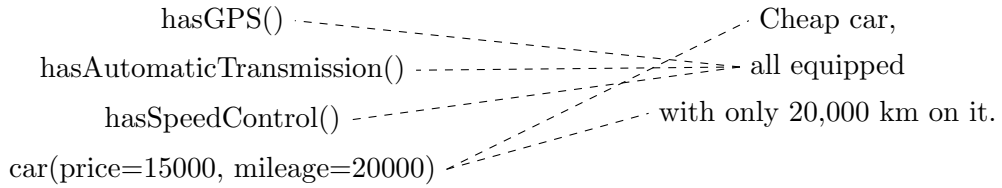


FIGURE 2.3: Alignments between a conceptual representation and text.

2.4 Machine learning

No matter how sophisticated the method is, the approaches for NLG presented until now need human intervention to adapt to new domains of application. This involves in particular defining or parametrizing the algorithms for each module, which requires knowledge and experience. Machine learning techniques theoretically allow to automatize the parametrization of modules and to adapt NLG architectures to several domains without human intervention. The promises of learning NLG systems have led to a growing interest in the field over the past decade.

Machine learning uses probabilistic models to represent the NLG problem. The model can cover the whole NLG process or only specific modules. Learning can also be more or less *supervised*. The strength of the supervision over the learning process is given by the amount of information given in the *training data*. The more information there is in the training data, the easier it is for the machine to learn dependencies. The drawback is that the training data needs to be annotated with all the required information, which makes the construction of big datasets harder. Unsupervised methods on the other hand only require little information about the training data, and often have bigger training data samples available. Since the size of the training dataset has a great impact on the performance of learning, unsupervised methods may be competitive with supervised methods on some tasks.

2.4.1 Learning alignments

The basic problem of NLG is to transform some input data into surface text. This can be modelled by alignments between conceptual representations and surface text chunks. Figure 2.3 gives examples of alignments between database entries, corresponding to the conceptual representation, and surface texts for a car selling application. The text is aligned with database records. A record is an object-field-value triplet. Each record and each field of each record is associated with a type. The type of a record defines the fields of the record and the type of a field the possible values for this field, like for instance integer values or string values. A supervised method would use a dataset where

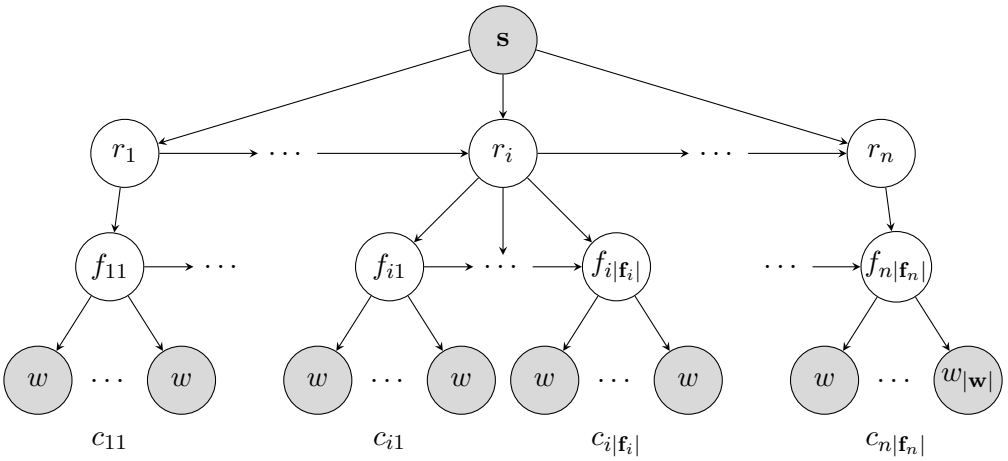


FIGURE 2.4: Hierarchical semi-markov model for alignments, reproduced from (Liang et al., 2009): first, records are chosen and ordered from the set **s**. Then fields are chosen for each record. Finally, words are chosen for each field. The world state **s** and the words **w** are observed, while (**r**,**f**,**c**) are latent variables to be inferred (note that the number of latent variables itself is unknown).

the alignments are given and would learn the correspondences between database records and fields and surface text. However the precise alignments between text and fields of database records are not always given. In this case, less supervised methods are needed.

Figure 2.4 shows an alignment model from (Liang et al., 2009), where the alignments are decomposed between a database, records, fields and words. Records and fields are used as latent variables and each level include markov dependency chains. The advantage of using hidden variables for records and fields is to avoid supervision for the particular alignments between words and fields. The model of Figure 2.4 can be trained on a collection of scenarios (sets of records) paired with surface text without much more information and learn to select the database records and fields, so to do content selection.

2.4.2 Learning to generate

Using alignments, the task of NLG can be modelled in different ways. Like for other system architectures, the independence assumptions are key to define the general architecture. Probabilistic models have their own way of defining independence assumptions, based on random variables independence rather than modules. Nevertheless, the same kind of independence assumption, and therefore the same kind of architectures, can be used with probabilistic models. For instance, some models explicitly refer to a pipeline architecture (Angeli et al., 2010), by modelling a sequence of choices: choose a record in the scenario, then choose a field in the record and choose a template to textualize the field (and then loop). These choices can be seen as doing content selection, lexicalisation

and realisation. Each choice is made dependent on all previous choices by a set of feature vector templates.

Since a NLG system based on a probabilistic model can be trained without human intervention, it is tempting to complexify the system and add more dependencies. The added complexity impacts the size of the training sets, but not the difficulty to use the system. Therefore architectures which solve several parts of the NLG problem jointly are more popular among learning systems than hand-crafted ones. This is the case for instance of (Konstas and Lapata, 2012, 2013a,b), who use probabilistic grammars to jointly model content selection, document planning, lexicalisation and realisation.

2.4.3 Machine learning and generic architectures for NLG

The above presentation and this thesis in general do not really do justice to machine learning techniques for NLG. My focus is clearly on generic architectures for hand-crafted NLG systems, with human intervention for the specialization of each module. This is actually a general trend in the NLG industry, where probabilistic methods are not quite as popular as for instance in the Natural Language Understanding (NLU) industry. I can see two main reasons for that:

- The lack of data. Although machine learning techniques don't theoretically need human intervention, they do need data, preferably annotated. There may be projects for which a high quantity of example texts along with their associated input data are available, but this is not the most common situation. NLG may be used to automate the generation of texts which where hand-written before, but in most cases, NLG technologies are used to generate texts which could not be written before, because of the quantity and regularity of the publications. Even when examples are available, NLG technologies are used to augment the general quality of the text produced, as well as automatize it, so the texts need to be reviewed and adapted or normalized. The work needed to build a good training dataset is not negligible in comparison with well formatted and efficient software construction methods.
- The lack of control. In many domains, the quality of the generated text is a necessity, especially for instance, if one of the goal of the application is to produce better and more normalized texts than the ones produced by humans before, or if end users are professional workers. Although great progress have been made, probabilistic method still have a non-negligible margin of error in their output. It is easier to predict and validate the output of a hand-crafted system than a learning one.

For some situations, probabilistic methods for NLG are perfectly adapted and used, even in industry. However the advantages of machine learning do not generalize enough to all practical applications to be the default method for building NLG systems yet. However this will undoubtedly be the case at some point, and learning methods are very promising.

Chapter 3

Document planning

Document planning involves selecting the content to generate and produce a document structure which contains the conceptual content of the text (Reiter and Dale, 2000). Content selection and document structuring, the two sub-tasks of document planning, can be viewed as separate tasks or may be done in an interleaved manner. Section 6.6 focuses on content selection and the input data of the NLG task. Section 3.2 gives an overview of the different kinds of structure that a document planner may produce. Section 3.3 presents in more details some algorithms for document planning.

3.1 Content selection

If data to natural language generation is considered as a module, then content selection represents the interface between NLG and the rest of the world. One of the greatest difficulties for building generic NLG systems is that there exists no precise and universal definition of the input. Content selection is directly impacted by this problem, and therefore is probably one of the most ill defined task of the NLG pipeline. There are, however, some representations or structures which are used in several applications. Several standardization attempts have been made, in order to capture the regularities in the potential inputs of NLG.

3.1.1 Domain representation and ontologies

Almost every NLG system has a conceptual representation or *ontology* of the domain. An ontology is a structured representation of the concepts (entities, events, relations, attributes, etc.) of the domain of application. This conceptual level of representation can either be the input of the NLG system, or an intermediary representation in a more

id	brand	price	color	options
001	Mercedes	$32,563	gray	GPS,autolock,cruise control,...
002	Renault	$23,802	black	GPS,automatic transmission,...
003	Toyota	$26,358	black	cruise control,self parking,...
...	...	...	...	...

FIGURE 3.1: Simple database example from the car selling application domain.

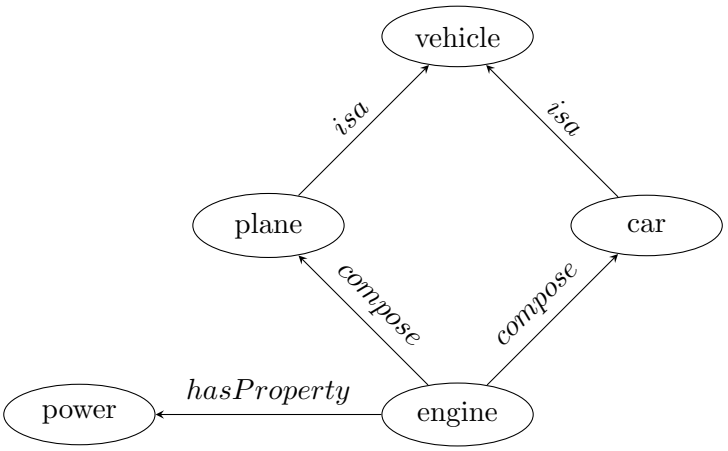


FIGURE 3.2: Example of ontology fragment with three relations between some concept in a vehicle ontology (*isa* is a shorthand for "x is a y").

complex architecture. In either way, the content selection module is concerned by the manipulation of ontologies. For this reason, most of the work on the standardisation of content selection has focused on different kinds of ontologies.

One of the simplest and most broadly used kind of ontology is relational databases. Figure 3.1 shows an toy database example. The table represents a concept of the domain (cars). Each line represents an existing entity in the domain and each column an attribute of the concept of car. Since databases are used in many software, it is often the case that NLG applications are plugged directly onto an existing one. Therefore, many generic content selection applications assume an input in a simple tabular form. This is especially true for machine learning-based approaches (see Section 2.4).

Other, more elaborate examples of standard formats for the conceptual representation come from the semantic web, such as the OWL (McGuinness et al., 2004) or RDF (Beckett and McBride, 2004) formats. Two examples of application of natural language generation for the semantic web are: explain concepts in an ontology for an end user, and help the designer of the ontology to understand quickly the structure of the ontology (see for instance Bontcheva 2005, Galanis and Androutsopoulos 2007). The general form of an ontology is a graph, where nodes are concepts and edges are relations between concepts (see Figure 3.2). It also has a logical form which can be used for inference.

Even in the case where a structured representation exists for the domain of application, this representation needs to be adapted for natural language generation. The reason is that a pure logical form does not usually contains all the information which allow to select a content to be expressed. There is usually a gap between the concepts of the domain and the logical forms which correspond to the messages which must be generated (Mellish and Pan, 2008). External factors may also impact content selection. In details, the issues which may arise in the content selection module are the following:

- If a precise goal drives the content selection, the conceptual representation may need to be adapted in order to be able to compare the utility of each concepts relative to this goal. Such modification may also be needed for instance if a notion of quantity of information is used.
- Some concepts that should be communicated may be implicit in the conceptual representation of the domain. In this case they must be inferred from the existing data.
- The opposite situation may happen, where there are too many concepts, and they should be merged into higher level concepts, which summarize the situation.
- The conceptual representation used may consider equal several representations which are not equal in regards to their textual form. These synonymous representations need somehow to be differentiated.
- All conceptual representations may not all be equally textualizable. In general, linguistic concerns may arise in the selection of concepts, as well as other constraints, such as the output size of the text.
- External factors may come into play, such as the discourse history and the user model.

Methods for content selection using ontologies include inference-based methods (Mellish and Pan, 2008, Bouayad-Agha et al., 2011, 2012, Bouttaz et al., 2011), and graph-based methods (O'Donnell et al., 2001, Dannélls, 2009, Demir et al., 2010). Each one exploits different forms of the structure of the information, either its logical structure or its graph structure, but in both cases, search and optimization methods are used to find the best subsets of concepts to select (see Section 3.3 for different approaches of this problem).

3.1.2 Unstructured input data

The input of a NLG system is not necessarily a fully structured ontology of the domain of application. Different degrees of structure exist, and unstructured input data involves

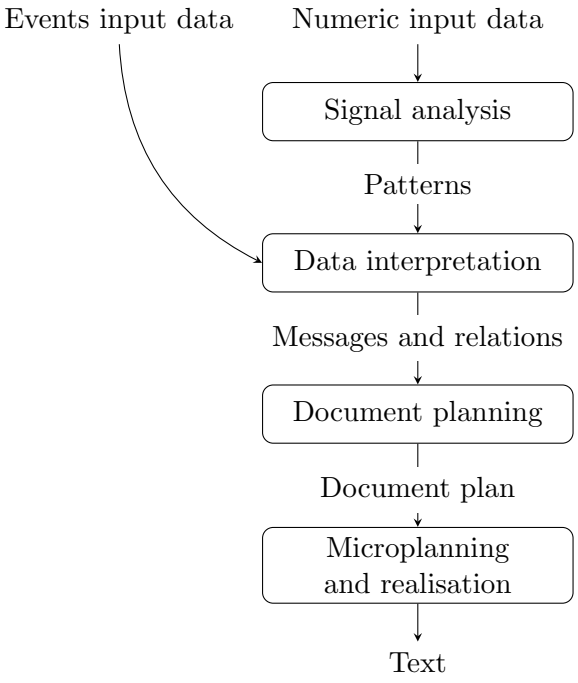


FIGURE 3.3: The extended task-based pipeline architecture. Adapted from (Reiter, 2007). Includes two new modules on top of the document planning module: signal analysis for raw numerical data and data interpretation for the creation of messages and merging of different data sources.

additional problems to be solved for the content selection module. In particular the system may need to preform new tasks, including:

- Pattern detection. In the case of a continuous raw input or low level numerical series, the system may need to perform signal analysis and pattern detection on the input data.
- Concept creation and relation detection. Several low-level patterns may be represented by a higher level concept. Relation between concepts may arise from long range dependencies between patterns or contextual knowledge. This problem is not specific to unstructured input data. Structured data can also have missing or redundant information (see Section 3.1.1).

An extension to the generic task-based pipeline architecture has been proposed in (Reiter, 2007), which includes processing of unstructured (or semi-structured) data. Figure 3.3 shows the extended architecture. The two new modules are signal analysis and data interpretation. The signal analysis module is in charge of detecting the patterns in the input data. Then the data interpretation module infers on these patterns to build concepts (or messages) which describe the data and relations between the concepts. This step may also involve the combination of several data sources, which can be more or less

structured. For instance the system may receive timed events in a structured format which add to the contextual knowledge. The other steps of the pipeline are the same as before. Other systems which include raw numerical data analysis are for instance (Wanner et al., 2010, Yu et al., 2004).

Another kind of unstructured data is text. For example, in dialogue applications, the input of the system is raw text entered by the user. In this case, the system must first analyse the text in order to understand it. This analysis part is similar to a signal analysis module. However in this case it involves natural language understanding rather than signal processing. Natural language understanding may also connect the NLG system to other kinds of inputs, like sentiment analyses from blog posts, or article summaries.

Although content selection needs to be adapted to very different inputs in very different domains, some regularities allow to reuse some of the components from one application to the other. Pattern analysis algorithms may be reused for different domains, some generic inference methods over ontologies also exist. Some generally applicable data format for the conceptual representation are available, like OWL. However, a lot of human expertise is still needed to build the domain knowledge necessary for natural language generation. This human expertise bottleneck is the same than for expert systems and is yet to be resolved. Machine learning helps a lot, but has some disadvantages (see Section 2.4), and the general issue of a global adaptive conceptual representation, which is akin to general artificial intelligence, is still unsolved. The tests realized for this work (see Chapter 9) limit the input to structured data in a format close to OWL.

3.2 Document structure

The structure of the document, or document plan, is the output of the document planner and involves the conceptual representation of the text, but also rhetorical relationships and formatting related concepts, like chapters, paragraphs, bullet lists, figures, etc..

3.2.1 Schemas

Schemas (McKeown, 1985), encode fixed patterns in the structure of documents. They define a tree structure where internal nodes are either objectives (e.g. “describe some object”), rhetorical relationships or structuring elements, and leaves are messages. Figure 3.4 shows some example of possible schemas. In essence, schemas are context-free grammars for document plans. They often use the Kleene star and plus symbols (zero or more and one or more respectively), and conditional nodes (nodes which may be removed depending on some condition).

```
describe-car ->
    describe-brand-and-price
    describe-options +
    describe-other-advantages *
    describe-disadvantages *
```
(1)

```
descriptive-sequence ->
    identification
    [elaboration] +
    [contrast] *
```
(2)

```
descriptive-paragraph ->
    introduction
    options-list
    conclusion
```
(3)

FIGURE 3.4: Examples of schemas for the definition of fixed document structure patterns. Each rule defines a sequence of elements, where each element is another schema (potentially a recursive definition) or a message. The rules allow conditional nodes and Kleene star/Kleene plus notation.

Conditional nodes associated with a context-free like grammar definition gives the schemas formalism a great expressibility (actually the expressibility of a general purpose programming language). Therefore there are many ways to use schemas, and often several ways to achieve the same result. In Figure 3.4, different global schemas for the description of a car are presented. The first one uses goals, with a main goal being decomposed into several sub-goals. The second uses rhetorical relationships, and describes the structure of the text in terms of rhetorical concepts. Finally, the third one simply describes the structure of the text with simple descriptive structures, like introduction or conclusion. Including text templates into schemas, one could even do the whole NLG process using only schemas. This illustrates the expressive power of schemas, which makes it a very popular technique for document planning (see for instance Reiter and Dale 2000, Bontcheva 2005). The downside of this expressive power is that schemas are less susceptible to be reused between different applications. The problem is similar to the one with text templates. When there are many ways to do the same thing, part of the architectural conception effort is reported on the organisation of the schemas. This configuration favours domain specific structures. In environments with variable inputs and goals, schemas may also be too rigid to cope for the diversity of document plans needed (again there is a similar problem with text templates). That said, schemas are the default choice for many practical applications. Their flexibility and simplicity make them suitable for many situations and more elaborate methods do not necessarily give more reusable or easy to use components. In the example application of Chapter 9, the formalism of RST is preferred to schemas, but this choice should not be taken as a definitive judgement on schemas.

3.2.2 RST

The decomposition of documents into goal descriptions or organisational structures like chapters, introduction, conclusion, etc., are often based on conventions rather than precise formalisms. Formalisms exist, however, which describe the structure of the text at a multisentencial level, linking communicative intentions to specific order of

Evidence relation	
constraints on N:	The reader might not believe N to a degree satisfactory to the writer.
constraints on S:	The reader believes S or will find it credible.
constraints on N and S:	The reader's comprehending S increases its belief of N.
effect:	The reader's belief of N is increased.

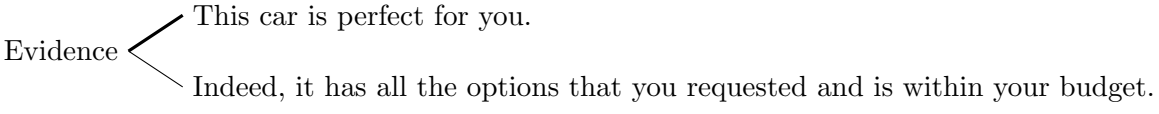
FIGURE 3.5: The evidence relation from the RST (adapted from Mann and Thompson 1988). N stands for nucleus and S stands for satellite.

presentation for the concepts. The conceptual structure of the text at a multisentencial level is called the *rhetorical structure* of the text. Theories about the rhetorical structure of text have been used extensively for document planning, either as an inspiration or as the core concept.

A popular theory about rhetorical structures for NLG is the Rhetorical Structure Theory (RST, Mann and Thompson 1988). RST describes relations between parts of the text. A relation between two parts holds if some condition on the meaning of each part is satisfied. A relation also defines an effect which is achieved by using the relation. The effect of a relation can be for example how the relation impacts the beliefs or sentiments of the person reading the text. More precisely, the relationships are binary relations defined to hold between two non-overlapping text spans. One text span is called the nucleus of the relation while the other is called the satellite of the relation (sometimes however relation may have two nuclei). A relation is defined by four fields:

1. A set of constraints on the nucleus of the relation
2. A set of constraints on the satellite (or the second nucleus) of the relation
3. A set of constraints on the combination of the satellite and the nucleus (or two nuclei).
4. An effect that the relation has, generally on the reader.

Figure 3.5 shows an example relation from Mann and Thompson 1988, the *evidence* relation. This relation describes a situation were the writer of the text provides some support to a claim. The following example illustrates a text with an evidence relationship between two sentences:



The thick branch of the relation represents the nucleus and the other one the satellite. The nucleus of a relation represents the salient part of the text, i.e. the one that matters most. The general rule of thumb is that the nucleus has a meaning independent from the satellite, but that the satellite cannot be understood without the nucleus. Sometimes no salient part can be found. In this situation, the convention is for the relation to have two nuclei. The arguments of rhetorical relations can be other rhetorical relations. The overall structure forms a tree with rhetorical relationships as internal nodes, and text spans (or messages) as leaves.

The RST is intended as a descriptive formalism. The definitions of the relations are informal (see Figure 3.5), and there is no closed set of relations which are intended to encompass all possible rhetorical relationships. Although rather comprehensive sets of relations exist, these sets are intended as a basis for further extensions and specializations for each domain of application (Mann and Thompson, 1988, Hovy and Maier, 1992).

3.2.2.1 RST and NLG

RST is more often used in NLG as a descriptive framework rather than a core concept of the document planner. For instance, a schema based document planner will often include rules with labels inspired from the RST, without any formal external definition of the relations. However there have been several successful use of RST with precise definitions for the relations and explicit manipulation of RST based rhetorical trees (Moore and Paris, 1993, Marcu, 1996, 1997).

The informal approach of RST makes it very adaptable, but also ambiguous. Some aspects of the theory may have several interpretations when it comes to an implementation. In particular, the following problems may arise (Marcu, 1996):

- There may be ambiguities between several relationships applicable to the same content. It is possible that an analysis of a text finds several relationships between a particular part of the text and several other parts. RST describes tree structures and cannot have a leaf element involved in more than one relation. Therefore we need some decision mechanism to disambiguate between several possible relations.
- There is no clear definition of the composition mechanism of rhetorical relations. The problem comes as follows: if there are two relations, each one on two text spans, what relations hold between these two relations ? Since relations are defined between text spans or messages, we need a mechanism to apply recursively the relations on themselves.

Marcu gives a formal definition of valid rhetorical structures. The problem of relations composition is solved by considering that a relation is represented by its nucleus, following the principles of Mann and Thompson (1988). In this set up, a relationship between a relation and other elements of the rhetorical structure is defined by the relationships between the nucleus of this relation and other elements of the rhetorical structure.

Most of the RST structures used in NLG do not include the effect of the relations. Often the intention is left implicit in the rhetorical structure. The developer of the document planner defines the structures with the intention in mind and there is no need to add any particular representation[1]. However, the effect or intention behind the relations can be made explicit. This is the case for instance in (Moore and Paris, 1993), who build document plans based on RST relations with explicit intentions. An explicit representation of the intention opens the way to more elaborate document planning. For instance it can be used to differentiate between different candidate structures. It is also almost indispensable if one is to revise the document plan based on new evidence. For instance (Moore and Paris, 1993) build a dialogue system, where the intention behind each document plan is remembered between consecutive calls in order to assess if the intended goals have been achieved in regards to the response of the user.

RST gives a good compromise between practical concerns and formalisation, and has been chosen to implement a document structuring module (see Chapter 7). Since dialogue situations were not in the scope of the tests (see Chapter refchap:results-and-perspectives), I used as a bootstrap the formalisation of (Marcu, 1996), without an explicit encoding of the intention or effect.

3.2.3 SDRT

A majority of NLG systems use RST like trees for representing the rhetorical structures. However some attempts have been made, which use other formalisms for representing the structure of discourse. One of these formalism is the Segmented Discourse Representation Theory (SDRT, Lascarides and Asher 2007).

Unlike the RST, the SDRT does not include any effect or intention related information. However it includes by default a representation for the semantic content of the text, where the RST is only concerned with the rhetorical relationships. The SDRT represents the interface between the rhetorics and the semantics of a text. Interleaving the rhetoric and semantic layers allows to make some inferences on the discourse structure

[1] An intention is any effect (generally on the reader) which is intended by using a particular relation. For instance the intention behind a relation may be to persuade someone that something is true, or to motivate someone to do something. Here I use the terms “effect of a relation” and “intention behind a relation” as synonyms.

and discover relationships (Roze, 2013), which can be useful for building document plans dynamically from a semantic representation of the text. The SDRT is also a slightly richer representation, as its underlying structure can be represented by a directed acyclic connected graph rather than a tree (Danlos et al., 2001). More generally, using a formal semantic representation as a basis for the definition of rhetorical relationships makes them more universally applicable, but also implies a specific semantic representation, which can be harder to use. While formal approaches are attractive in theory, the practical problems encountered in document planning (at least in the context of this thesis) usually do not justify the added complexity, and the semi-formal approach of RST has been preferred over SDRT.

3.3 Algorithms for document planning

Algorithms for document planning can be roughly separated between top-down and bottom-up algorithms. The top-down approach generally puts the focus on some goal and builds the structure iteratively from this starting point. On the other hand, bottom-up algorithms will focus on global coherence or more diffuse goals either to select the content or build the document plan. Another distinction which can also be made is between algorithms that perform content selection and document structuring as two separate tasks or as a single task.

3.3.1 Top-down goal driven

Top-down algorithms for document planning take as a starting point a communicative goal or a high level objective. This overarching goal is then refined into more and more precise sub-goals, until a set of atomic goals that achieve the initial goal is found. As an example, take a car selling application where the goal is to present arguments in favour or against some car with respect to the user needs. The overall goal of the application can be used as a basis to select which arguments should or shouldn't be included in the text as well as the order of presentation of the arguments (see Carenini and Moore 2006 for an example of goal oriented argumentation generation).

Schemas are a natural implementation of the top-down approach. A schema is a high level construct which is defined by lower level (or recursive) structures. Therefore, simply expanding a schema top-down left to right creates a document plan in a goal-driven manner. By creating different global schemas and using conditional branching, one can attain a fairly good variability. However schemas are sometime too rigid and cannot adapt to very different situations.

More flexible algorithms based on hierarchical planning have been used (Hovy, 1993, Young and Moore, 1994, Carenini and Moore, 2006, Paris et al., 2010). Planning algorithms are used to build plans based on a set of actions in order to achieve one or several goals. An action operator is typically composed of a set of preconditions that must be true before the action is performed, and a set of postconditions which define what is true after the action has been performed. A hierarchical plan decomposes plans into higher level and lower level actions, the lower level actions composing the higher level ones. Document planning usually involves intermediary goals based on RST-style rhetorical relationships and atomic actions which represent speech acts, or messages.

Dynamic planners have the possibility of revising existing plans, which is of particular interest in dialogue situations, given that the plan has explicit intentional content (Young and Moore, 1994). Planners also have the advantage of having specialized modelling languages, like STRIPS (Fikes and Nilsson, 1971), which helps in the definition and adaptation of the document planner to different domains.

3.3.2 Bottom-up data driven

Unlike top-down techniques, bottom-up techniques for document planning are focused on the interactions between low level elements, rather than an overarching communicative goal. There still may be a goal in a data-driven approach, but it will typically be more diffuse and not necessarily explicit. In general the goal of data driven approaches is akin to global coherence, or low energy configurations.

An example of data-driven content selection is the ILEX system (O'Donnell et al., 2001). In this system the content selection module explores a graph from a starting point using a heuristic defining the relevance of items and a beam-search algorithm. Another example of graph-based bottom-up approach to content selection uses the page rank algorithm (Demir et al., 2010). With content selection based on the logical form of the conceptual representation, a data-driven approach algorithm is forward-chaining (breadth-first inference from initial facts) and variations on that basis (Mellish and Pan, 2008).

For document planning, bottom-up approaches generally assume that a set of messages have been selected and focus on building the document plan or rhetorical structure based on these messages. The task of building a rhetorical structure from a set of messages is similar to text parsing, except that the input symbols are unordered. The possible structures are typically defined by a set of rhetorical relations on the initial set of messages. The relations are either defined extensionally by a given set, or intentionally by formal definition of the relations relative to the conceptual representation of the

messages. The goal of the document structuring module is then to select the appropriate relations and compose them together. This can be done for instance by a depth-first search with pruning algorithm (i.e. constraint programming, Marcu 1996, 1997). More generally search or optimization algorithms can be used to explore the set of all possible structures.

Data-driven techniques are good at building coherent texts in various situations. This is especially useful when the input of the NLG system varies a lot and many situations must be handled. However bottom-up strategies have trouble converging toward a single overarching goal. Therefore using a bottom-up or a top-down approach often depends on how the problem is naturally expressed: either as a goal oriented planning process, or as a global coherence problem.

The document planner developed for this thesis (see Chapter 7) uses the bottom-up approach. This choice comes from the domain of the application which has been used as a test bed, which is better represented as a global coherence problem (see Chapter 9). However a top-down implementation is also possible, and even desirable in other contexts.

3.3.3 Interleaved content selection and document structuring

Choices concerning content selection and document structuring can be interleaved into a single algorithm. This configuration is more natural with top-down strategies. Indeed, in a goal driven approach, the goal is used as basis for decisions in both content selection and document structuring. Therefore it is rather natural in this case to not differentiate between these two kinds of decisions. The typical algorithm for a goal oriented interleaved approach to document planning builds the structure of the document from the text in a top-down fashion and selects the content to be expressed in order to complete the structure when a leaf is reached.

Another possibility, which can be used with both goal-driven and data-driven systems is revision of the document plan (Rambow, 1990). Document plan revision is a structure optimization process which applies on a complete document plan rather than some initial structures or goal. Such system may revise some of its previously made choices (either for content selection or document structuring) in order to improve the quality of the initial document plan. This technique is not common for document planning, but has been used extensively for microplanning, which is the topic of Chapter 4.

3.4 Summary

The document planning module takes as input some data and uses a conceptual representation of the domain to complete, select or create messages, and organizes them into a document plan. The input data may be more or less structured. From the less structured to the more structured, we may have: raw numerical input, text, tables, relational databases and full-fledged ontologies. Unstructured data like numerical data or text is more often processed before document planning, in a module either considered as an extension of the NLG pipeline or as a separate system. Therefore the input to the document planner is often considered to be an ontology with more or less internal structure. The output of the document planner, the document plan, is typically a tree (e.g. a schema), which may or may not be based on a linguistic rhetorical theory like RST or SDRT. To bridge the gap between the initial conceptual representation and the document plan, the two main approaches are top-down, goal oriented algorithms and bottom-up data-driven algorithms. Formalisms exist, which describe the data used by the document planner, like OWL, RDF, RST, SDRT, etc.. The algorithms are generally described using a general purpose programming language. However some specialized languages exists for systems using planners, like STRIPS.

Most of the techniques presented in this chapter use classical search to find suitable structures. For instance, schemas search through dynamic structures, while planners search through the space of possible plans. All inference-based methods also use classical search to browse the space of valid logical expressions. The search may be depth first or breadth first, with or without heuristic. Data-driven approaches may also optimize an objective function.

Depending on the data involved and the algorithms used, the developer of the document planning module may have more or less decisions to make. The number of decisions which have to be made has a direct impact on the extendibility, reusability and ease of use of the module. For instance, schemas have a great expressibility and allow custom conditional branching mechanisms, which allow to control precisely the flow of execution. While being simpler to learn and to use, such an approach also requires great care and a lot of knowledge to be able to build generic, reusable schemas. On the other hand, other techniques based on more constrained formalisms, like SDRT, are harder to learn and to use, but allow some generic treatments and inference mechanisms, applicable to all systems based on this formalism. RST is generally accepted (in NLG) as a good compromise between formalisation and simplicity.

Chapter 4

Microplanning

The microplanning module transforms a conceptual representation of the text, embodied by the document plan, into a precise syntactic representation. While the conceptual representation involves concepts, fields and rhetorical relations, the syntactic representation of the text involves less abstract elements, like words and sentences. Section 4.1 introduces the structures manipulated by the microplanner, as well as the constraints involved in the transformation from conceptual to syntactic representation. Then I present algorithms for lexicalisation and aggregation in Sections 4.2 and 4.3. The referring expressions generation module is treated separately in Section 4.4. Finally, I give an overview of techniques for realisation in Section 4.5.

4.1 Structures and constraints

The input of the microplanner is the output of the document planner. Therefore it is a document plan, which is typically a tree whose internal nodes are rhetorical relationships or structuring elements and leaves are messages (Reiter and Dale, 2000). Depending on the level of granularity of the document plan, messages can themselves be seen as simple attribute-value pairs or more complex graph structures (Nogier and Zock, 1992).

The output of the microplanner is a precise representation of the sentences of the text. This representation is often a syntactic tree structure based on, or at least inspired from some linguistic theory, but it could also be more basic representations, like orthographic strings or a relatively flat template representation. A flat output representation means less decisions to be made by a potential realiser module and shifts the boundary between microplanning and realisation. Several situations in between are possible. In this chapter, I assume the standard situation where the output of the microplanner is a syntactic

tree, and where all decisions concerning morphology and function words are undertaken by a realisation module.

4.1.1 The structures

The output structure of the microplanner embodies several choices that have to be made in order to express the document plan as surface text. These choices can be expressed as follows:

- What are the sentence boundaries ? The sentence is the main organisational unit for the microplanner. Actually the job of the microplanner could be seen as designing sentences (microplanning is sometimes called sentence planning). The difficulty here is to determine what is the right amount of information to convey in each sentence.
- Which words should be used ? Words are the atomic pieces of the output structure, and the microplanner needs to determine which words to use in each sentence. Each word or group of words may include some structure itself, depending on the language and the syntactic category. For instance, a French noun will always have a gender associated with it.
- What is the tense and mood of each sentence ? More generally, several choices about the form of each sentence or phrase can be made. For instance, some information may be written either as an independent clause or a subordinate clause.
- In what order the words should be arranged inside each sentence ?

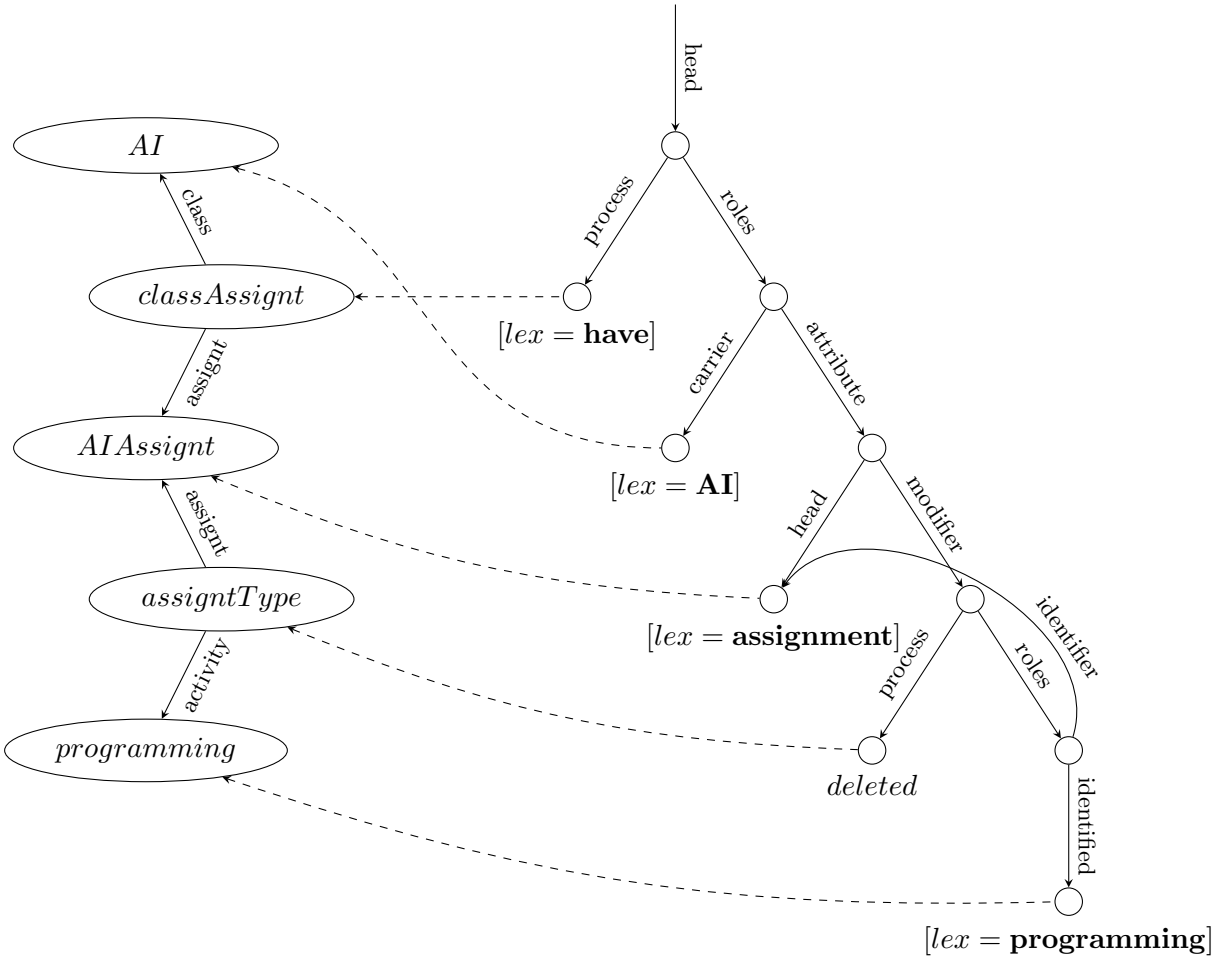
These choices are interdependent. For instance, if a message is expressed as a subordinate clause rather than a sentence, which is a choice about the form of the clause associated with the message, then the boundaries of sentences are impacted. Other choices may convey information implicitly, impacting the number of words needed to convey the information. The order of the words is also generally heavily impacted by the mode of the sentences. The dependencies between different choices are represented by the constraints which apply on the sentence planning process.

Figure 4.1 shows examples of structures for the input and the output of a microplanner. Although the representations vary, matrix representation in one case and graph representation in the other, the example structures are relatively similar. The structure underlying the matrix representation is a tree or a graph, depending on whether the matrix contains shared references or not. The second example shows a graph structure for

$$\begin{bmatrix} type : RainEventMessage \\ period : \begin{bmatrix} month : 05 \\ year : 1996 \end{bmatrix} \\ day : \begin{bmatrix} day : 27 \\ month : 05 \\ year : 1996 \end{bmatrix} \\ rainType : heavy \end{bmatrix}$$

$$\begin{bmatrix} type : PPSAbstractSyntax \\ head : \textbf{fall} \\ features : [tense : past] \\ subject : \begin{bmatrix} type : PPSAbstractSyntax \\ head : \textbf{rain} \\ features : [definite : false] \\ modifier : \textbf{heavy} \end{bmatrix} \\ modifier : \begin{bmatrix} type : PPSAbstractSyntax \\ head : \textbf{on} \\ object : \begin{bmatrix} type : ReferringNP \\ object : \begin{bmatrix} day : 27 \\ month : 05 \\ year : 1996 \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Example of message (input of the microplanner) and its associated proto-phrase specification (output of the microplanner), reproduced from Reiter and Dale 2000. The corresponding text is: "*Heavy rain fell on January, 27th*". This is an example of coarse grained representation, with a message associated as a whole to a proto-phrase specification.



Example of mapping between a conceptual graph (input of the microplanner) and a simplified syntactic structure (output of the microplanner), adapted from Elhadad et al. 1997. The corresponding text is: "*AI has programming assignments*". This is an example of fine grained representation, with each unit of the message being mapped to small syntactic structures which compose the representation of a clause.

FIGURE 4.1: Example of input and output structures for the microplanner. The representation can either be coarse grained or fine grained.

both the input and the output. Aside from the nature of the structures, tree or graph, all labels are either application or theory dependent. For instance, the labels of the output structure of the second example are associated with the FUF realiser (Elhadad, 1993).

An important property of the structures used by a microplanner is their granularity. The granularity of the structures corresponds roughly to the size of the text associated with the smaller conceptual units manipulated as such by the microplanner. For instance, if the smaller conceptual units of the input are messages which are associated with whole sentences, then the representation is said to be coarse grained. If on the other hand, the microplanner maps precise concepts to words or group of words, as in the second example, then the representation is said to be fine grained.

4.1.2 The constraints

Constraints decide, among all the combinations of words, sentences, etc., which ones are valid representations of the document plan (and possibly rank them). We can group constraints into four broad categories: semantic, grammatical, pragmatic and stylistic constraints.

4.1.2.1 Semantic constraints

Semantic constraints ensure that all the information contained in the document plan, and only this information, is expressed by the syntactic representation of the text. Semantic constraints are directly dependent on the document plan, and more generally on the conceptual level of representation. For instance, if the document plan contains a message describing the color of a car, then the text should contain at least a word for the car and a word for the color: if the car is black, then a semantic constraint could be that the words “car” and “black” appear in the text, with “black” referring to “car”. This constraint may include several surface forms, like “The car is black.”, “the black car”, “the car, which is black by the way”, etc..

Most of the semantic constraints involved in microplanning are of the form “this concept can be expressed by these words”. Therefore, they could be seen as simple set membership constraints, defining a one-to-many mapping between conceptual forms and syntactic forms. However this is generally not enough to describe the interface between the concepts and the syntactic forms. The first problem is that the mapping between conceptual and syntactic forms is generally many-to-many, i.e. a single expression may

(1) Wall Street indexes *opened* **strongly**.
(*time* in verb, **manner** as adverb)

(2) Stock indexes **surged** *at the start of the trading day.*
(*time* as prepositional phrase, **manner** in verb)

(3) The Denver Nuggets beat the Boston Celtics **with a narrow margin**, 102-101.
(game result in verb, **manner** as prepositional phrase)

(4) The Denver Nuggets **edged out** the Boston Celtics 102-101.
(game result and **manner** in verb)

FIGURE 4.2: Floating constraints. Examples from the stock exchange market and basketball games description domains, reproduced from Elhadad et al. 1997

convey multiple meanings. Second, some concepts may sometimes be realized as sentences and sometimes as words in different syntactic roles. Therefore the mapping is generally not a simple concept to sentence or concept to word mapping, but something in between. This kind of variability is sometimes referred to as "floating constraints" (Elhadad et al. 1997, see Figure 4.2).

Along with these mapping difficulties, there may also be some subtle interdependencies between the words chosen for a particular concept, and the other concepts in the document plan. For example, the word "love" generally implies an interaction between two human beings, while the word "like" is more general. If a concept can be expressed by one or the other, then the chosen verb may sometimes be incompatible with the other concepts in the document plan (e.g. if the entity which loves is not human). Another example from (Danlos, 1987) is the verb "assassinate", which should be used to describe the event of killing only when the killed is a famous person (in news reports). In both cases, the word chosen is dependent on properties at the conceptual level (the humanity or fame of the participants), and these properties are not directly associated with the core concept (loving or killing).

Last but not least, are semantic constraints which involve long range dependencies and domain knowledge. For instance, when choosing words to refer to an entity in the document plan. A reference can be ambiguous, depending on the previous words used to refer to this entity or other ones. An ambiguous reference prevents the transmission of the information. Therefore there is a constraint on the expression of the meaning of a referent, which involves other entities in the document plan, and possibly implicit knowledge about these entities. This particular problem is typically treated separately in a referring expression generation module.

4.1.2.2 Grammatical constraints

Grammatical constraints define, in a particular language, which sentences are valid and which ones are not, independently from their meaning. There are several theories which result in different kinds of grammars. Each theory makes assumptions about the underlying structure of text (word types and properties, constituents or dependencies, etc.). The structures manipulated by the microplanner, and the constraints attached to them, are therefore dependent on the theory which has been chosen to represent the syntactic structure of the text. Grammatical constraints are theoretically independent of the domain of application. Moreover they are often represented explicitly in a grammar. Therefore this is probably the type of constraint which is the most likely to be reused from one application to the other. The catch is that the constraints are still dependent on the representation used for the syntactic structure of the text.

4.1.2.3 Pragmatic constraints

Pragmatic constraints are constraints relative to the environment and situation in which the NLG system is. These constraints may involve several external factors:

- Who the system is talking to. For instance, (Mahamood and Reiter, 2011) use affect modelling to adapt the discourse in very affect intensive environments. Another example is (Janarthanam and Lemon, 2014) who adapt the generation of referring expression to the level of expertise of the reader.
- The media. This may include restrictions on the size of the output text. The text may also be embedded in complex graphical representation, like annotations in graphs (Mahamood et al., 2014).
- Any other external factor. There could be for instance a requirement imposing dynamically a particular level of detail on the text.

4.1.2.4 Stylistic constraints

Stylistic constraints are preferences which do not impact the informational content of the text or its grammaticality. A text which does not satisfy stylistic constraints should still be understandable and grammatically correct. Yet stylistic constraints may be almost mandatory. For instance, let's take the example of repetitions. A general simplistic rule about repetitions might be something like: "Thou shall not repeat the same word twice". Failing to satisfy this constraint may result in unacceptable texts in some situations. For

example, generating the description of a car without taking care of repetitions might end up in texts such as the following:

> *The car is black. The brand of the car is Peugeot. The car has GPS, automatic transmission and cruise control. The car costs $15,000. The car is good.*

While being perfectly understandable and grammatically correct, this text is just not acceptable for a car selling application, because it fails to apply the simplest stylistic constraints. Like repetitions, some stylistic constraints are generally applicable, but are often hard to formalize. Sometimes repetitions are needed, sometimes not. The line between texts which are "styled" and those which are not is often difficult to draw.

Style constraints may apply on about any choice point in the microplanner (Paiva and Evans, 2005) and may be dependent on both internal or external factors. For instance, an application generating text for particular media, like graph annotations (Mahamood et al., 2014) or SMS, may apply some stylistic constraints (independently from the size requirements). More generally application domains often impose some preferences over word choices or sentence structures. Other internal factors may include personality or psychological models (Mairesse and Walker, 2011).

4.2 Lexicalisation and aggregation

Several independence assumptions can be made in order to modularize the microplanner. The independence assumption which is probably the most current is between referring expressions generation on one side and the rest of the decisions, corresponding to lexicalisation and aggregation in the task-based pipeline architecture, on the other. Referring expression generation is treated in Section 4.4. This section and Section 4.3 present techniques for solving the lexicalisation and aggregation tasks, whether this particular modularisation is used or not. This section focuses on the task-based pipeline architecture.

The task-based pipeline architecture splits the creation of the output structure into two independent tasks (ignoring for now referring expressions generation): lexicalisation and aggregation (Reiter and Dale, 2000). The principle of this modularization is to separate between the creation of a draft structure, which is a textualisation of the document plan, and an optimization phase which optimizes the fluency and readability of the text, while (usually) keeping the same informational content.

4.2.1 Lexicalisation

The lexicalisation module maps the messages of the document plan to sentence specifications (or proto-phrase specifications). This mapping is typically done using templates. Each message is associated with one or more templates, and the lexicalisation module chooses a template for each message, while instantiating the variable inside each template using the properties of the messages. The instantiation of the variable may be handled by specific algorithms, or generic ones based on the types of the properties of the messages and a given property-to-variable correspondence.

A simple way to choose between concurrent templates for a message is to simply pick one at random. However, there are often many constraints which rule over the possible choices. Therefore choosing at random is often not possible, without breaking some constraints. A typical implementation is then a conditional mapping, which can rule out templates based on internal and external factors.

The complexity of the lexicalisation module also heavily depends on the granularity of the internal representation (see Figure 4.1). Fine grained representations will necessitate a mapping between smaller units with many dependencies between the individual units. The dependencies between small conceptual or syntactic units is added to the complexity of the mapping between conceptual and syntactic units. In this case, some kind of of advanced search algorithm is often needed. Handling "floating constraints", which amounts to implement a many-to-many mapping between conceptual and syntactic units can also complexify the lexicalisation module (see Elhadad et al. 1997 for an example of algorithm that handle this case).

4.2.2 Aggregation

Once the lexicalisation module has selected the structures associated with the messages in the document plan, the job of the aggregation module is to remove redundancies and to optimize the fluency of the text. While the lexicalisation module typically works at the sentence level, the aggregation module works with multiple sentence representations, merging them together and potentially changing the order of presentation.

The configuration of an aggregation module is centered around the definition of aggregation operators, or rules[1]. An aggregation rule is a transformation rule which detects a pattern in the given structures and applies a transformation when this pattern is detected (Callaway, 2003). Different sets of operators, arranged in different categories,

[1]Some authors use the term "revision" instead of aggregation. See (Shaw, 1998) for a discussion on the difference between the two terminologies.

have been proposed (Robin, 1994, Shaw, 1998, Dalianis, 1999, Callaway, 2003). As an example, (Reiter and Dale, 2000) differentiate between different kinds of aggregations based on their complexity. We can group aggregation rules into four categories, from the simpler to the more elaborate:

- Simple conjunction, which concatenates two independent sentences, potentially using a connective such as "and", or "but".
- Conjunction via shared participants. Merges two sentence with the same main predicate that share a syntactic constituent, such as the subject of the verb.
- Conjunction via shared structure. Merges two sentences which do not necessarily share the main predicate.
- Syntactic embedding. Merge syntactic structures, potentially by modifyin the nature of the elements which are merged.

Aggregation, like other microplanning modules, involves both the conceptual representation and the syntactic representation. Aggregation is subject to constraints in both domains. Constraints on the syntactic structure of the text ensure that the structures are still valid sentences once they have been merged. This involves low level structures, like punctuation, or some addition to the structure, like connectives. In order to enforce the constraints on the syntactic structure, only relatively shallow syntactic structures are needed. However, aggregation rules are subject to more subtle, semantic constraints. In this case, a deeper structure for the syntactic level, potentially with some semantic information, is often needed. These semantic constraints prevent to reuse all aggregation rules from one domain to another. Some reusability can be achieved by reusing only parts of the aggregation operators (either the detection part or the action part, see Callaway 2003). A generic algorithm for aggregation is possible, but some assumptions on the syntactic structure have to be made (Harbusch and Kempen, 2009).

4.3 Semantic aggregation and composition-based methods

NLG systems may not use any module dedicated to aggregation, and yet achieve results similar to what can be done with an aggregation module. The modularization between lexicalisation and aggregation solves the microplanning problem by drafting the output structure, and then optimize this structure. But the choices handled by the aggregation module may be handled by different modules, either during document planning or lexicalisation, by shifting the complexity upward. There are three broad categories of

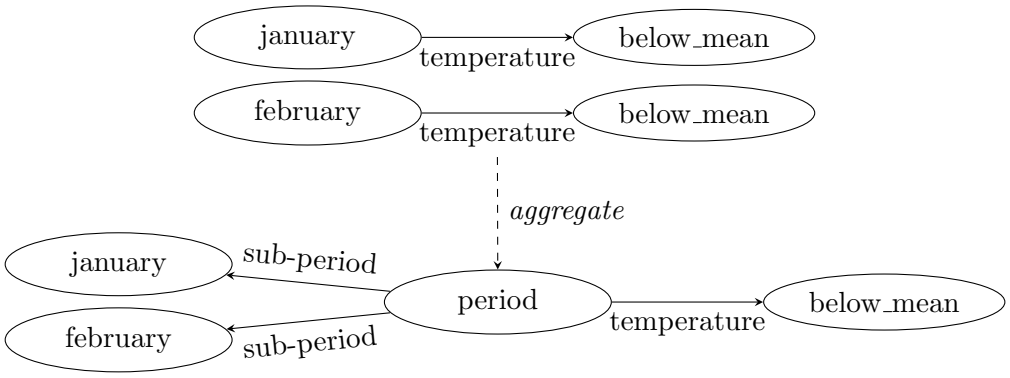


FIGURE 4.3: Example of semantic aggregation corresponding to the aggregation of "January was colder than average" and "February was colder than average" into "January and February were colder than average". The two month concepts are embedded into an abstract period concept, which can then be textualized. Semantic aggregation only manipulates symbols at the conceptual level and no syntactic structure is involved.

alternative methods to (syntactic) aggregation: semantic aggregation, complex mapping and composition.

4.3.1 Semantic aggregation

Semantic aggregation, as opposed to syntactic aggregation, is focused on packing the information at a conceptual level. One of the choices made by the aggregation module, is whether any particular redundancy should be removed or left as is. The problem of redundancy and information packing is distributed among document planning and microplanning. Therefore depending on the situation it may be more appropriate to handle redundancies during document planning. For instance, most conjunctions by shared participants (or shared structure) could be handled at the conceptual level. Figure 4.3 shows an example of aggregation that uses only symbols at the conceptual level. In this case, since the conceptual representation can change dynamically depending on the possible aggregations, the lexicalisation module must be adapted too, in order to lexicalise all possible input concepts.

4.3.2 Complex mapping and composition

One may also remove some redundancies by using a more complex mapping between conceptual and syntactic structures. Usually, each message or concept is associated to a syntactic structure in a rather straightforward manner. For example, if a concept is used twice in the document plan, then each version of the concept will be mapped to its own syntactic structure. This could be seen as a one-to-one or one-to-many mapping between conceptual and syntactic structures. On the other hand, if one was

to decide that two concepts were to be associated to the same output structure, then the redundancy would disappear automatically. A many-to-many mapping also includes cases where output structures express the meaning of several input structures. Here the complexity is shifted from the aggregation rules to the lexicalisation module, which must handle more constraints in order to produce a valid mapping between input and output structures (Elhadad et al., 1997).

Independently from the redundancy problem, the choices relative to the informational content of the text and the choices relative to its fluency can be done jointly, using composition of small syntactic structures instead of aggregation of coarse grained structures. For instance, the configuration of the lexicalisation module could include different syntactic realisations with different syntactic roles for a message, such as the verb "to increase" and the noun phrase "the increase" for the concept of augmentation. These realisations could then be used in different contexts to produce "The price of the car increased", or "There was an increase in the price of the car". Similarly, one may associate both a clause form and a subordinate clause form to a message and use either one depending on the situation, thereby removing the need for embedding aggregation rules. Instead of using aggregation, which transforms a draft structure, a composition method controls the composition of structures using constraints to build directly the final structure. Composition of structures is often used when the system uses a linguistic formalism. For instance, the system G-TAG (Danlos, 2000, Meunier, 1997) uses composition of TAG trees that represent fine grained syntactical constructs.

The difference between aggregation and what we call here the composition method can be understood from a combinatorial optimization perspective, as the difference between classical search and local search (see Section 2.3). Messages or concepts can be viewed as the variables of the problem, and the sets structures that lexicalise them as the the domains of the variables. Constraints on the compositions of variables then define the set of valid output structures. This setup corresponds to the classical search approach. On the other hand, aggregation rules define transformations from syntactic structures to other syntactic structures. Therefore they act as local moves[2] that transform solutions to the lexicalisation problem into other "close by" solutions. This set up corresponds to local search.

Although classical search and local search are different methods, they do not define search spaces that are different in essence. For complex problems, it is actually often the case that both classical search and local search are used in combination on the same search space, a classical search being used to provide a heuristic solution which serves

[2]In a search space, a solution is close to another solution if one can be transformed into the other by a small number of transformations, called local moves. Exploring a search space by listing solutions close to an initial solution using local moves is called local search (see Section 2.3.1).

as the starting point of a local search. So the difference between an architecture using an aggregation module and one that uses a complex lexicalisation module (potentially with semantic aggregation), is more a matter a perspective on the problem, rather than contradicting assumptions on what the problem is. However this claim should be tempered. It is not clear how all kinds of aggregation rules could be expressed by different kinds of choices or constraints, particularly if they involve reordering and long range aggregations. Also, different perspectives on the same problem can make a huge difference when it comes to the configuration of the system, and the two approaches are not equivalent on that matter.

The position taken in this thesis is to use semantic aggregation along with a composition method (see Chapter 8) for microplanning. While aggregation is well suited in some situations, using it as a default technique for microplanning doesn't seems right, as it forces a particular algorithmic approach where it doesn't seem to be needed. Moreover, a composition-based approach is more in line with what is done during the document planning and realisation phases, and allows a better unification of the text generation process (see Chapter 5).

4.4 Referring expression generation

The generation of referring expressions is usually handled once the structure of the sentences has been decided (Reiter and Dale, 2000). This independence assumption is justified by the fact that it is generally possible to restrict the set of possible referential expression to the ones with the syntactic category of noun phrase. By fixing the syntactic category of referential expressions one can build sentences without knowing exactly what the lexicalisation of the different entities will be, making the decisions of a lexicalisation and aggregation module independent from the referring expressions generation module. However, the reverse is generally not true. Referring expressions are very dependent on the context in which they are generated. For instance, the syntactic structure of previous utterances, and the particular lexicalisation of previously generated referential expressions may impact the generation of a new referential expression. This makes the referring expression generation module dependent on the decisions of previous modules.

Inside the referring expression generation module, we can distinguish between two kinds of decisions: whether to use a pronoun or a definite description, and when the choice is to use a definite description, which one to select. These choices are embodied by a *pronominalisation strategy* and a *disambiguation strategy*.

4.4.1 Pronominalisation strategy

A simple example of pronominalisation strategy is to use a pronoun if the previous sentence has mentioned the entity we want to refer to, and to use a definite description otherwise (Reiter and Dale, 2000). This method, however, is rather limited and usually results in ambiguities, like in the following example:

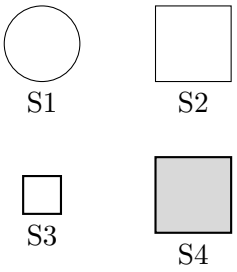
We have selected a blue car and a red car for you. **It** *(the blue car) has an automatic transmission.*

A notion of *salience* (i.e. most noticeable or important) is necessary to distinguish between entities that can be pronominalized and the others. Frameworks like the centering theory (Grosz et al., 1995), can be used to compute the salience of the different previous entities in the discourse, in order to decide whether pronominalisation should be used or not. Computing the salience requires information about the syntactic structure of previous utterances, for instance the syntactic role of the different entities. Once the salience of previous entity has been calculated, one can decide to use a pronoun only if the entity to express is the most salient of its type.

4.4.2 Disambiguation strategy

Once it has been decided that a pronoun should not be used, the next step is to build a definite description. The problem here comes from potential *distractors* that can make the reference to an object ambiguous. Figure 4.4 shows an imaginary visual set up, where we have to refer to geometrical shapes. It is clear from this set up, that referring to an object as "the square" is ambiguous. The job of the disambiguation strategy is to find the set of properties that unambiguously refer to the object we want. For instance, if we want to refer to the shape S4, a possible definite description could be "the filled square", or "the thick large square".

Here the model of the problem is rather simple. One need to search through the set of all possible sets of properties that can be used to refer to the object, and select one that unambiguously refer to it. The fact that a set of properties refers to an object in an unambiguous manner is derived from a property matrix such as the one in Figure 4.4: a set of properties is unambiguous iff one single row contains all the properties in the set. For instance, the sets $\{square, thick, large, filled\}$, $\{square, thick, large\}$, $\{thick, large\}$, $\{square, filled\}$, etc., all unambiguously refer to the shape S4. One



	type	thickness	size	filled
S1	circle	thin	large	empty
S2	square	thin	large	empty
S3	square	thick	small	empty
S4	square	thick	large	filled

FIGURE 4.4: Example of ambiguous definite description generation problem. We have a visual scene with several geometrical shapes (on the left), and we can refer to these entities using their properties (table on the right).

possible (inefficient) algorithm is to brute force breadth-first[3] search the set of sets of properties of our entity, until one is found to unambiguously refer to it.

The set of properties that describe our entity should unambiguously refer to it, but it should also be as small as possible. These two *desirata* correspond to the Gricean Maxims of Quantity (Grice et al., 1975):

1. Make your contribution as informative as is required.
2. Do not make your contribution more informative than is required.

It rules out strategies that systematically take the full set of properties of the object to refer to it. On the other hand, systematically generating the shortest possible unambiguous description is both computationally hard (c.f. the breadth first algorithm above) and not necessarily "human-like". A good trade-off is a heuristic search on the model of Algorithm 1.

Algorithm 1: Basic referring expression generation algorithm. Uses a heuristic function *nextProperty* to select the next property to include in the description until the description is unambiguous.

```
function getDescription(entity, context)
    D ← ∅
    while D is ambiguous given the context do
        p ← nextProperty(entity, context)
        remove distractors that do not have property p from the context
        add p to D
    end
    return D
end
```

This algorithm is a depth-first heuristic search on the set of possible description: the description is built iteratively by adding one property at the time to the description;

[3]Here, a breadth-first search builds all sets of one property, then all sets of two properties, then all sets of three properties, etc., until it finds a solution.

and at each step, the property to add to the description is given by a heuristic function. Changing the heuristic function gives different behaviours. Some example heuristic functions are:

- Take the property that maximizes the number of distractors removed from the context (Dale, 1989). This corresponds to a greedy search driven by the objective of a minimal description.
- Use a preference order on the properties of the entity. At each iteration take the first preferred property in the list of properties that exclude at least one other entity from the context (Dale and Reiter, 1995). The preference order is given by the designer of the system, depending on the domain of the application.

In the implementation and setup presented in Part II, I assume that referring expression generation is handled by the existing Yseop technology, in a manner similar to Algorithm 1.

4.5 Realisation

The role of the realisation module varies, depending on the structures it is expected to realise. The input structures of a realiser may range from a semantic or thematic description of the clauses to generate, to surface strings with almost no syntactic information. Different levels of abstraction for the input structure means different kinds of decisions to be made:

- Decide word order. The input structure may contain the words contained in each sentence, but let the realiser put them in order. The order generally depends on the diathesis of the sentences (e.g. active or passive). It may also depend on more subtle rules, like rules for ordering modifier in "the beautiful big red car".
- Apply Inflections. Apply the morphological rules to the words depending on their properties, like number or gender (e.g. verb conjugation).
- Decide whether to include non-mandatory words or not (e.g. "the car over here" and "the car that is over here").
- Map semantic representations to syntactic representations, if the input is described using semantic entities and relations.

The possible choices and constraints that apply on the problem are often represented using a grammar in a linguistic formalism. For instance, the realiser FUF (Elhadad and Robin, 1996) is based on functional unification grammars and KPML (Bateman, 1997) on systemic-functional grammars. The generation of surface strings from an input structure using a grammar uses classical search algorithms, like chart generation (Kay, 1996), which corresponds to a breadth-first search through possible textualisations of the input.

Precise rules that generate all the possible correct sentences and only them are hard to come up with, especially if some pragmatic or stylistic constraints come into play. Some realiser, like HALogen (Langkilde-Geary and Knight, 2002) or OpenCCG (White et al., 2007) use an "overgeneration and ranking" method. The principle of this method is to use a relatively small set of grammar rules that loosely define the set of possible sentences, but allow to generate invalid or awkward sentences. Then, for a particular input, the set of all possible sentences for this input is generated and ranked using a statistical language model. The output of the realiser is then the best sentence according to the language model. In order to keep all the possible sentences for a particular input into memory, compact data structures, like lattices or forests (sets of trees with shared nodes) are used. A typical language model for ranking sentences is a simple n-grams model.

Other realisers assume an input[4] at a rather low level of abstraction, and only assume the role of checking for valid syntactic structures and doing some inflections/reordering operations. An example of such a realiser is simpleNLG (Gatt and Reiter, 2009), and its extensions for French (Vaudry and Lapalme, 2013), German (Bollmann, 2011) and Brazilian (De Oliveira and Sripada, 2014).

Like for referring expression generation, in the implementation and setup presented in Part II, I assume that realisation (mainly morphology related operations, like inflections, contractions, etc.) is handled by the existing Yseop technology.

4.6 Summary

The microplanner takes as input a document plan and outputs a specification for sentences in natural language. It involves a lot of constraints that range from conceptual or application specific ones to general grammatical rules and fuzzy stylistic concerns. In the task-based view of microplanning, these constraints are handled by three different modules, namely lexicalisation, aggregation and referring expression generation.

[4] In the case of simpleNLG, which is a JAVA library, the term "input" is a bit fuzzy, since the structures are provided by the library and used at wish anywhere in the program.

Between these three tasks, referring expression generation is probably the most studied as an independent, well defined module. Some systems, in particular those based on linguistic formalisms, do not make a clear distinction between lexicalisation and aggregation and compose complex structures together to build the output in one go. In these systems, the constraints usually handled by the aggregation component are somehow handled by the document planner and/or the lexicalisation module, shifting the complexity of the system upward in the pipeline. However, given the rather different perspectives on the problem, it is hard to make a precise comparison between the two approaches.

The three task-based modules use rather different approaches to solve their particular problem. Lexicalisation uses simple templates and mappings, with potentially some rules for automatically mapping elements or filling templates. Aggregation uses transformation operators and solves an optimisation problem, which could be seen as a local search in the space of sentence structures. Referring expression generation often uses classical search algorithms to disambiguate references, and rules for computing the salience of elements. Non task-based methods usually use classical search methods for exploring the space of valid structures, using some kind of grammar. This method is also used for grammar based realisation modules, potentially combined with a statistical language model for ranking solutions. Because there are realisers of different complexity, the frontier is a little blurry between what should be handled by the microplanner and what should be handled by the realiser.

There are several parts of a microplanner that can be reused from one application to the other. For instance, grammars encode the grammatical rules of a language once and for all, and can be reused in any context. Some aggregation rules are also applicable in most cases, and referring expression disambiguation can be applied on any object with properties. The developer in charge of configuring a microplanner usually has access to some libraries in a general purpose programming language, which provide algorithms and predefined behaviours and give access to grammars. The knowledge needed to configure a microplanner depends, as for a document planner, on the complexity of the structures manipulated. Systems based on a particular formalism need knowledge about this formalism to be used efficiently. It is also necessary to understand the purpose of the different modules/algorithms, and more abstract and powerful mechanisms often need more skills and knowledge to be used.

Chapter 5

Using ACG for natural language generation

In Chapters 2, 3 and 4, I gave an overview of the problems encountered in natural language generation and the principal methods used to solve them. Among all the possibilities, the final choice for achieving the initial goal of the thesis has been to implement the ACG formalism. In this chapter I elaborate on the reasons of this choice. Section 5.1 first reminds some basic software quality concepts, in relation to the goal of the thesis. Then Section 5.2 shows how an ACG based architecture answers the problematic. Finally Section 5.3 compares this solution with other approaches introduced in the previous chapters.

5.1 Software quality in an industrial context

Software quality is a vague notion. It is generally expressed as a set of desirable properties that a system should have. In the case that interests us, a list of desirable properties of a system could be[1]:

- Correctness: the ability of a system to perform the task it has been created for.
- Extendibility: the ease with which a system may be modified as a result of a change of specification.
- Reusability: the ability of the system to be reused, in whole or parts for new applications.

[1]The definitions are from Meyer, 1988.

- Compatibility: the ease with which the system can be combined with other software.
- Robustness: the ability of a system to function in abnormal or unusual conditions.
- Ease of use: the ease of learning how to use a system, prepare the input data, operate the system, interpret the result and recover from usage errors.
- Efficiency: the appropriate use of hardware resources.
- Verifiability: the ease of creating test or proof procedures for validating the correctness of the system.

Most of these properties only make sense for an evolving system, in a context where people use it and adapt it to their needs over months or years. A NLG system for a particular application may only be concerned with correctness and efficiency, but a NLG framework embedded in a rich environment should be concerned with all of these properties. They embody somehow the vague notions of simplicity and quality of a system, as opposed to the notions of complexity and poor design. These desirable properties are achieved through a good modular design. Modules allow to encapsulate dependencies into well defined regions of a program and to control precisely the flows of information in the system. A general rule of thumb is to have only a few, easily identifiable modules, with as few interfaces as possible between them, the interfaces being as weakly coupling as possible.

A perfect design is of course not possible, the reason being that some of the desirable properties of a system are incompatible and impose trade-offs. A trade-off which seems particularly important in the design of a NLG framework is the one between extendibility and reusability on one side and ease of use on the other. The problem is the following: if one wants to build a powerful system, which includes many constraints and is at the same time easily extendible and has reusable component, one is bound to use very abstract concepts and complex theories. Abstractness is the natural answer for the combined pressure of the complexity and the concision requirements. However abstract concepts are also more difficult to use, and require a longer training in order to be usable in practice. This training time is a major obstacle to the scalability of the framework, and should be left to a minimum.

The position on this trade-off adopted in this thesis is to maximize extendibility and reusability, at the expense of usability. The strength of a linguistic approach to NLG, embodied here by an ACG based architecture, is to provide a unified view of NLG, with only a few key, abstract concepts. While this approach positively impacts extendibility and reusability, it also impacts negatively the “ease of use” of the system, as one needs

to master very abstract mechanisms to develop within the framework. This position is justified by the fact that ACG is used as a kernel, on which different less abstract systems can operate. For instance, one could decide to implement a simple template-based system using this kernel, the template based system would be as easy to use as any other. The important thing is to be able to build different systems based on different assumptions in a unique, efficient framework.

5.2 Using ACG for natural language generation

Linguistic approaches to NLG, like MTT (see Section 2.2 and Figure 2.2), split the generation process between different levels of abstraction. Each theory defines its own levels of abstraction (although some consensus exist for the main levels), using grammars that encode the different structures at each level and their interactions. Some theories concentrate on a single level of abstraction, like syntactic theories, or rhetorical structure theories, while other describe relations between levels, like the syntax-semantic interface, or the relations between the height levels of the MTT. Pretty much all formal linguistic theories have been used at some point or another to generate text, even sometimes in an industrial context (Coch, 1996). The problems often encountered by linguistically motivated architectures is that an advanced linguistic knowledge is needed to operate the system and that implementations are often slow, compared to template-based approaches.

The common elements in the different linguistic theories have recently pushed the field toward more and more abstract formalisms, which can represent several other formalisms and apply generic algorithms for parsing or generation. Examples of such abstract formalisms include Hyperedge Replacement Grammars (Feder, 1971, Pavlidis, 1972, Drewes et al., 1997) and Abstract Categorial Grammars (De Groote, 2001). By taking a theory neutral stance, abstract formalisms open the way to the formalisation of very different theories, or even pragmatic approaches, within a single coherent architecture.This allows to separate more clearly between computational concerns and linguistic or ergonomic issues, and to optimize the system independently of the chosen formalism. This allows to overcome the speed issue usually associated with linguistically motivated systems.

5.2.1 Theory neutral abstraction levels and transformation of structures

Like RAGS, ACG relies on a generic low-level data type in order to represent different high-level constructs. An abstraction level is a set of these low-level structures. The

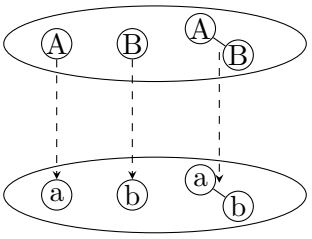


FIGURE 5.1: Each level of abstraction is a set of structures. In each set, structures can be composed to build bigger structures. A mapping between the atomic structures of each set recursively defines a mapping between the two sets.

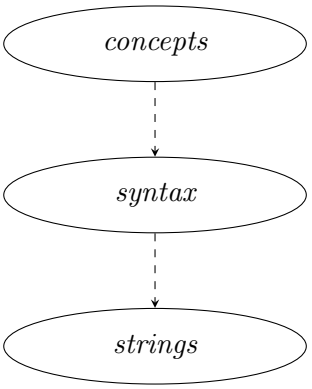


FIGURE 5.2: A Pipeline of structure transformations, with three levels of abstraction and two transformations. *concepts* contains conceptual structures, *syntax* syntactic structures and *strings* output strings. Generation is done by transforming structures from the *concepts* level of abstraction to the *strings* level of abstraction.

set is defined recursively from a few atomic structures that can be composed together to build bigger structures. Two abstraction levels can be connected through a mapping between the atomic structures of each level. This defines a morphism between the two sets of structures, which allow to convert any structure from one of the abstraction level to its image(s) in the other abstraction level (see Figure 5.1).

Levels of abstraction can be stacked to form a pipeline. Then using the mappings between the different levels, one can transform a structure from any level to a structure in another level[2]. This mechanism can be used to generate text, by transforming structures at a high level of abstraction (for instance conceptual representations) into structures at a lower level of abstraction (see Figure 5.2). The fact that each level of abstraction uses low-level theory neutral structures allows to use a single algorithm for every step of the generation process, which implies a great lot of simplifications and optimizations.

To summarize, the features of an architecture based on ACG are:

[2]Note here that a transformation may refer to the computation of the image or the inverse image of an element by a functional mapping (see Chapter 6).

- Theory neutral structures organized into different levels of abstraction. Levels of abstraction can be used to represent structures like syntactic trees or rhetorical structures.
- Levels of abstraction are connected by mappings, which constitute the interfaces between the different levels.
- Generic operations to transform structures of one level of abstraction into structures of another level of abstraction.

This responds to the goals of this thesis as follows:

- The NLG process is normalized using a low-level data type, a standard interface throughout the system, normalized transformations. This simplifies the implementation of the system and has a significant impact on the extendibility, reusability and maintainability of the system.
- ACG has enough expressivity to perform document planning and microplanning.
- Many existing linguistic resources can be represented using ACG.
- The separation of the computational and linguistic concerns makes it more efficient in practice and allows to use it in production systems.

5.3 Comparison with other approaches

As the ACG based architecture responds to practical problems in a rich environment, it is best compared with pragmatic approaches to NLG, like the task-based pipeline architecture and the RAGS project.

5.3.1 Comparison with the task-based pipeline architecture

As an exercise for comparison purposes, the task-based pipeline architecture could be described in terms of transformations of structures. Figure 5.3 shows an example architecture which could be used to simulate the task-based architecture. Based on these levels of abstraction, we can interpret the different modules of the task-based pipeline architecture as follows.

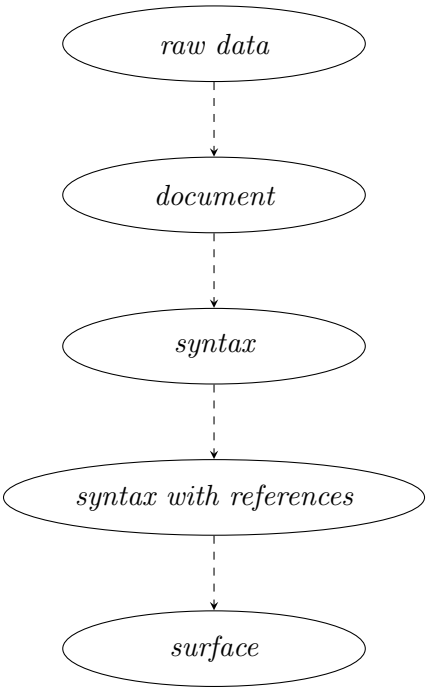


FIGURE 5.3: Possible levels of abstraction for the task-based pipeline architecture. The *raw data* level represents the input (database, signal, etc.), the *document* level contains document plans, the *syntax* level proto-phrase specifications or other syntactic constructs, the *syntax with references* level contains the same syntactic constructs, but augmented with the textualisation of referents, and the *surface* level contains the output strings.

5.3.1.1 Document planning

The overall document planning process could be seen as building a structure in the *document* level of abstraction. This operation could be done in different ways. For instance as a single interleaved content selection and document structuring process.

Another possibility is to consider content selection as a transformation from a set of structures in an external source, the *raw data* level of abstraction, into another set of structures in the *document* level, as shown in Figure 5.3. The resulting set of structures can then be composed (potentially using rhetorical relationships) in order to build a document plan.

Since the content selection process is relatively ill defined, in practice it is safer to assume that this operation is done by an external system (i.e. not with the same structure transformation or generation process than the other modules). However document structuring can be expressed naturally as a structure generation process in a level of abstraction (or eventually, as a transformation between two levels of abstraction, see Chapter 7).

5.3.1.2 Microplanning

Lexicalisation is naturally expressed by a transformation of document plans of the *document* level, into syntactic structures. Aggregation on the other hand is a bit different. Once the structures have been transformed from the *document* level to the *syntax* level, the aggregation module aggregates the resulting structures into new syntactic structures. The couple lexicalisation/aggregation can therefore be seen as a transformation from *document* to *syntax*, which first finds a heuristic solution, and then does a local search inside the the *surface* abstraction level from this heuristic solution.

This interpretation of the aggregation module shows a bias of the task-based architecture toward a specific method of resolution of the problem of microplanning. By choosing to represent a step of a particular method (local search) as a module, the architecture rules out other methods and mixes modules describing static mappings with modules describing dynamical processes, which is not ideal.

Another interpretation of the aggregation module within an architecture based on levels of abstraction would be to see the presence of an aggregation module as a property of the mapping between the *document* level and the *syntax* level. If the mapping is a straightforward one-to-one mapping between the structures in both levels, then there is no aggregation. If, however, several structures in the *document* level happen to be mapped to a single one in the *syntax* level, then the system can be said to perform aggregation[3]. Interpreting aggregation this way allows to handle cases were aggregation is distributed over several levels of abstraction. This still allows to use local search techniques, but the decision would then be only motivated by computational concerns.

Finally, the referring expression generation module transforms the structures of the *syntax* level to structures in the *syntax with references* level by mapping referents to their textualisation. The transformation is of course non-trivial and involves long range dependencies and domain knowledge. It is not clear yet how one could use the same transformation process for this module and the other modules. Since the technology used for prototyping the architecture already includes a referring expression generation module, this issue has been left for future research.

5.3.1.3 Realisation

The realisation process is a transformation from the *syntax with references* level to the *surface* level. Like referring expression generation, the Yseop technology already includes

[3]We take some liberties here with the ACG formalism for the purpose of the discussion. It is not clear how this could actually be achieved in the proposed framework. In the current implementation of the architecture, a composition-based method is used, which sidesteps the issue.

a realisation module. The details on how the transformation from the *syntax with references* level to the *surface* would be done in practice using the proposed architecture have been left for future research[4].

5.3.2 Comparison with RAGS

The proposed architecture shares with RAGS the principle of formalisation and normalisation of the data types. However, unlike RAGS, it takes a strong position on the way to articulate the different modules and to process the information. This position has the advantage of simplifying the implementation and allows for better optimizations, which is an important feature to have in an industrial contexts. The downside is that it makes the architecture less flexible than RAGS.

[4]Using a grammar based realiser, the problem would be relatively straightforward, since ACG allows to represent many existing formalisms. However the main issue here is performance, and a lot of not-so-linguistic concerns, like special symbols, dates and numbers generation, special punctuation, bullet lists, etc.. Therefore an implementation is needed to validate the concept in practice.

Part II

Abstract Categorial Grammars and Natural Language Generation

Chapter 6

Abstract Categorial Grammars

The formalism of Abstract Categorial Grammars (ACG, De Groote 2001) is a formalism based on typed λ-calculus (see for instance Barendregt and Barendsen, 1984) which generalizes several other formalisms, like context-free grammars, or mildly context sensitive formalisms. Section 6.1 presents the ACG formalism in details. Then Section 6.2 presents the representation of the TAG formalism in ACG, which is used for developing linguistic resources (see Chapter 8).

6.1 Definitions

6.1.1 The signatures

An abstract categorial grammar (De Groote, 2001, Pogodalla, 2009) defines two languages called the *abstract language* and the *object language*. Each language is a set of typed λ-terms defined by a *higher-order signature* $\Sigma = (A, C, \tau)$, where:

- A is a finite set of atomic types.
- C is a finite set of constants.
- $\tau : C \to \mathscr{T}(A)$ is a function associating a type built on A to every constant in C. The set of types built on A, $\mathscr{T}(A)$, is defined recursively as follow:
  - If $a \in A$, then $a \in \mathscr{T}(A)$;
  - If $\alpha, \beta \in \mathscr{T}(A)$, then $(\alpha \to \beta) \in \mathscr{T}(A)$.

Each type in $\mathscr{T}(A)$ can be assigned an *order*. The order $o(\alpha)$ of a type α in $\mathscr{T}(A)$ is defined recursively as follows:

- $o(\alpha) = 1$ if $\alpha \in A$
- $o(\alpha \to \beta) = max(o(\beta), o(\alpha) + 1)$

The set of λ-terms built on Σ, $\Lambda(\Sigma)$, is defined recursively as follows:

- If $c \in C$, then $c \in \Lambda(\Sigma)$ (constant).
- If $x \in X$, then $x \in \Lambda(\Sigma)$, where X is a set of variables (variable).
- If $x \in X$, $t \in \Lambda(\Sigma)$ and x is a free variable (see below) of t, then $(\lambda x.t) \in \Lambda(\Sigma)$ (abstraction).

  Any variable which appears in an abstraction (here x) is sayed to be *bound*. Any variable which is not bound is sayed to be *free*. For instance, in the term cxy, the variables x and y are free, and in the term $\lambda x.cxy$, x is bound and y is free.
- If $t, u \in \Lambda(\Sigma)$, then $(tu) \in \Lambda(\Sigma)$ (application).

To be more precise, abstract categorial grammars only use *linear* λ-terms, and possibly *almost linear* λ-terms. A linear λ-term is also called a non-erasing, non-duplicating λ-term. A λ-term of the form $\lambda x.t$ is linear if and only if the variable x appears exactly once in t. A λ-term of the form $\lambda x.t$ is almost linear if and only if x appears exactly once in t or x appears several times in t and has an atomic type. In other words, almost linear terms can only duplicate variables of atomic type. The relation between λ-terms and their type is defined by a set of inference rules:

- $\vdash_\Sigma c : \tau(c)$ (constant)
- $x : \alpha \vdash_\Sigma x : \alpha$ (variable)
- $\dfrac{\Gamma, x : \alpha \vdash_\Sigma t : \beta}{\Gamma \vdash_\Sigma (\lambda x.t) : (\alpha \to \beta)}$ (abstraction)
- $\dfrac{\Gamma \vdash_\Sigma t : (\alpha \to \beta) \quad \Gamma \vdash_\Sigma u : \alpha}{\Gamma \vdash_\Sigma (tu) : \beta}$ (application)

The first rule defines the type of a constant as the type which is assigned to it by the function τ of the signature. The second rule simply states that variables have a type. The third rule defines the type of an abstraction (a λ-term of the form $\lambda x.t$) as $\alpha \to \beta$, where α is the type of the variable (x) and β is the type of the body (t) of the abstraction. Finally, the fourth rule defines the type of an application (a λ-term of the form tu), where the left-hand side (t) has a complex type of the form $\alpha \to \beta$ and the right-hand side (u) has the type α, as β.

6.1.2 The lexicon

A *lexicon* $\mathscr{L} : \Sigma_1 \to \Sigma_2$ between two higher order signatures $\Sigma_1 = (A_1, C_1, \tau_1)$ and $\Sigma_2 = (A_2, C_2, \tau_2)$ is a mapping from λ-terms built on Σ_1 to λ-terms built on Σ_2. It is defined as a pair $\mathscr{L} = (F, G)$ such that:

- $F : A_1 \to \mathscr{T}(A_2)$ is a function that associates a type of Σ_2 to every atomic type of Σ_1. The extension of F to all types in $\mathscr{T}(A_1)$ is noted $\hat{F}$.
- $G : C_1 \to \Lambda(\Sigma_2)$ is a function that associates (almost linear) λ-terms built on Σ_2 to the constants of Σ_1. The extension of G to all λ-terms in $\Lambda(\Sigma_1)$ is noted $\hat{G}$.
- The type of the image of a constant c by G is the image by $\hat{F}$ of the type of c, $G(c) : \hat{F}(\tau(c))$.

6.1.3 Abstract categorial grammar definition

Finally, an ACG is a quadruple $\mathscr{G} = (\Sigma_1, \Sigma_2, \mathscr{L}, s)$, where Σ_1 and Σ_2 are the higher-order signatures of the abstract language $\mathcal{A}(\mathscr{G})$ and object language $\mathcal{O}(\mathscr{G})$ respectively, $\mathscr{L}$ is a lexicon from Σ_1 to Σ_2 and s is a type of Σ_1 called the *distinguished type* of the grammar. The terms of the abstract language are required to be of type s:

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

The λ-terms of the object language must have an inverse image by $\mathscr{L}$ in $\mathcal{A}(\mathscr{G})$:

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

The order of an ACG is the maximum order of the types assigned to the constants of the abstract language. The order of a lexicon $\mathscr{L} = (F, G)$ is the maximum order of the images of its abstract atomic types by F. An ACG $\mathscr{G}$ of order n with a lexicon of order m is denoted $\mathscr{G}(n, m)$.

6.1.4 Composition of grammars

Two abstract categorial grammars $\mathscr{G}_1$ and $\mathscr{G}_2$ may be composed in three different way:

- $\mathscr{G}_1$ and $\mathscr{G}_2$ both have the same abstract signature.

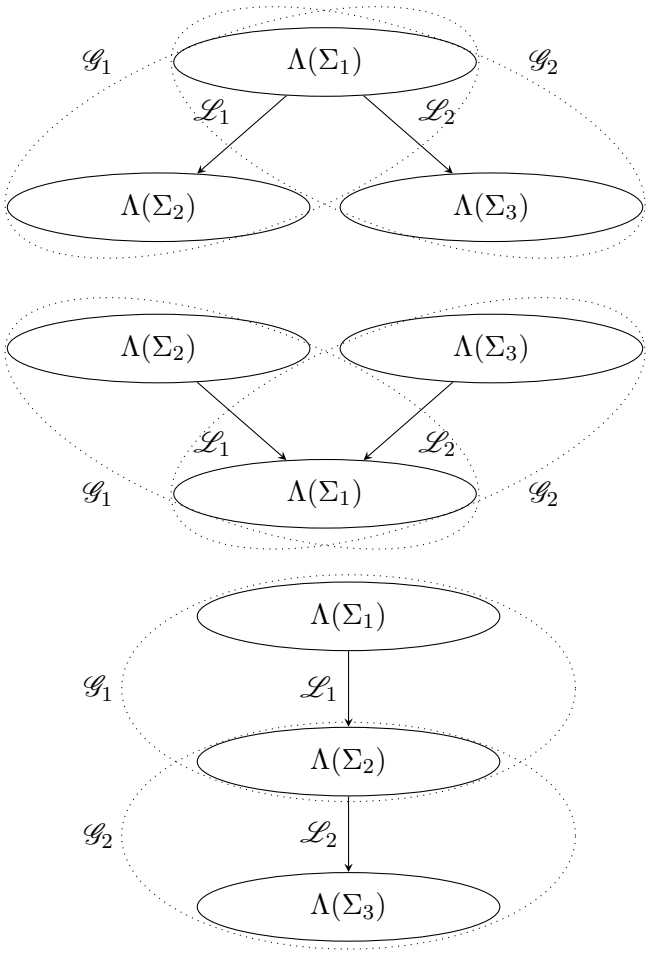


FIGURE 6.1: The three different ways to compose abstract categorial grammars together: either by shared abstract signature (top), by shared object signature (middle), or by the abstract signature of one being the object signature of the other (bottom).

- $\mathscr{G}_1$ and $\mathscr{G}_2$ both have the same object signature.
- The object signature of $\mathscr{G}_1$ is the abstract signature of $\mathscr{G}_2$ (or the object signature of $\mathscr{G}_2$ is the abstract signature of $\mathscr{G}_1$).

The three possibilities are depicted in Figure 6.1.

6.2 Tree adjoining grammars as abstract categorial grammars

The linguistic resources developed in parallel of the ACG implementation presented in Chapter 8 use Tree Adjoining Grammars (TAG, Joshi 1985) for representing the syntactic level of the text (see Section 8.3). This choice has been made for various reasons:

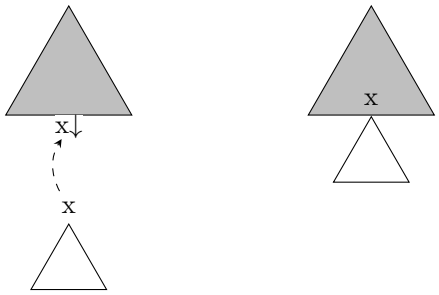


FIGURE 6.2: The TAG substitution operation. A non-terminal leaf node marked with the substitution symbol ↓ is substituted by a tree whose root is the same non-terminal symbol (here the non-terminal symbol is x).

Firstofall, TAG is a popular formalism in NLG, with several existing microplanner systems using it (Danlos, 2000, Gardent and Kow, 2007), including in an industrial context (Meunier, 1997, Danlos et al., 2011). Moreover, TAG is one of the formalism for which the encoding in ACG has been described in details (De Groote, 2002, Pogodalla, 2009). G-TAG (Danlos, 2000), an extension of TAG specialized in NLG has also been encoded in ACG (Danlos et al., 2014, Maskharashvili, 2016)[1]. Finally, there are existing grammars for different languages (for instance X-TAG, XTAG Research Group 2001, for English) which can be converted into the ACG formalism and used in our system. Grammars such as X-TAG cover many linguistic phenomenons, which ensure us that the microplanning module can represent the texts that we might want to produce in realistic applications. Section 8.3.3 shows some examples of textual variations of a conceptual representation using TAG, which are typically hard to represent with template based approaches (though even more elaborate syntactic constructs are possible).

6.2.1 Introduction to TAG

Like context-free grammars, TAG uses a set of non-terminal symbols and a set of terminal symbols. However, unlike context free grammars, the basic unit of TAG is a *tree*. The nodes of a tree are non-terminal symbols and the leaves are either terminal or non-terminal symbols. Trees do not need to contain a terminal symbol. A tree which contains at least a terminal symbol is called a *lexicalised* tree, and the terminal(s) *lexical anchor(s)*. By extension, a TAG containing only lexicalised tree is called a lexicalised TAG, or LTAG.

There are two basic operations which allow to combine trees, namely *substitution* and *adjunction*. Figure 6.2 shows the substitution operation, which allow to substitute a non-terminal leaf node by another tree with the same non-terminal at the root. The

[1]The linguistic resources presented in Section 8.3 do not use G-TAG, however it probably should (see the perspectives in Section 9.4).

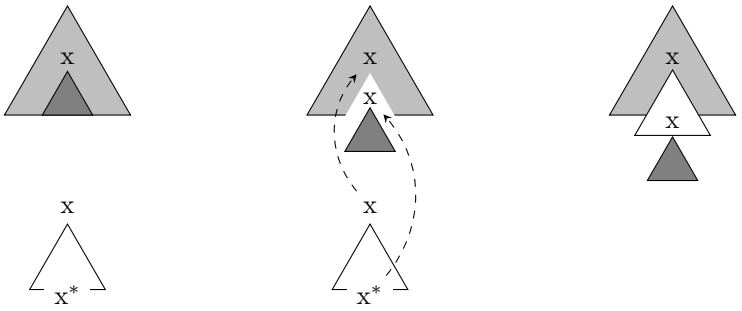


FIGURE 6.3: The TAG adjunction operation. A tree is inserted inside another tree. A β-tree with a non-terminal symbol (here x) at the root and the same non-terminal symbol at a leaf node marked with the adjunction symbol $*$ (the foot node) replaces a node with the same non-terminal symbol inside another tree. First the tree which receives the adjunction is split at the node with the non-terminal symbol (middle). Then the β-tree is inserted at the location of the split and the detached subtree is substituted at the foot node of the β-tree.

nodes where substitutions can happen are marked with the symbol $\downarrow$. Figure 6.3 shows the adjunction operation, which allows to insert a tree inside another one. Trees which can be adjoined to other trees are called *auxiliary trees* or β-trees. Non-auxiliary trees are called *initial trees*. The auxiliary trees all contain a leaf called the *foot node* of the tree, which is the same non-terminal symbol than the root node of the tree, marked with the symbol $*$. This node indicates the substitution site for the subtree of the tree on which it is adjoined (see Figure 6.3). The sequence of substitutions and adjunctions which lead to particular tree is called a derivation, and is represented in the form of a *derivation tree* (see Figure 6.4).

6.2.2 Encoding TAG with ACG

In the ACG representation of TAG (De Groote, 2002, Pogodalla, 2007), the abstract signature represents derivation trees and the object signature the trees themselves. The object signature defines a set of trees with no constraints, and the abstract signature controls the possible substitutions and adjunctions. The initial and auxiliary trees are defined in the lexicon.

More formally, let $G = \{\Sigma, N, I, A, S\}$ be a tree adjoining grammar, where Σ is a set of terminal symbols, N is a set of non-terminal symbols, I a set of initial trees, A a set of auxiliary trees and S the distinguished non-terminal symbol. The ACG $\mathscr{G} = \{\Sigma_1, \Sigma_2, \mathscr{L}, s\}$ which defines the same tree language than G is defined as follows.

Let $\Sigma_1 = \{A_1, C_1, \tau_1\}$ be the abstract signature, with A_1 being a set of atomic types, C_1 a set of constants and τ_1 a function mapping constants from C_1 to types built on A_1 defined as follows:

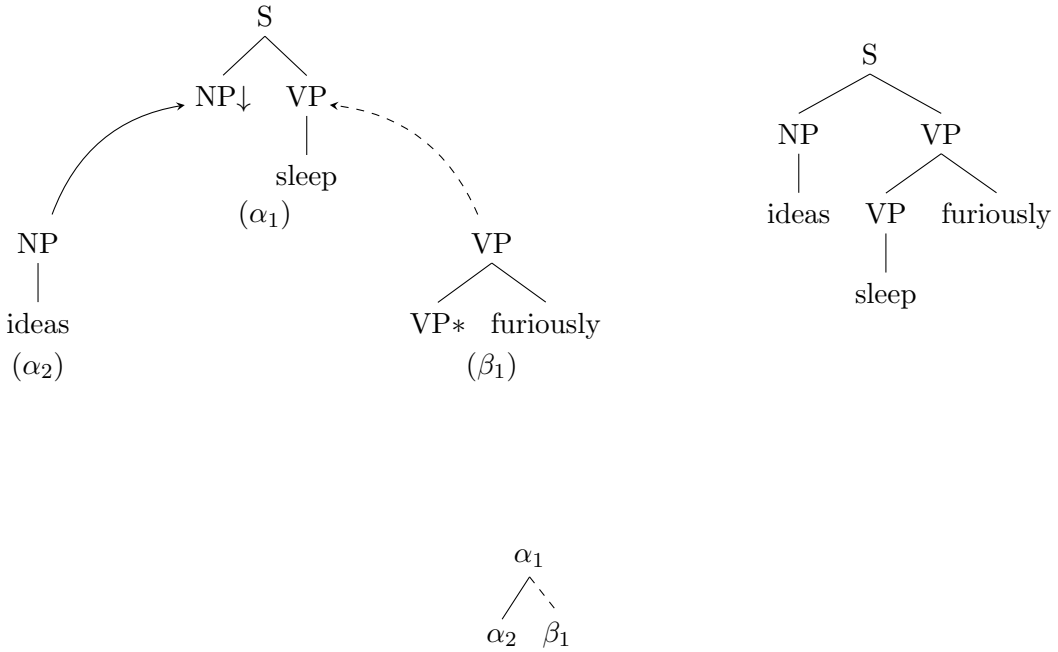


FIGURE 6.4: The sequence of substitutions and adjunctions which have been performed in order to build a tree is represented by a derivation tree. Top-left are the trees which are combined in order to give the tree on the top-right. Plain arrows represent substitutions and dashed arrows are adjunctions. The sequence of substitutions and adjunctions to perform in order to obtain the tree on the top-right is encoded in the derivation tree at the bottom. The nodes of a derivation tree are initial or auxiliary trees and the arcs represent substitutions and adjunctions operations (plain arc for substitution and dashed arc for adjunction).

- A_1 is the union of N (the non-terminal symbols) and N', the non terminal symbols with subscript A (for "Adjunction"). For instance, if $N = \{S, NP, VP\}$, then $A_1 = \{S, NP, VP, S_A, NP_A, VP_A\}$
- C_1 is the union of I and A (i.e. the set of all trees, initial or auxiliary), plus a set of identity tree for each non-terminal symbol in the grammar $\{I_{\alpha_1}, \ldots, I_{\alpha_n}\}$.
- The type of a constant encodes all possible substitutions and adjunctions operations on the tree it represents:
  - The type of a constant c_{T_i}, with T_i being an initial tree is:

$$c_{T_i} : \beta_{1A} \to \cdots \to \beta_{mA} \to \alpha_1 \to \cdots \to \alpha_n \to \gamma$$

    The atomic types $\beta_{1A} \to \cdots \to \beta_{mA}$ represent adjunctions on interior nodes (nodes with non-terminal symbol) of T_i, where m is the number of interior nodes which can receive an adjunction in T_i. The atomic types $\alpha_1 \to \cdots \to \alpha_n$ represent substitutions on leaves of T_i marked with the symbol $\downarrow$, where n is

the number of such leaves. Finally γ is the non-terminal symbol at the root of T_i.

– The type of a constant c_{T_a}, with T_a being an auxiliary tree is:

$$c_{T_a} : \beta_{1A} \to \cdots \to \beta_{mA} \to \alpha_1 \to \cdots \to \alpha_n \to \gamma \to \gamma$$

The β_i and α_i subtypes represent adjunction and substitutions respectively, like for T_i. The auxiliary tree in itself is represented as a function which takes a tree and returns a tree. The two last subtypes $\gamma \to \gamma$ encode this representation. γ is the non-terminal symbol at the root and foot of T_a. One could see the first γ as the foot node, and the last γ as the root node, the effect of the function being to substitute the argument at the foot node and return the resulting tree (see the description of $\mathscr{L}$ below).

Now let $\Sigma_2 = \{A_2, C_2, \tau_2\}$ be the object signature, with A_2 being a set of atomic types, C_2 a set of constants and τ_2 a function mapping constants from C_2 to types built on A_2 defined as follows:

- A_2 contains only one atomic type τ (which stand for "tree").
- C_2 contains the terminals of G (Σ), plus a set of constants for each non-terminal in N (see below).
- The the type of the constants associated with terminals is τ. The set of constants associated with a non-terminal α is:

$$c_{\alpha i} : \underbrace{\tau \to \cdots \to \tau}_{i \text{ times}} \to \tau$$

for $1 \leq i \leq k$, where k is the maximal number of children of the interior nodes labelled with α in all the trees of G. For instance, if the non-terminal symbol S is used in different trees with one, two or three child nodes, then C_2 will contain $c_{S_1} : \tau \to \tau$, $c_{S_2} : \tau \to \tau \to \tau$ and $c_{S_3} : \tau \to \tau \to \tau \to \tau$.

The lexicon $\mathscr{L}$ between Σ_1 and Σ_2 is defined as follows:

- $\mathscr{L}(\alpha) = \tau$ for every non-terminal symbol (without subscript) in A_1 and $\mathscr{L}(\alpha_A) = \tau \to \tau$ for all non-terminal symbols with subscript A in A_1. So initial trees are simply interpreted as trees (type τ) and auxiliary trees are interpreted as functions taking a tree and returning a tree (type $\tau \to \tau$).

$$c_{sleep} : S_A \to VP_A \to NP \to S \qquad c_{furiously} : VP_A \to VP \to VP$$

$$\mathscr{L}(c_{sleep}) = \lambda s_a vp_a np.s_a(S_2\ np\ vp_a(VP_1\ sleep)) \qquad \mathscr{L}(c_{furiously}) = \lambda vp_a foot.vp_a(VP_2\ foot\ furiously)$$

FIGURE 6.5: The representation of trees in ACG. The first line shows an initial tree and an auxiliary tree, the second line the constants in the abstract signature that represent the trees and on the last line the image of these constants by $\mathscr{L}$ in the object signature. The variables representing adjunctions (s_A, vp_A) are second order variables of type $\tau \to \tau$, and the variable representing substitution (np) is a first order variable of type τ. The variable $foot$ of type τ represents the argument of the auxiliary tree and placed at the foot of the tree.

- The image of the constants of C_1 are built by combining the constants in C_2 in order to form the corresponding tree, and every argument of the tree, representing substitutions and adjunctions, are represented as first order variables and second order variables for substitution and adjunction respectively (see Figure 6.5). The identity trees $\{I_{\alpha_1}, \ldots, I_{\alpha_n}\}$ are all mapped to the identity $\lambda x.x$ (they represent the "no adjunction" operation, used to fill the gaps in the arguments of the trees when combining them).

Finally, the distinguished type s of $\mathscr{G}$ is the distinguished non-terminal symbol S of G. This completes the definition of a TAG using ACG.

6.2.3 Encoding strings with ACG

It is also possible to represent the string language associated with a tree adjoining grammar using ACG (De Groote, 2002, Pogodalla, 2007). A *string* is represented by a second order λ-term of type $o \to o$. For instance, a terminal symbol t of G (which is a string) is represented by a constant $s_t : o \to o$. Several strings can be concatenated to form a longer string. For instance, three constants s_1, s_3 and s_3 of type $o \to o$ can be concatenated to form the string: $\lambda x.s_1\ (s_2\ (s_3\ x)) : o \to o$. Strings can be concatenated using concatenation operators, which take strings (type $o \to o$) as arguments and return a string. For instance, the operator for concatenating two strings is: $\lambda xyz.x\ (y\ z) : (o \to o) \to (o \to o) \to o \to o$.

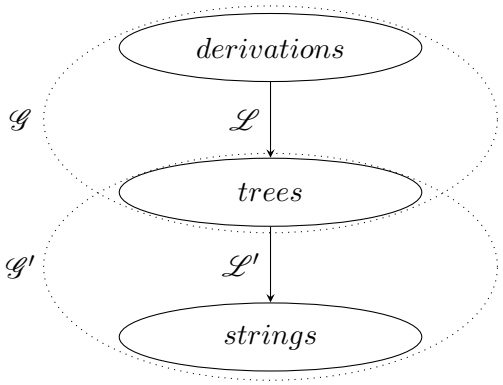


FIGURE 6.6: The composition of two abstract categorial grammars $\mathscr{G}$ and $\mathscr{G}'$ representing the tree and string languages of a tree adjoining grammar.

We can represent the string language of G by adding a new ACG $\mathscr{G}' = \{\Sigma'_1, \Sigma'_2, \mathscr{L}', s'\}$, whose abstract signature is the object signature of $\mathscr{G}$ (see Figure 6.6). The object signature of $\mathscr{G}'$, $\Sigma'_2 = \{A'_2, C'_2, \tau'_2\}$ is defined as follows:

- A'_2 only contains the atomic type o, used to build string of type $o \to o$.
- C'_2 contains a constant s_t for each terminal symbol t in C_2.
- The type of all constants in C'_2 is $o \to o$.

The lexicon $\mathscr{L}'$ of $\mathscr{G}'$ is defined as follows:

- The type τ of A_2 is mapped to the type $o \to o$.
- Each terminal t in C_2 is mapped to the constant s_t in C'_2. The other constants c_{α_i} which represent non-terminals are mapped to concatenation operators, depending on the number of arguments they have. For instance, the constants $c_{S_1} : \tau \to \tau$, $c_{S_2} : \tau \to \tau \to \tau$ and $c_{S_3} : \tau \to \tau \to \tau \to \tau$, which represent the non-terminal S with one, two and three child nodes respectively, are mapped as follows:

$$\mathscr{L}'(c_{S_1}) = \lambda x.x$$
$$\mathscr{L}'(c_{S_2}) = \lambda xyz.x\ (y\ z)$$
$$\mathscr{L}'(c_{S_3}) = \lambda wxyz.w\ (x\ (y\ z))$$

By combining $\mathscr{G}$ and $\mathscr{G}'$ we obtain the string language of G. One can then either generate strings from derivation trees or find the derivation trees of input strings.

Chapter 7

Document planning with ACG

The goal of the document planning module is to produce a precise description of the text to generate at the conceptual level, also called document plan, or document structure (see Chapter 3). This document plan may be built from many different kinds of input, depending on the application. A typical input for an industrial application can be represented as set of objects in an object oriented programming language. These objects may represent various kind of information, such as boolean values, numerical values, strings, which may represent for instance the name of the interlocutor, or more structured representations such as events, messages or rhetorical relationships. This diversity of situations of varying complexity causes some problems when one tries to build a general purpose NLG framework[1]. In the context of an ACG based NLG framework, the main issue revolves around the question of what level of abstraction should be used to represent the document planning process, and how the input information should be represented as a λ-term.

This chapter explores a general approach to document planning using ACG, which can be used in scenarios of very different complexities. This approach can be summarized as follows: The level of abstraction of document plans is represented by a signature (say Σ_{docs}), containing constants for relations and messages in such a way that the set of second order λ-terms which can be built on this signature is finite. For the input information, I distinguish between the information representing some text (e.g. a name, an event which should be textualized, a numerical value), and the information representing the context in which the text is generated (usually represented as boolean values). This distinction may be seen as the distinction between the conceptual and pragmatic aspects of the generation process. The first kind of input information is represented as constants in the signature Σ_{docs}, the second kind of information is represented as different types

[1] A NLG framework is a set of tools which are used to create NLG systems (see Chapter 1).

which can be used to select a subset of the document plans built on Σ_{docs}. The set of second order λ-term which can be built on Σ_{docs} for a particular input is then the set of document plans for this input.

Section 7.1 provides the details of the problem of document planning using ACG and the rationale for using the method described above. In Sections 7.2 and 7.3 I show how to represent schemas, a technique often used in an industrial context, using an ACG based NLG framework. Finally, Section 7.4 goes all the way to data-driven document structuring, by showing how to build RST rhetorical trees (Mann and Thompson, 1988) using ACG. This last document planning technique has been implemented in a test application presented in Chapter 9. Note that the main focus of this chapter is on the document structuring part of document planning, all data analysis and filtering being considered as handled for the most part by some external system.

Discourse modelling using ACG has been studied in (Maskharashvili, 2016) in the context of D-STAG (Danlos, 2011), an extension of TAG using SDRT. There is to my knowledge no system based on ACG using RST, as described in Section 7.4 and implemented in Chapter 9. Schemas (Sections 7.2 and 7.3) are close to context free grammars, which have been studied extensively in the context of ACG (Maskharashvili, 2016), however the processing of schemas given here differs from the usual processing of context free grammars in ACG, as they cannot be easily associated with a string language (see Section 7.1). The examples of document plans used in this chapter are deliberately simplified for the sake of the demonstration. Being able to represent the equivalent of schemas, which is the most common method for document planning, ensures that we cover the needs of most applications in practice, including complex ones. Still many of these applications do not need elaborate document planning methods. One of the main interest in using ACG for document planning in this case is to be able to use the same internal formalism for modules which are usually thought as separate (Reiter and Dale, 2000), thereby simplifying implementation and maintenance, and making a ground for more advanced methods, ACG allowing to represent existing formalisms such as RST in the context of more elaborate document planning techniques as shown in Section 7.4.

7.1 Using ACG in the context of a NLG framework

7.1.1 The static and dynamic aspects of NLG systems

A NLG framework allows to build NLG systems for a large variety of applications. A NLG system built for an application is a program which receives an input and outputs text. The range of inputs that the NLG system can receive is specific to the application.

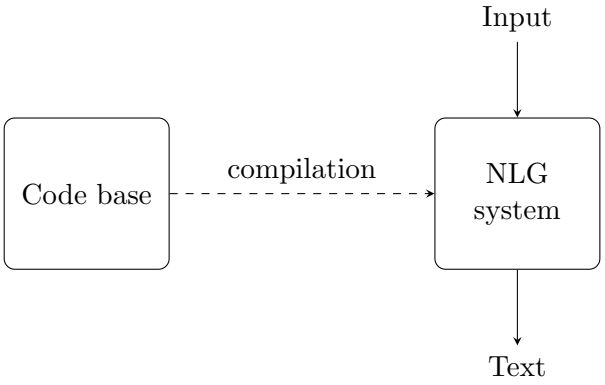


FIGURE 7.1: The general workflow of NLG systems creation. The code base specifies the set of texts that can be produced and all the operations realized by the NLG system (described for instance as functions in a general purpose programming language, such as JAVA or C++). The code base is then compiled into a program which accepts a particular input (depending on the application), and outputs texts. Definitions in the code base are said to be static definitions, while the objects created by functions in the compiled NLG system are said to be dynamically created.

The workflow for creating NLG systems using a NLG framework is described in Figure 7.1. A *code base* describes the NLG system. This code base is generally written in a general purpose programming language, such as JAVA or C++. In this case the NLG framework takes the form of code libraries and resource files, such as dictionaries (along with a dedicated programming environment, see Chapter 1). The code describing the NLG system is then compiled into a program, which can be used, for instance as a web service or as a command line tool.

It is important to distinguish between the information which is known before the NLG system is compiled, and the information which is computed each time the NLG system is called on some input. The information known before the compilation is said to be *statically defined*, or *known at compilation time*. On the other hand, information computed by the NLG system is said to be *dynamic* or *known at runtime*. A simple example is shown in Figure 7.2. This example shows the output of a NLG system which generates a commercial email for a sales campaign. The text in bold font is the part of the output which has been computed at runtime, and the rest of the text is defined statically. One can create such a NLG system by using a text template, i.e. a string with slots for variables. Here the variables are the name of the recipient of the email, the maximum discount of the sales campaign and the date of the end of the sales campaign. Once the text template is compiled, it becomes a program which accepts a string (the name of the recipient), an integer (the maximum discount) and a date (the end date, here we assume it is given in some numerical format). These three variables are used to generate dynamically text such as the ones in bold font, and the rest of the text never changes. While being simplistic, this example clearly shows the difference between the

```
Dear John,

Come at our store to enjoy an exceptional discount (up to 40% !) on all our products.
This promotion ends on Friday 25th. Hurry up !

Sincerely,

The Shop Team
```

FIGURE 7.2: Text template for a commercial email. The bold text is computed dynamically, while the rest of the text is defined statically.

static information, which is known before the NLG system is even compiled, and the dynamic information, which depends on the input, and is computed by the NLG system at runtime (here the computation mainly revolves around converting numbers for the discount and the date into strings).

7.1.2 ACG based NLG systems

Using the terminology introduced in Figure 7.1, we can describe an ACG based NLG framework as follows: The code base is a grammar, i.e. signatures and lexicons[2]. The NLG system is a program using the compiled grammar in order to convert input λ-terms, built on one of the signatures of the grammar, into λ-term built on another signature of the grammar. This conversion might be a simple mapping using one of the lexicons of the grammar, or might involve the inversion of a lexicon, in order to parse an input λ-term built on the object signature of a lexicon into a λ-term built on its abstract signature (the operation might also be a composition of mappings and inverse mappings, see Chapter 8). Since we are in the context of NLG, I assume that an ACG always contains a signature $\Sigma_{strings}$, which defines a string language (see Section 6.2.3), and that the NLG system built from an ACG is always used to translate a λ-term built on some signature in the ACG into a λ-term of Σ_{string}, this λ-term constituting the output of the NLG system.

This description constitutes what might be called a "pure" ACG based NLG framework. In practice, the code base might contain other definitions than signatures and lexicons, and the NLG system might perform other operations than converting λ-terms between signatures. For instance, the input of the NLG system might not be a λ-term, and some conversion operation might be needed, or there might be other modules which perform operation not handled by the ACG module, such as referring expressions generation.

[2]For simplicity, here and in the following, the term ACG or grammar also encompasses the composition of several lexicons.

Here I assume that the code base, containing both the ACG definitions and the other definitions is written in a general purpose programming language, such as JAVA or C++. In the following, I call the language used to write the code base the *NLG framework language*. Also, the term *NLG system* refers to the program described by the whole code base, including the code which is not specific to ACG. Saying that some computation is performed by the *wrapping NLG system* (as opposed to computation performed by the ACG module) means that this computation is described in the code base using the NLG framework language independently from the definition of the ACG.

A natural way of using ACG to build a NLG system is to define the object signature of a lexicon as the input signature (i.e. the signature on which the input λ-terms are built). The abstract signature of the lexicon represents an infinite set of structures, and the input λ-term built on the object signature is used to select a subset of these structures. For instance, for a microplanner, we might have an input signature corresponding to the semantic level of abstraction, and an abstract signature corresponding to the set of derivation trees of a natural language. A set of output texts, represented by their derivation trees, are selected using the input λ-term built on the signature representing the semantic level of abstraction (see Section 8.3). By analogy, for a document planner, we might have an input signature corresponding to the level of abstraction used for the input of a particular application (e.g. events, rhetorical relations, numerical values, etc.), and an abstract signature corresponding to document plans for this application.

This way of approaching the definition of a NLG system using ACG has been adopted for microplanning, but another approach is used in this thesis for document planning. The main problem here resides in the fact that there often is no formalised structure in the input of a NLG system. For instance, a typical input might be represented as an finite, unordered, set of variables $\{V_1, \ldots, V_n\}$ (possibly with complex domains, such as objects in an object oriented programming language). Using the approach above, we might want to build a λ-term from this input, which could then be parsed in order to find document plans associated with it. However by representing the input as a λ-term, we give it an order, or structure, which caries no meaning. Following this logic, we also need to interpret the constants of the signature representing document plans into the input signature, but this interpretation must then be coherent with the arbitrary structure given to the input, which adds to the complexity of the definitions without adding any valuable information. While in some situations, the input, or part of the input, might be given a meaningful structure, it doesn't seem to be the case for the inputs observed in many real world applications.

For this reason (see also the discussion Section 7.4.2.3), the general approach adopted for document planning in this thesis is not to select a subset of structures from an infinite

set using the input information, but to directly define a finite set of structures (i.e. document plans) which can be generated by the NLG system as the set of second order λ-terms which can be built on the input signature. The general approach is to define constants corresponding to the document structure and constants corresponding to the input information, and to generate second order λ-terms from the signature resulting from these definitions (see the examples below).

This leads us to another problem that an ACG based NLG system might face, namely the representation of input information at different levels of abstraction. This problem might arise in the context of microplanning as well as document planning and can be summarized as follow: Some of the input information might represent a portion of the output text that the NLG system must generate, but not be represented at the level of abstraction of the input signature. For instance, the input of a NLG system might contain numerical values, dates, or surface strings, that need to be represented as λ-terms in the input signature, whether the input signature represents the semantic level for the microplanner, or document plans for the document planner. Ideally, every possible input information should have its λ-term representation at the level of the input signature and its associated grammar describing the conversion of this information into text. This is easy for input information such as a strings, which only needs to be "copy-pasted", but it might become more complex when converting numerical values (e.g. an integer using the signed magnitude representation) into their textual counterparts ("one", "forty-two", etc.) or other more complex information such as dates. While interesting, this problem still requires a fair amount of research, and has been left for future studies. In this thesis I use the fact that the ACG based NLG framework is embedded into an existing NLG framework. This NLG framework already possesses efficient methods for converting such input information into text. Therefore I use placeholders for the input information which cannot be easily represented in the input signature, and let the existing NLG framework convert this information into text in a post precessing module.

As a minimal illustration of the method used for document planning and the manipulation of placeholders in the ACG module of a NLG system, we may generate the text of Figure 7.2 using basic text template structures. Let $\mathscr{G}_{email} = \{\Sigma_{templates}, \Sigma_{strings}, \mathscr{L}, S\}$ be an abstract categorial grammar, where $\Sigma_{templates}$ is the abstract signature and $\Sigma_{strings}$

the object signature of $\mathscr{L}$, and we have[3]:

$$\Sigma_{templates} = \{S, X_{name}, X_{discount}, X_{date},$$
$$t_{email} : X_{name} \rightarrow X_{discount} \rightarrow X_{date} \rightarrow S,$$
$$t_{V_{recipient}} : X_{name},$$
$$t_{V_{maxDiscount}} : X_{discount},$$
$$t_{V_{endDate}} : X_{date}\}$$

$$\Sigma_{strings} = \{o,$$
$$s_1 = \text{“}\texttt{Dear}\text{”} : o \rightarrow o,$$
$$s_2 = \text{“}\texttt{,
Come at our store to enjoy an exceptional discount (up to }\text{”} : o \rightarrow o$$
$$s_3 = \text{“}\texttt{ !) on all our products.
This promotion ends on }\text{”} : o \rightarrow o,$$
$$s_4 = \text{“}\texttt{. Hurry up !
Sincerely,
The Shop Team}\text{”} : o \rightarrow o,$$
$$s_{V_{recipient}} = \text{“}\textbf{John}\text{”} : o \rightarrow o,$$
$$s_{V_{maxDiscount}} = \text{“}\textbf{40}\text{”} : o \rightarrow o,$$
$$s_{V_{endDate}} = \text{“}\textbf{Friday 25th}\text{”} : o \rightarrow o\}$$

$$\mathscr{L} = \{S, X_{name}, X_{discount}, X_{date} : o \rightarrow o,$$
$$t_{email} : \lambda v_1 v_2 v_3 z.s_1\ (v_1\ (s_2\ (v_2\ (s_3\ (v_3\ (s_4\ z))))))),$$
$$t_{V_{recipient}} : s_{V_{recipient}},$$
$$t_{V_{maxDiscount}} : s_{V_{maxDiscount}},$$
$$t_{V_{endDate}} : s_{V_{endDate}}\}$$

At the abstract level, the structure of the document is represented by a single text template: t_{email}, which takes as parameter three input variables, $V_{recipient}$, $V_{maxDiscount}$ and $V_{endDate}$, represented by the constants $t_{V_{recipient}}$, $t_{V_{maxDiscount}}$ and $t_{V_{endDate}}$ of type X_{name}, $X_{discount}$ and X_{date} respectively. The signature $\Sigma_{strings}$ contains the predefined texts used by the template and the placeholders for the input variables[4]. Here for clarity,

[3]For the sake of concision, here and in the following definitions of signatures, I simply list the types, represented as standalone symbols, followed by constants associated with their type using the convention *constant* : *type*. In the definition of lexicons, type mappings and constant mappings are listed in this order using the convention *abstract type* : *object type* and *abstract constant* : *object constant* respectively. The convention $type_1, \ldots, type_n$: *type* means that the all the abstract types $type_1$ through $type_n$ are mapped to the object type *type*.

[4]In the definition of this signature, the notation s = “`text`” : $o \rightarrow o$ means that the symbol s is used as a placeholder for the string “`text`”, and that the string is associated with the type $o \rightarrow o$. More generally, since we are in the context of a NLG system written in a general purpose programming language wrapping the ACG, s might represent any kind of object.

I used the HTML tag `<br>` to indicate a line break. The bold text associated with the constants $s_{V_{recipient}}$, $s_{V_{maxDiscount}}$ and $s_{V_{endDate}}$ represents text generated dynamically by the wrapping NLG system, using existing functions for converting strings, numbers and dates into the right format. These texts correspond to the input $\{V_{recipient} = \text{“}John\text{”}, V_{maxDiscount} = 40, V_{endDate} = Date(day : 25, month : 12, year : 2016)\}$[5] and change for each input of the NLG system.

From this definition, the ACG module generates text by generating all λ-terms of type S in the abstract signature and getting their image by $\mathscr{L}$. In this simple example, the ACG module always generates a single λ-term: $\lambda z.s_1\ (s_{V_{recipient}}\ (s_2\ (s_{V_{maxDiscount}}\ (s_3\ (s_{V_{endDate}}\ (s_4\ z))))))$, the interesting part of the computation being the conversion of the input variable values into text, which is handled by the wrapping NLG system. In the rest of this chapter, this method for document planning, i.e. generating second order λ-terms from an input signature containing constants representing the document structure and placeholders for input variables, is used in more elaborate scenarios, up to the point where we are able to simulate the existing document planning methods used in most of the practical applications today.

7.2 Basic document structures

For a particular application, the structure of the document is often rigid and predictable. In such cases, the usual technique for representing the structure of a document in a NLG system is schemas (McKeown, 1985, see Section 3.2.1). The concept of schema is a bit fuzzy, the specifics depending on the implementation. In this section, we will first work with a very simple definition of schema, and enrich it bit by bit in the next sections in order to cover the most common usages of schemas in practice.

A schema has a name, a type and a body, which is an ordered list of schema types[6], and *messages* (see Chapter 3). The messages represent chunks of text and can be parametrized using an input variable. As an example, let's use the following schemas to describe the structure of the text of Figure 7.2:

[5] Here $Date(day : 25, month : 12, year : 2016)$ designates an object (for instance in JAVA), with class *Date* and fields *day*, *month* and *year* initialized with the values 25, 12 and 2016 respectively. Most language have their own representation for dates, which might substitute for this example.

[6] For now, the examples assume that recursive definitions, i.e. schemas containing their own type in their body, are not allowed. This case is treated separately in Section 7.3.2.1.

```
Email commercialEmail
    Greetings(V_recipient)
    Content
    Signature()
```

```
Content defaultContent
    ExceptionalDiscount(V_maxDiscount)
    DiscountLimit(V_endDate)
```

These two schemas are nested to form a tree structure. The first one gives the overall structure of an email, by dividing it between a greeting formula, a content section and a signature. The second one further divides the content section between a presentation of the discount on one side and a warning on the time limit of the discount on the other. `Greetings`, `Signature`, `ExceptionalDiscount` and `DiscountLimit`, are the messages of our document plan and are parametrized using the three input variables of our example. At this level, we do not need to define precisely what messages are. They could be implemented for instance as text templates, semantic representation or custom functions in the programming language used to implement the NLG system. We can represent the document plans defined by our schemas and messages as second order λ-terms as follows:

- The types of the schemas, and the names of the messages and input variables are used as atomic types.
- A constant is created for each schema. The return type of the constant associated with a schema is its type. The arguments of a schema are the elements of its body.
- A constant is created for each message. The return type of the constant associated with a message is the atomic type representing this message. The arguments of a message are the input variables which parametrize it
- A constant is created for each input variable, and its type is the atomic type representing the input variable.
- These types and constants constitute a signature. The document plans defined by our schemas and messages are the second order λ-terms built on this signature. If there is a *root schema*, i.e. an initial schema which is used to build all possible schemas, then the atomic type associated with this root schema defines the set of λ-terms built on the above signature which represent the valid document plans. For instance, if the root schema is the first schema in our example, then the λ-terms representing the valid document plans are of type *Email*, from the name of the root schema (see below for a complete example).

As an illustration, let's build an ACG with example textualisations[7] for our schemas and messages. Let Σ_{doc} be the signature built from our two example schemas. Σ_{doc} is defined as follows:

$$
\begin{aligned}
\Sigma_{doc} = \{ & Email, Content, \\
& Greetings, Signature, ExceptionalDiscount, DiscountLimit, \\
& V_{recipient}, V_{maxDiscount}, V_{endDate}, \\
& t_{commercialEmail} : Greetings \rightarrow Content \rightarrow Signature \rightarrow Email, \\
& t_{defaultContent} : ExceptionalDiscount \rightarrow DiscountLimit \rightarrow Content, \\
& t_{Greetings} : V_{recipient} \rightarrow Greetings, \\
& t_{Signature} : Signature, \\
& t_{ExceptionalDiscount} : V_{maxDiscount} \rightarrow ExceptionalDiscount, \\
& t_{DiscountLimit} : V_{endDate} \rightarrow DiscountLimit, \\
& t_{V_{recipient}} : V_{recipient}, \\
& t_{V_{maxDiscount}} : V_{maxDiscount}, \\
& t_{V_{endDate}} : V_{endDate}\}
\end{aligned}
$$

A simple way to associate a textualisation with the document plan defined by Σ_{doc} is to add a signature $\Sigma_{strings}$ representing text templates, and link the two signatures through

[7] Here for the sake of the illustration I use basic textualisations which corresponds to text templates, with no complex microplanning involved. Other textualisations are possible (and usually preferable), such as logical sentences, which describe the text at a semantic level and can be processed by a microplanning module (see for instance the definitions in Chapter 8).

a lexicon (say $\mathscr{L}_{doc\text{-}str}$) as follows:

$\Sigma_{strings} = \{o,$

$s_1 =$ "`Dear`" $: o \rightarrow o,$

$s_2 =$ "`,`" $: o \rightarrow o,$

$s_3 =$ "`Come at our store to enjoy an exceptional discount (up to `" $: o \rightarrow o,$

$s_4 =$ "` !)  on all our products.`" $: o \rightarrow o,$

$s_5 =$ "`This promotion ends on `" $: o \rightarrow o,$

$s_6 =$ "`.  Hurry up !`" $: o \rightarrow o$

$s_7 =$ "`Sincerely,<br>The Shop Team`" $: o \rightarrow o$

$s_8 =$ "`<br>`" $: o \rightarrow o$

$s_{V_{recipient}} = text(V_{recipient}) : o \rightarrow o$

$s_{V_{maxDiscount}} = text(V_{maxDiscount}) : o \rightarrow o$

$s_{V_{endDate}} = text(V_{endDate}) : o \rightarrow o\}$

$\mathscr{L}_{doc\text{-}str} = \{Email, Content,$

$Greetings, Signature, ExceptionalDiscount, DiscountLimit,$

$V_{recipient}, V_{maxDiscount}, V_{endDate} : o \rightarrow o,$

$t_{commercialEmail} : \lambda greetings\ content\ signature\ z.greetings\ (s_8\ (content\ (s_8\ (signature\ z))$

$t_{defaultContent} : \lambda discount\ limit\ z.discount\ (limit\ z),$

$t_{Greetings} : \lambda recipient\ z.s_1\ (recipient\ (s_2\ z)),$

$t_{Signature} : s_7,$

$t_{ExceptionalDiscount} : \lambda discount\ z.s_3\ (discount\ (s_4\ z)),$

$t_{DiscountLimit} : \lambda date\ z.s_5\ (date\ (s_6\ z)),$

$t_{V_{recipient}} : s_{V_{recipient}}$

$t_{V_{maxDiscount}} : s_{V_{maxDiscount}}$

$t_{V_{endDate}} : s_{V_{endDate}}\}$

Let $\mathscr{G}_{doc} = \{\Sigma_{doc}, \Sigma_{strings}, \mathscr{L}_{doc\text{-}str}, Email\}$ be the ACG used by a NLG system to generate emails such as the one in Figure 7.2. Like in the example of Section 7.1.2, the textualisations of the input variables use placeholders whose values are computed at runtime. The runtime computation is represented in the definitions by the calls $text(V_{recipient})$, $text(V_{maxDiscount})$ and $text(V_{endDate})$ (in the definition of the signature $\Sigma_{strings}$), where $text()$ is a function written in the NLG framework language that converts the value of

an input variable into a string. Given these definitions, a possible general algorithm for the wrapping NLG system is the following:

1. Given an input (for instance $\{V_{recipient} = \text{“}John\text{”}, V_{maxDiscount} = 40, V_{endDate} = Date(day : 25, month : 12, year : 2016)\}$), complete $\mathscr{G}_{doc}$ by replacing the calls to the function $text()$ by their dynamic value.

2. In the resulting ACG, generate all the λ-terms of type $Email$ (the distinguished type of the grammar, which represents the root schema) on the signature Σ_{doc}. In our example, there is only one λ-term of type $Email$:

   $$t_{commercialEmail}\ (t_{Greetings}\ t_{V_{recipient}})$$
   $$(t_{defaultContent}\ (t_{ExceptionalDiscount}\ t_{V_{maxDiscount}})(t_{DiscountLimit}\ t_{V_{endDate}}))$$
   $$t_{Signature}$$

   This λ-term corresponds to a document plan, so the the second step can be considered here as a document planning (more precisely document structuring) module.

3. Get the image of the generated λ-terms by $\mathscr{L}_{doc\text{-}str}$. The image of the example λ-term of the previous step gives us the text of Figure 7.2.

This example illustrates the most basic document planning technique: simply define one static document plan, which will be used for every possible input. The next step is to define a set of document plans, in which case we also need to implement some selection mechanism for choosing a subset of the possible document plans given the input information.

7.3 Complex document structures

There are two main forces driving the complexity of the structures generated by the document planner. The first one is the range of outputs it is expected to produce. This factor varies depending on the range of things the NLG system is expected to talk about and on the number of ways it is expected to talk about the same thing. The second one is the dependence of the system on the context of execution. This factor is linked to the first one, as a NLG system which need to talk about a lot of things needs a way to select somehow what it will talk about in a given situation (i.e. for a given input). But this second factor also varies on the number of interactions the NLG system is expected to have with the external world. For instance, a NLG system involved in dialogues with humans has to be very sensible to the context of execution. In this section, I

explore different methods for handling these two aspects in the case of an ACG based NLG system. The only practical way to introduce these methods is through simplified examples, but it should be kept in mind in the following that most of these methods only make sense for sizeable systems, with an evolving code base, and that their goal is to respond in the simplest manner possible to the needs which arise when building realistic complex NLG systems.

7.3.1 Describing larger sets of document plans

7.3.1.1 Synonyms

The first way to introduce variations in the document structure is to add synonyms. Synonymous elements can be substituted freely without changing the informational content of the text. From the perspective of schemas, synonyms correspond to different schemas with the same type, or to groups of messages or input variables organized into classes of objects. For instance, we could redefine the schemas of the previous section as follows:

```
Email commercialEmail
    Greetings(V_recipient)
    Content
    Signature()
```

```
Content defaultContent
    ExceptionalDiscount(V_maxDiscount)
    DiscountLimit(V_endDate)
```

```
Content reversedContent
    DiscountLimitAlt(V_endDate)
    ExceptionalDiscountAlt(V_maxDiscount)
```

Here we have one schema of class `Email` (`commercialEmail`), and two schemas of class `Content` (`defaultContent` and `reversedContent`). The schema `reversedContent` introduces two new messages: `DiscountLimitAlt` and `ExceptionalDiscountAlt`, which represent alternative phrasing for the two sentences of the content of our commercial email. Using the schema `reversedContent` instead of the schema `defaultContent` might result for example in the following text:

Dear **John,**

You have until **Friday 25th** to come at our store to enjoy an exceptional discount.
All our products are on promotion (up to **40%**) !

```
Sincerely,

The Shop Team
```

Where the two main pieces of information of the text are presented in reverse order. While this text is not exactly synonymous to the text in Figure 7.2 from a reader perspective, as long as the NLG system is concerned they are indeed synonymous, as there is no way to distinguish between the two structures at the level of schemas. In a similar manner, one may introduce variability in the potential output texts by having classes of messages, representing different equivalent phrasings for a particular piece of information, or classes for the input variables, representing different possible textualisation of the input information. For instance, the variable $V_{maxDiscount}$ may be textualized as a number or as text such that the input $V_{maxDiscount} = 40$ may be textualized as "`40`" or "`fourty`".

Synonyms have a straightforward representation in ACG. Different elements with the same class are represented as different constants with the same return type. For instance the three schemas `commercialEmail`, `defaultContent` and `reversedContent` can be represented by the following constants (in the signature Σ_{doc}):

$$t_{commercialEmail} : Greetings \rightarrow Content \rightarrow Signature \rightarrow Email$$
$$t_{defaultContent} : ExceptionalDiscount \rightarrow DiscountLimit \rightarrow Content$$
$$t_{reversedContent} : DiscountLimitAlt \rightarrow ExceptionalDiscountAlt \rightarrow Content$$

The same principle applies if we use classes of messages and classes of input variables. With the new definitions, there are now two λ-terms of type $Email$, which represent the two possible structures of our commercial email. Since there are now several possibilities when generating λ-terms, the NLG system must choose a variant for the output[8]. Here and in the following, we will assume that all the variants are equivalent and that the NLG system uses a uniform distribution to choose at random one of the possible output. This is not mandatory, in more advanced systems, one might use other distributions, for instance based on a language model.

As an alternative way of representation for synonyms using ACG, one might also modify slightly the definition of a lexicon in order to allow different textualisations for the

[8]Eventually, the NLG system might return all the possible variants. This could be interesting for instance in the context of a development environment, where the designer of the NLG system needs to know all the possible outputs of the system. In this thesis I assume that the NLG system always choose a single output text.

same constant. For instance, let $number()$ and $text()$ be two functions of the wrapping NLG system taking an integer variable and returning a string, with the first one simply converting the integer to string and the second one spelling the number (e.g. $number(40)$ = "40" and $text(40)$ = "$fourty$"). We might have two constants $t_{V^{number}_{maxDiscount}} : V_{maxDiscount}$ and $t_{V^{text}_{maxDiscount}} : V_{maxDiscount}$ in Σ_{doc} for the input variable $V_{maxDiscount}$, with $\mathscr{L}(t_{V^{number}_{maxDiscount}}) = number(V_{maxDiscount})$ and $\mathscr{L}(t_{V^{text}_{maxDiscount}}) = text(V_{maxDiscount})$. But another possibility is to have a single constant $t_{V_{maxDiscount}}$ in Σ_{doc} with two images by the lexicon: $\mathscr{L}(t_{V_{maxDiscount}}) = \{number(V_{maxDiscount}), text(V_{maxDiscount})\}$. This second notation can be viewed as a shorthand for the first one, where the two constants $t_{V^{number}_{maxDiscount}}$ and $t_{V^{text}_{maxDiscount}}$ are left implicit. While equivalent in principle, the two notations have different usages in practice. If we regroup several constants behind a single label, as in the second notation, then we cannot access the individual constants when building λ-terms in the signature, and therefore we assume that the group of constants (here $t_{V^{number}_{maxDiscount}}$ and $t_{V^{text}_{maxDiscount}}$) are perfect synonyms and can always be substituted for each other. In the first notation, we allow explicit references to either one of the constants in the group (and therefore we need to define the individual constants). Therefore the second method might be seen here as syntactic sugar for manipulating sets of constants in a simplified manner. Going further along this line of thinking, we can extend the type system of ACG with features in order to represent and manipulate sets of constants more easily.

7.3.1.2 Types with features

Adding synonyms is the basic tool for increasing the variability of the output of the document planner (and more generally the NLG system). However it does not scale very well, as it doesn't offer any tool for manipulating and structuring large sets. Such control can be obtained by extending ACG using types with features. Using types with features is not equivalent to using types without features in general, and the properties of ACG do not hold any more. For this reason we restrict ourselves here to features with finite extension only, which can always be reduced to the usual types.

Features can be viewed as a compact way of representing a set of types. An atomic type α with feature f is noted:

$$\alpha[f]$$

A feature has a (finite) extension or *domain* (i.e. a set of values). Let the domain of f be $\{v_1, v_2, v_3\}$. Each value represents a variant of the atomic type which is associated with the feature. In the case of $\alpha[f]$, the variants are:

$$\{\alpha_{f=v_1}, \alpha_{f=v_2}, \alpha_{f=v_3}\}$$

Following this principle, a constant associated with a type with feature represents a set of constants, each one being associated with a different variant of the type. If c is a constant of type $\alpha[f]$, then it represents the following set of constants:

$$\{c_{f=v_1} : \alpha_{f=v_1},$$
$$c_{f=v_2} : \alpha_{f=v_2},$$
$$c_{f=v_3} : \alpha_{f=v_3}\}$$

There may be several features associated with one atomic type. In this case the atomic type associated with the features represents a set containing one type for each element in the Cartesian product of the domains of the features. For instance, let g and h be two features with domain $\{v_1, v_2\}$. The type $\beta[g, h]$ represents the set:

$$\{\beta_{g=v_1,h=v_1}, \beta_{g=v_1,h=v_2}, \beta_{g=v_2,h=v_1}, \beta_{g=v_2,h=v_2}\}$$

The same principle applies for complex types containing several atomic types with features, i.e. such a complex types represent a set of complex types, one for each element in the Cartesian product of the domains of all the features in all the atomic types of the complex type. For instance, the type $\alpha[g] \rightarrow \beta[h]$ represents the set:

$$\{\alpha_{g=v_1} \rightarrow \beta_{h=v_1}$$
$$\alpha_{g=v_1} \rightarrow \beta_{h=v_2}$$
$$\alpha_{g=v_2} \rightarrow \beta_{h=v_1}$$
$$\alpha_{g=v_2} \rightarrow \beta_{h=v_2}$$

Using types with features allows to easily represent and manipulate sets of constants. However as we have described it, it does not do much except giving an alternative representation for synonyms as described in Section 7.3.1.1. When used in a lexicon, a constant associated with a type with feature represents a set of constants which all have the same image by the lexicon, as in the second method for representing synonyms of Section 7.3.1.1. However, types with features allow to go further and give a precise control over the sets of constants by allowing to easily *select* subsets of sets defined through types with features. The selection mechanism works by adding *constraints* to the types. The simplest kind of constraint is the equality constraint. For instance, we

can add an equality constraint $f = v_1$ to the type $\alpha[f]$, which is written as follows[9]

$$\alpha[f = v_1]$$

Since there is only one possible value for the feature f which satisfies the constraint, this type represents a set with a single variant : $\{\alpha_{f=v_1}\}$. equality constraint may also concern two different features. For instance, the type $\beta[g = h]$ represents the set:

$$\{\beta_{g=v_1,h=v_1}, \beta_{g=v_2,h=v_2}\}$$

It is also possible to add constraints in complex types. In order to link features from different atomic types, it is possible to introduce new variables as follows:

$$\alpha[g = x] \rightarrow \beta[h = x]$$

Here x is a variable introduced for the purpose of adding an equality constraint between the features g and h (the constraint $g = h$ is inferred from the constraints $g = x$ and $h = x$). Beyond equality constraints, pretty much any constraint can be used, including arithmetic constraints or global constraints[10]. For instance, another typical constraint is the set membership constraint, which allows to restrict a feature to a subset of its domain (such as in $\alpha[f \in \{v_1, v_2\}]$).

Using features in combination with constraints allows a great expressivity. Using terms from the constraint programming world (see Section 2.3.1), features (which correspond to variables) and constraints form a *search space*. Since the domains of the features are finite, any constraint can ultimately be represented as a finite set of tuples of values. This scenario is called *finite domain constraint programming*. Even being limited to finite domains, the framework of constraint programming allows to represent a large class of problems and gives us a powerful tool for representing and manipulating sets of constants.

Types with features can be used in the static definition of an ACG in order to organize the constants into coherent groups. For instance, let's take the two constants $t_{V^{number}_{maxDiscount}} : V_{maxDiscount}$ and $t_{V^{text}_{maxDiscount}} : V_{maxDiscount}$ from Section 7.3.1.1, which represent different textual variants of the input variable $V_{maxDiscount}$. We might use,

[9] I use the following method for the notation: if the feature does not appear in any constraint, then the "naked" feature is added in the brackets. If it appears in a constraint, then it is considered as introduced by the constraint and is not added independently in the bracket. Naked features and constraints are separated by commas.

[10] In this last case, it might be interesting to differentiate between the type on one side, which introduces the variables of the problem (i.e. the features), and constraints on the other. This would probably require a different notation though.

instead of the these two constants, the following definition:

$$t_{V_{maxDiscount}} : V_{maxDiscount}[format \in \{number, text\}]$$

Where $format$ is a feature with domain $\{number, text\}$. This definition combines the two variants of the input variables into one constant, parametrized by the feature $format$. Each variant still needs to have a different textualisation. To define the image of each variant by a lexicon, I will use the following notation (here with our example lexicon $\mathscr{L}_{doc\text{-}str}$):

$$\mathscr{L}_{doc\text{-}str}(t_{V_{maxDiscount}}[format = number]) = number(V_{maxDiscount})$$
$$\mathscr{L}_{doc\text{-}str}(t_{V_{maxDiscount}}[format = text]) = text(V_{maxDiscount})$$

Here the notation $c[constraint]$, where c is a constant associated with a type with feature, represents the subset of the constants represented by c where $constraint$ is a constraint added to the return type of c (i.e. $constraint$ is a restriction on the set of constants represented by c). Given these definition, we can then select different textual variants of the input variable $V_{maxDiscount}$ directly in the definition of the constants representing messages as follows:

$$t_{ExceptionalDiscount} : V_{maxDiscount}[format = number] \rightarrow ExceptionalDiscount$$
$$t_{ExceptionalDiscountAlt} : V_{maxDiscount}[format = text] \rightarrow ExceptionalDiscount$$

While this example might seem simplistic, it actually shows the usage of ACG and of the types with features in a realistic context. Indeed, the concept of format is used in many situations in practice, usually associated with low-level data types such as: integers (as numbers, cardinals or spelled out), floating point precision numbers (use the notion of precision, scientific notation, etc.), numbers associated with units (currency, time, mass, etc.), dates and many others. Therefore it is important to be able to represent and manipulate these concepts in an efficient way. The types with features allow us to build parametrized sets of constants easily and to handle these scenarios. Of course the problem of formats is only one example, and the types with features can be used in other contexts, in order to build parametrized sets of messages or schemas (see for instance the example of Section 7.3.2). In any case, it is a powerful tool for managing large sets of document structures (and text structures).

In the case of types with features, the comparison with the method of schemas is not easy. Schemas usually tend to be parametrized with dynamic parameters, where the parametrization described in this section only uses information known at compilation

time. The two methods which resemble the most to the one described in this section are probably the following:

- Schema templates (in the sense for instance of C++ templates). Some languages allow to use static parameters in order to describe generic functions or classes which can be adapted to different scenarios. Implementations of schemas in such languages can make use of these static parameters to describe different versions of a schema.
- Inheritance is the basis of the organisation of information in object oriented languages. Using inheritance in combination with schemas allows to give a compact representation of several schemas, the different classes of the inheritance tree being the different variants. The comparison however is somewhat abusive, as the specialisation of classes usually involve dynamic mechanisms, such as initialization mechanisms (e.g constructors). A full comparison of the two approaches is outside of the scope of this thesis.

As a conclusion, we remark that many languages are not designed to provide advanced mechanisms for using the information available at compilation time (such mechanisms are sometimes called metaprogramming). Using ACG, which is a declarative formalism, to describe statically the structure of our documents, gives an advantage, as it offers much of the power of a general purpose programming language (in particular when augmented with types with features), and can be computed at compilation time, leaving the dynamic part of the NLG system to the minimum necessary. It is of course not a specificity of ACG, but rather of declarative formalisms. Imperative formalism also usually provide some metaprogramming facilities. However in such languages, it is more natural (and easier) to control the flow of execution using imperative style programming. Conversely, it is not natural for a declarative formalism to control the flow of execution at runtime (for example using the cut operator in Prolog). However any practical system still needs ways of doing so

7.3.2 Improving context sensitivity

Until now, the examples only showed how to create sets of synonymous document plans. In this section I explore ways of dynamically selecting a subset of the statically defined document structures. Using schemas, the typical way of introducing context sensitivity is to use conditional statements. For instance, let's adapt the commercial email example, so it can be used to generate a SMS instead of an email:

```
Document commercialDocument(media)
    if(media = SMS)
        ShortContent(V_recipient, V_maxDiscount, V_endDate)
    else
        Greetings(V_recipient)
        Content
        Signature()
    endif
```

This schema replaces the schema `commercialEmail` of Section 7.3.1. It is parametrized by a variable `media` which contains the target media of the NLG system (which is given for instance in the input of the NLG system). This variable is used in a conditional statement in order to switch between two different structures. `ShortContent` is a message which represents a condensed version of the email. For instance, given the input $\{V_{recipient} = \text{“}John\text{”}, V_{maxDiscount} = 40, V_{endDate} = Date(day : 25, month : 12, year : 2016)\}$, the text associated with `ShortContent` could be:

John, The Shop Team invites you to an exceptional discount event (up to **40%** !) until **Friday 25th**.

The behaviour of the schema `commercialDocument` can be simulated using ACG by creating two constants, one for each structure described in `commercialDocument`:

$$t_{SMS} : ShortContent \rightarrow Document[media = SMS]$$
$$t_{email} : Greetings \rightarrow Content \rightarrow Signature \rightarrow Document[media = email]$$

The return type of these two constants is $Document[media]$, where the domain of the feature $media$ is $\{SMS, email\}$. This static definition represents the different variants of the document. A particular variant can then be chosen at runtime, by choosing to generate either λ-terms of type $Document[media = SMS]$ or $Document[media = email]$ (i.e. by choosing dynamically the distinguished type of the grammar). This decision can be taken by the wrapping NLG system, depending on the input information. This example can be generalized into the following procedure for representing dynamic variations using ACG:

- All the dynamic variants are represented statically. Each variant has a different return type in order to be able to distinguish it from the others.
- The distinguished type of the grammar is chosen dynamically and λ-terms are generated using this type. This restricts the possible λ-terms to the ones of the chosen distinguished type.

Types with features are useful in this context, as they allow to represent variants in a compact way, and to use different kinds of constraints to restrict the possible structures. For instance, if the SMS and email variants of our commercial document had the same structure, but different messages, the information about the media could be encoded into a feature and passed down to the messages as follows:

$$\begin{aligned} t_{document} : Greetings[media = x] &\rightarrow Content[media = x] \\ &\rightarrow Signature[media = x] \\ &\rightarrow Document[media = x] \end{aligned}$$

$t_{document}$ represents the schema `commercialDocument` in the event of the two variants SMS and email having the same general structure (the impacts on the other schemas and messages are omitted). Here the feature *media* is used to propagate the information about the media to the arguments of $t_{document}$. Each argument, either representing a schema or a message, is then restricted depending on the input information about the media. This information can even be transmitted deeper into the structure, to input variables. In this setup, the different variants of every input variable, message or schema can be exposed through the distinguished type of the grammar. A precise selection can then be performed by constraining the different features at runtime. As an illustration, we could imagine a complex document structure selected dynamically using the following type:

$$\begin{aligned} Document[&media = email, recipientType = customer, \\ &numberFormat = number, dateFormat = standard\text{-}US, \dots] \end{aligned}$$

With the different features impacting the document structure, the textualisation of the messages or the textualisation of the input variables representing output text. It may become cumbersome to systematically expose all the variants of all schemas, messages and input variables through the distinguished type, since it forces us to report the information about the potential variants on the types of all the constants. In some cases, it might be interesting to restrict a set of constants simply by restricting the domain of a feature at runtime. For instance, a feature *dateFormat*, associated with the types of constants representing date objects might be restricted globally, without relying on the propagation of features through the whole document structure. Another way to say the same thing is that we could remove dynamically some of the constants which have been defined statically in the ACG. We are then left with the document structures that don't contain these constants. For instance in the case of constants representing date objects, removing all the variants but the one corresponding to a particular format (say *standard-US* which would write dates in the format $mm/dd/yyyy$) results in texts

where only the chosen format is used. It is of course of the responsibility of the designer of the NLG system to make sure that there will always be at least one possible document structure left, no matter which constants are removed at runtime.

To summarize, we have to complementary mechanisms here: the dynamic selection of the distinguished type of the grammar, and the modification of the grammar itself at runtime. The selection of the distinguished type may be seen as a parametrization of the grammar, while the modification of the definitions at runtime might be useful for more transversal dynamic behaviours, such as selecting formats. Both methods play the same role than the parsing of an input λ-term for selecting a subset of an infinite set of structures in the situation described in Section 7.1.2, i.e. they select the right subset of document plans that might be generated given the input information.

7.3.2.1 Recursive definitions

Schemas often include recursive definitions, using for instance the Kleene star symbol ($*$, zero or more) or the $+$ symbol (one or more). For instance, staying on a commercial email application, part of the input could be given as a list of product descriptions, with the size of the list being known only at runtime. This could result in the following schema

```
Document commercialDocument
    Greetings(V_recipient)
    Product+
    Signature()
```

For this example, we will assume that the NLG system builds one message of class `Product` for each product description of the input. The schema `commercialDocument` simply lists the input product descriptions (there should be at least one) between a greeting formula and a signature. This schema can be represented by the following constant:

$$t_{commercialDocument} : Greetings \rightarrow Product \rightarrow Signature \rightarrow Document$$

The problem comes with the definition of the constants associated with the product descriptions of the input. We let aside for now the constraint that there should be at least one product description. For each product description P_i of the input, we could associate a constant t_{P_i} of type $Product \rightarrow Product$, and add a constant $t_{stop} : Product$ as a stop element. For instance, for the input product descriptions $\{P_1, P_2, P_3\}$, we

would have:

$$t_{P_1} : Product \rightarrow Product$$
$$t_{P_2} : Product \rightarrow Product$$
$$t_{P_3} : Product \rightarrow Product$$
$$t_{stop} : Product$$

However, there is a hidden assumption with the notation `Product+` in the schema `commercialDocument`, which is that any particular product description should appear only once in the document. In other words, when we build the structure of the document, we consume the input product description. To model the consumption of input elements, we need to use a feature and a few constraints as follows:

$$t_{P_1} : Product[set = x] \rightarrow Product[set = x \cup \{P_1\}, P_1 \notin x]$$
$$t_{P_2} : Product[set = x] \rightarrow Product[set = x \cup \{P_2\}, P_2 \notin x]$$
$$t_{P_3} : Product[set = x] \rightarrow Product[set = x \cup \{P_3\}, P_3 \notin x]$$
$$t_{stop} : Product[set]$$

The feature *set* represents the set of consumed input product description and its domain is the set of sets of product descriptions. Each input product description P_i is associated with a constant $t_{P_i} : Product[set = x] \rightarrow Product[set = x \cup \{P_i\}, P_i \notin x]$, which keeps tabs on the consumed product descriptions on one side and verifies that a product description is not used twice using the constraint $P_i \notin x$ on the other. Now the constant associated with the our initial schema can be rewritten as follows:

$$t_{commercialDocument} : Greetings \rightarrow Product[|set| > 0] \rightarrow Signature \rightarrow Document$$

The constraint $|set| > 0$ ensures that there is one or more product description listed. We can see here that we are at the limit of content selection, as one could perfectly limit the number of listed product description, thereby doing some basic filtering on the input, as follows:

$$t_{commercialDocument} : Greetings \rightarrow Product[|set| > 0, |set| < 3] \rightarrow Signature \rightarrow Document$$

In this case, a maximum of two input product descriptions can be listed, the other ones being filtered out. This however is very limited. A true filtering strategy would most probably need some mean of ranking the different elements in order to filter out only the least interesting ones (among other criteria). The analysis of content selection techniques using ACG is outside the scope of this thesis.

To summarize, the treatment of recursive definition given here relies on the dynamic definition of a search space in the way of logic constraint programming, using the input information to define constants, features and constraints which define a set of document plans. The constraints are used to keep the set of document plans finite, and it relies on the fact that the domains of the features are finite sets, which ensures that the properties of ACG still hold (all the definitions using constraints above can be rewritten using a finite number of constants and mappings).

7.4 Data-driven document structuring using RST

With the description of recursive document structures, we have now seen a sufficient panel of techniques for document structuring using ACG for simulating schemas. The general approach is to describe the set of document structures that the NLG system may produce, and select a subset of the potential document structures by restricting the choices either by removing constants or selecting dynamically the distinguished type of the grammar. Being able to simulate schemas, the default document planning technique for most NLG systems, shows that ACG can be used in the context of a NLG framework to perform document planning in a large variety of applications. By pushing the notion of consumption of input elements in order to build document structures using recursive definitions a little further, we can describe yet another method for document structuring: bottom-up, data driven document structuring. More precisely, in this section I focus on data-driven document structuring using rhetorical relationships from RST (Mann and Thompson, 1988), as it is the most popular rhetorical structure theory for NLG (Reiter and Dale, 2000).

7.4.1 Bottom-up document structuring

In the bottom-up setup, the input of the document structuring module is composed of two sets: a set of messages, and a set of rhetorical relationships between these messages. Here we assume that these two sets are computed from the input information by the wrapping NLG system (for instance in a content selection module). At this level, messages can be considered as simple symbols, which represent text spans. The goal is to build a rhetorical tree from the input relations. For instance, in the context of an online car selling application (see Section 9.3 for a detailed description of this application), we

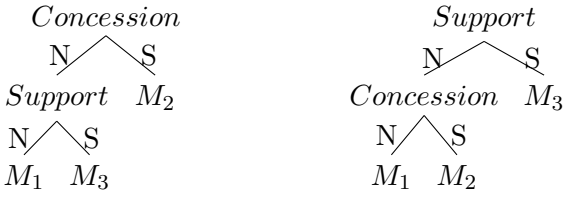
could have the following input:

$$
\begin{aligned}
messages : M_1 &\sim \text{"the power is 160hp"} \\
M_2 &\sim \text{"the car is not as powerful as you required"} \\
M_3 &\sim \text{"the car is the most powerful of its category"} \\
relations : &Concession(M_1, M_2) \\
&Support(M_1, M_3)
\end{aligned}
$$

Which could correspond to the following text:

Despite being not as powerful as you wished, this car has 160hp, which makes it the most powerful car in its category.

In the formalisation of RST (Marcu, 1996), the relations are composed together through their most salient part, the nucleus (a relation may have two nucleus, see below). The general principle is that a relation can be composed with another relation if the nucleus of the first one is either the nucleus or the satellite of the second one. In the example above, M_1 is the nucleus of both relation $Concession(M_1, M_2)$ and $Support(M_1, M_3)$, and the trees that can be built are:



In order to compose the relations together to obtain trees, the nucleus (or the two nucleus) is said to be *promoted* (Marcu, 1996). The relation as a whole is then represented by its promoted element and can be composed with other relations.

7.4.2 Modelisation

Let $\Sigma_{rhetorics}$ be the signature representing our document structures (i.e. rhetorical trees). Since the set of rhetorical trees is defined by the input messages and relations, which are assumed to be computed at runtime by the wrapping NLG system, all the definitions below are also assumed to be added at runtime.

7.4.2.1 Messages and relations with one nucleus

We have two types of constants: constants representing messages and constants representing relations (which we will call simply messages and relations from now on). Like for tree adjoining grammars (see Section 6.2.2), we can represent trees using the type τ. Messages are always the leaves of the trees, so their type is simply τ. Relations are nodes with two children, so their type is $\tau \rightarrow \tau \rightarrow \tau$. There are two main constraints which define a valid rhetorical tree:

- A node cannot have two parents. We need somehow to enforce this constraint while composing the relations together, and in particular make sure that the leaves (the messages) are not used twice.
- The rhetorical relationship of a particular node of the tree must hold between the two promoted elements of the children of this node.

These constraints are represented as two features of the type τ: *span* and *promotion* (which makes the type $\tau[span, promotion]$). The span feature represents the set of leaves of a tree. Its domain is the power set of the set of messages in the input. The feature promotion represents the promoted element of a tree. If relations can only have one nucleus, then the domain of the feature promotion is simply the set of messages in the input. The set of constants of the signature *rhetorics* is defined from an input as follows.

Let $\mathscr{I} = \{\mathscr{M}, \mathscr{R}\}$ be an input, where $\mathscr{M} = \{M_1, \ldots, M_n\}$ is a set of messages and $\mathscr{R} = \{R_1(A_1^1, A_1^2), \ldots, R_m(A_m^1, A_m^2)\}$ is a set of relations on the messages in $\mathscr{M}$. For each message M_i in $\mathscr{M}$, we add the following constant to the signature $\Sigma_{rhetorics}$[11]

$$M_i : \tau[sp = \{M_i\}, prom = M_i]$$

In other words, since a message is a leaf, it only spans itself and promotes itself. For each relation $R_j(A_j^1, A_j^2)$, we add the following constant to the signature *rhetorics*:

$$R_j : \tau[sp = x, prom = A_j^1] \rightarrow \tau[sp = y, prom = A_j^2] \rightarrow \tau[sp = union(x, y), prom = A_j^1]$$

Where $union(x, y)$ is the union of two disjoint sets (so $sp = union(x, y)$ is equivalent to $sp = x \cup y \wedge x \cap y = \emptyset$). The span of the arguments of the relation are merged only if merging them results in a tree. The promoted message is simply the promoted message of the nucleus of the relation (here A_j^1 by convention). Finally, the set of valid rhetorical

[11] *span* is abbreviated by *sp* and *promotion* by *prom*.

trees in the signature *rhetorics* is the set of second order λ-terms of type:

$$\tau[sp = \{M_1, \ldots, M_n\}, prom = x]$$

Namely, the trees which span all the messages of the input. As an example, applying these rules to the input of Section 7.4.1 yields:

$$\begin{aligned}
M_1 &: \tau[sp = \{M_1\}, prom = M_1] \\
M_2 &: \tau[sp = \{M_2\}, prom = M_2] \\
M_3 &: \tau[sp = \{M_3\}, prom = M_3] \\
Concession &: \tau[sp = x, prom = M_1] \rightarrow \tau[sp = y, prom = M_2] \rightarrow \tau[sp = union(x, y), prom = M_1] \\
Support &: \tau[sp = x, prom = M_1] \rightarrow \tau[sp = y, prom = M_3] \rightarrow \tau[sp = union(x, y), prom = M_1]
\end{aligned}$$

7.4.2.2 Relations with two nucleus and semantic aggregation

For some relations, none of its arguments can be flagged as the most salient part. In this case the relation has two nucleus. For instance, relation of contrast, as in "*this car has an integrated GPS, but it has no sunroof*" has two nucleus. In this case, we need to modify the types in order to allow the promotion of multiple messages. The basic idea is to change the domain of the feature *promotion*, so to represent sets of messages instead of messages (in fact the same domain than the feature *span*). If we allow a relation to promote several messages, we also need relations to take sets of messages as arguments (else we could not combine a relation with two nucleus to any other relation). If we have a relation $R_j(A_j^1, A_j^2)$, *where* A_j^1 *and* A_j^2 *are sets of messages*, then we add the following constant to the signature *rhetorics*:

$$R_j : \tau[sp = x, prom = A_j^1] \rightarrow \tau[sp = y, prom = A_j^2] \rightarrow \tau[sp = union(x, y), prom = A_j^1 \cup A_j^2]$$

The effect of the relation with two nucleus (independently of its meaning) is to aggregate several messages, which can be then used as a whole as an argument of another relation.

More generally, allowing sets of messages as arguments of relations allow to perform semantic aggregation. We can directly create constants corresponding to sets of messages and then use them as arguments of relations. The constant associated with a set $\{M_1, \ldots, M_n\}$ of messages is:

$$M_{1..n} : \tau[sp = \{M_1, \ldots, M_n\}, prom = \{M_1, \ldots, M_n\}]$$

Using these aggregates as arguments of relations is a way of performing semantic aggregation. If the relations of the input do not directly allow aggregates as arguments, we

can generate new relations which do. For instance, if we have these three relations in the input:

$$R_1(M_1, M_2)$$
$$R_1(M_1, M_3)$$
$$R_1(M_1, M_4)$$

Which only differ by their second argument. Then we can create the following relationships which allow for the aggregation of arguments:

$$R_1(\{M_1\}, \{M_2\})$$
$$R_1(\{M_1\}, \{M_3\})$$
$$R_1(\{M_1\}, \{M_4\})$$
$$R_1(\{M_1\}, \{M_2, M_3\})$$
$$R_1(\{M_1\}, \{M_2, M_4\})$$
$$R_1(\{M_1\}, \{M_3, M_4\})$$
$$R_1(\{M_1\}, \{M_2, M_3, M_4\})$$

Depending on which relation is chosen while building a rhetorical tree in the signature $\Sigma_{rhetorics}$, the arguments of the relation are aggregated or not. Since the number of aggregates grows very rapidly with the number of messages in the input, this method is limited to relatively small sets of messages.

7.4.2.3 Discussion

Again, with the given definition, we are at the limit of content selection. By varying the type representing a valid rhetorical tree, we could potentially perform some content selection. For instance the type: $\tau[sp = x, prom = y]$ selects all rhetorical trees, even if they don't span all the input messages. By using a heuristic search on the trees of the signature *rhetorics*, one could combine document structuring and content selection in order, for instance, to maximize the number of messages in the tree while minimizing some cost function (or use a heuristic based on some kind of equilibrium definition).

An interesting result of using ACG for describing both top-down schema-like document structuring and bottom-up data driven document structuring, is that we can now combine the two approaches. We can see the bottom-up approach defined in this section as an extreme case of the recursive structure definitions described in Section 7.3.2.1.

Therefore we could theoretically combine statically defined top-down goal-oriented definitions and dynamic definition describing a bottom-up document structuring problem. The detailed analysis of this approach however is out of the scope of this thesis.

The document structuring method described in this section relies heavily on definitions which are computed dynamically. Indeed, in this section I have assumed that not only the textualisation of the different messages, but also all the constants of the signature $\Sigma_{rhetorics}$ are defined at runtime by the wrapping NLG system. This scenario assumes that we have no information whatsoever about the input messages and relations at compilation time. In practice however this is not true. NLG systems usually accept a limited number of messages, which can be described statically. In this case the definitions relative to the messages are defined statically and restricted at runtime to include only the definitions relative to the input messages. The textualisation of the rhetorical relationships are also known at compilation time, and the job of the wrapping NLG system is simply to create the constants associated with the input relations and link them to their textualisation.

Alternative approaches The general approach to document planning described in this chapter uses ACG in an unconventional way. The input information is not represented as a single λ-term but as constants in a signature, and the definitions of the grammar may be modified depending on the input information (e.g. by removing constants, for instance for selecting a format, or by adding definitions from input objects). This makes the definitions of the grammar used for text generation dependent on the input and therefore it cannot be used for reversing the generation process, as the properties of ACG would normally allow. A more standard approach would require to define a lexicon $\mathscr{L}_{doc\text{-}in}$ with an abstract signature Σ_{doc} representing the document plans and an object signature Σ_{in} representing the input. For instance, in the context of our bottom-up document structuring example, the input $\{\{M_1, M_2, M_3\}, \{Concession(M_1, M_2), Support(M_1, M_3)\}\}$ could be represented as the conjunction:

$$M_1 \wedge M_2 \wedge M_3 \wedge Concession(M_1, M_2) \wedge Support(M_1, M_3)$$

Which could then be represented as a λ-term in the signature Σ_{in}. However λ-terms cannot represent commutative and transitive relations such as the conjunction relation, so we would probably need to represent the input using several λ-terms for the different permutations.

In the context of document planning with schemas, where the input can be represented as a set of variables $\{V_1, \ldots, V_n\}$, we could again represent the input in the form of a

conjunction:

$$V_1 \wedge \cdots \wedge V_n$$

Which could then be represented as a λ-term in the signature Σ_{in}. In this case, the schemas, represented as constants in the signature Σ_{doc} could be interpreted in Σ_{in} in the same way nodes of TAG trees are interpreted into a string language (i.e. as concatenation operator). The input would simply need to be ordered so that it is accepted by the grammar. In the case where no schema is defined recursively, which concerns a non negligible part of industrial applications, there is only one λ-term accepted by the grammar and we perform a lexicon inversion (a costly operation, see Chapter 8), for the sake of this single λ-term.

Given these problems and the fact that we are in the context of an existing industrial NLG framework whose sole purpose is to generate texts, the approach based on the definition of a finite set in the signature Σ_{doc} has been chosen, this alternative method being both efficient in simple cases and powerful enough to represent more complex scenarios. However this choice depends on contextual factors, and alternative approaches should be tested and compared more thoroughly in order to find a more general method. This question has been left for future researches.

7.5 Conclusion

In order to use ACG as a kernel technology of a NLG framework (see Chapter 5), we need to be able to perform both document planning and microplanning. The linguistic resources used for microplanning are presented in Chapter 8. In this chapter I explored different techniques for document planning and how they could be represented using ACG. In particular, we can show that the technique of schemas can be simulated using ACG, and that RST based data driven document structuring can also be represented using ACG. With these two techniques, we cover most of the document planning used by NLG applications in practice. In summary, by showing that besides text structures, we can also represent realistic document structures, this chapter confirms that we can use ACG as a kernel technology for a NLG framework.

Chapter 8

Implementation

In Chapter 7, I have shown how ACG could theoretically be used as the core technology of a document planner, and more generally as the core technology of a NLG system, for building applications in the context of a NLG framework. In this Chapter, I present an actual implementation of an ACG based NLG framework. This implementation has been developed as an extension of the existing NLG framework of the Yseop company. The implementation does not (yet[1]) cover all the propositions of Chapter 7. In particular it does not handle types with features. However it covers all the basic usages of ACG plus the manipulation of sets of constants as presented in Section 7.3.1.1. The main interest of the work presented in this chapter is to show that ACG can be successfully implemented in an industrial context and integrated into an existing NLG technology, with the possibility of delegating some parts of the NLG process not yet handled by ACG to the existing technology. For instance, in the current implementation, ACG is used to handle document structuring, lexicalisation and aggregation, but delegates content selection, referring expressions generation and realisation to the existing technology. Having a smooth integration of ACG in the existing technology is an important aspect for the conciliation of the imperatives of simplicity, flexibility and efficiency inherent to practical applications on one side, and the possibility of exploration and research necessary in order to find the best possible usage of ACG for NLG on the other. Section 8.1 first presents the existing Yseop technology and the general architecture of the ACG implementation within this technology. The Section 8.2 dives into the internal mechanisms for generating and transforming structures using ACG. Finally Section 8.3 presents the linguistic resources which have been developed in parallel of the ACG kernel, mainly for microplanning purposes. The usage of theses linguistic resources in the context of realistic NLG applications is presented in Chapter 9.

[1] Yseop has shown an interest in developing the prototype implementation further, so a future version of this work will probably include the aspects which have been left aside here by lack of time.

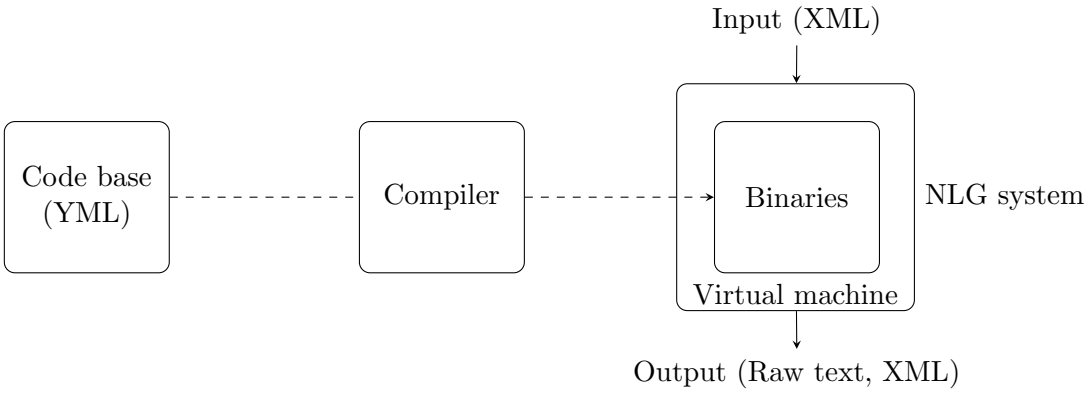


FIGURE 8.1: The general workflow of NLG systems creation using Yseop technology. The code base is written using a proprietary language similar to JAVA: Yseop Modelling Language (YML). The code base is compiled by a program written in C++ into a binary format. At runtime, the binaries are loaded by a virtual machine (also written in C++), which executes the code base on a XML input and outputs either raw text on the standard output or XML through sockets.

8.1 General architecture

The Yseop technology follows the general architecture for NLG frameworks of Figure 7.1, with a code base describing the NLG system being compiled into a program which accepts a given range of input and outputs natural language. The code base is written in a proprietary language called Yseop Modelling Language (YML), which resembles JAVA. The compilation is handled by a compiler program written in C++. The output of the compiler is a file in a custom binary format which is loaded at runtime by a virtual machine, which is also written in C++ (see Figure 8.1).

The ACG extension of the Yseop technology is provided as a set of *predefined classes*. Predefined classes are classes which are available in all applications written in YML, similar to classes from the standard libraries in JAVA or C++. However, unlike them, the predefined classes of YML are not written in YML, but have a C++ implementation in the virtual machine. This configuration allows to provide high level concepts in the YML language with an efficient C++ implementation (since it is interpreted, the code written in YML is usually a bit slower than the same code written directly in C++, hence the custom C++ implementation for critical components). The ACG extension of YML provides predefined classes for the basic components of ACG: lambda terms, signatures and lexicons. Additionally, a class for *composed lexicons* is added, which encapsulates the composition of several lexicons.

8.1.1 Lambda terms

λ-terms are represented using an abstract class `LambdaTerm`, which is specialized into three subclasses: `Constants`, `Variable` and `Abstraction`. The class `LambdaTerm` holds the type of a λ-term, and the logic of the β-reduction process[2]:

```
class LambdaTerm
  Collection type
  static LambdaTerm betaReduce(Abstraction lhs, LambdaTerm rhs [, Symbol arg])
```

A type is represented by an ordered collection, were the last element of the collection is the return type and the other elements are the arguments (which can also be collections). For instance, the collection `[alpha, [alpha, beta], beta]` represents a type $\alpha \to (\alpha \to \beta) \to \beta$. An atomic type is represented by a collection with the symbol of this atomic type as its only element (the symbols used for atomic types can be any object). The implementation also includes the notion of *named arguments*. Any argument of a type can be associated with a *name* symbol, which can be used to refer to this argument. This association is done using a map. For instance, the collection `[{x : alpha}, {y : [alpha, beta]}, beta]` still represents a type $\alpha \to (\alpha \to \beta) \to \beta$, but the first and the second arguments can be referred as `x` and `y` respectively. This feature is used for declaring variables in λ-abstractions and for specific β-reduction operations (see below).

The function `betaReduce` takes two λ-terms `lhs` (left hand side) and `rhs` (right hand side) and returns the result of applying `lhs` on `rhs` (the operation has no side effect). The function `betaReduce` also includes a slight modification of the usual β-reduction process by allowing to select the argument of the left hand side which is used in the operation. For instance, if we have a constant c and a λ-term $\lambda xy.xy$, then the function `betaReduce` allows to apply $\lambda xy.xy$ on c such that the result is the λ-term $\lambda x.xc$ (the second argument y is consumed instead of the first one x as it should be the case in normal β-reduction). This behaviour is available through an optional argument of the function `betaReduce`, noted here as `[, Symbol arg]` (the brackets represent the optionality). `arg` is the named argument of `lhs` which should be used for the β-reduction[3].

The class `LambdaTerm` is never instantiated (it is the equivalent of an abstract class in JAVA, or a virtual one in C++). Instead, it is specialized into the three basic building blocks of λ-term: constants, variables and λ-abstractions. Constants and variables do

[2]Here and in the following, I use a simplified JAVA-like syntax, with line breaks replacing semicolons and fewer brackets. Ordered lists are noted `[]` and maps `{}`. Other syntactic specifics are explained where they occur.

[3]This is only syntactic sugar, the same behaviour can be achieved using a combinator.

not add any field or functions to the base class, and are simply used to instantiate the basic building blocks of a λ-term:

```
class Constant extends LambdaTerm

class Variable extends LambdaTerm
```

λ-abstractions however, have a `body` field, which represents the body of the λ-abstraction:

```
class Abstraction extends LambdaTerm
  LambdaTerm body
```

As indicated above, the variables of a λ-abstraction are declared directly in its type (see the example below). The syntax of the YML language does not enforce any particular constraint on the λ-terms that can be built. However, the compiler and the virtual machine (for dynamically created objects) include checks which enforce almost-linear λ-terms (non deleting and non duplicating for variables with non-atomic type). Also, any λ-term which is built explicitly must have a type. The following shows an example of creation and manipulation of λ-terms:

```
Constant a([alpha]) // a : α
Constant b([alpha, beta]) // λx.bx : α → β
Abstraction term([{[alpha, beta]: x}, {alpha: y}, alpha], x(y))
LambdaTerm result = LambdaTerm::betaReduce(term, a, y) // result = λx.xa
result = result(b) // result = ba
```

There are a few things to explain about this example. First, objects are created using a constructor (in the previous definitions, the constructor has been left implicit). All classes which inherit from `LambdaTerm` take a type as their first constructor argument (for instance `a([alpha])`). Additionally, objects of the class `Abstraction` require a second argument, their body. Second, on the third line, the variables `x` and `y` are implicit declarations, which means that `x` and `y` are objects of class `Variable`. Finally, a special syntax allow to use objects of class `LambdaTerm` to be used as functions. Using such an object as function is equivalent to using the function `betaReduce` on the object and its argument. The implicit declarations and special syntax are possible thanks to modifications of the compiler.

In details, the two first lines of the examples create two constant objects a and b of type α and $\alpha \to \beta$ respectively. Note that the internal representation of b is $\lambda x.bx$. More generally, all λ-terms are represented internally in their η-long form. The internal representation of λ-terms uses term graphs (Plump, 1999) as in (Kanazawa, 2011)[4]. The

[4]The details of the internal representation of λ-terms are very dependent on the specific architecture of the compiler and virtual machine, and are not developed in this thesis. However the implementation follows closely the representation described in the given papers.

third line creates a λ-abstraction, corresponding to $\lambda xy.xy : (\alpha \to \beta) \to \alpha \to \beta$. The fourth line is an example of β-reduction using a named argument, and the last line a classical β-reduction, using the object `result` as a function.

In summary, λ-terms are implemented in a relatively straightforward way in YML, using object oriented programming concepts. However a few modifications have been included in order to ease development as much as possible. In particular, the fact that the code of the compiler is available allows to add some syntactic sugar, for instance for implicit variable declarations or using objects as functions. The possibility of using named arguments for β-reduction also adds some flexibility to the way λ-terms can be built. In particular, it is very useful for manipulating resources such as a grammar, as shown in Section 8.3.

8.1.2 Signatures and lexicons

The predefined classes for signatures and lexicons follow closely the formal definitions of these concepts. The class `Signature` contains a set of atomic types and a set of constants:

```
class Signature
  Collection atomicTypes
  Collection constants
  Collection generate(Symbol type)
  bool checkType(Collection type)
  bool checkTerm(LambdaTerm term)
```

It also provides a function `generate`, which takes a type built on the set `atomicTypes` and returns the set of terms of this type in the signature (an error is thrown if the set is infinite), and two function `checkType` and `checkTerm`, which check respectively if a type or a λ-term is in the signature or not. In practice, the goal of the function `generate` is to generate document plans, such as described in Chapter 7. The other two functions are mainly used internally and for debugging purposes.

The class `Lexicon`, contains an abstract signature, an object signature and a mapping of constants from the abstract signature to λ-terms of the object signature. Note here that it is allowed to map a constant from the abstract signature to several λ-terms of the object signature:

```
class Lexicon
  Signature abstractSignature
  Signature objectSignature
  Map mapping
```

```
  Collection realize(LambdaTerm term)
  Collection parse(LambdaTerm term)
  Collection translate(LambdaTerm term, Signature origin, Signature target)
  Collection generate(Symbol type, Signature origin, Signature target)
```

There are four functions which can be used on a lexicon object. The function `realize`[5] takes a λ-term from the abstract signature and returns its images in the object signature. The basic behaviour of this function is to substitute each constant in the input λ-term by its image, using the `mapping` object (and then β-reduce the result). However it is a bit different from the usual mapping function, as it needs to handle constants with several images and therefore returns a collection instead of a single λ-term (the collection may still contain a single image). If several constants in the input λ-term have multiple images, then the result contains one λ-term for each possible combination of constant substitution.The size of the resulting collection is the size of the Cartesian product of all the image sets of the constants of the input λ-term.

The function `parse` takes a λ-term from the object signature, and returns its inverse images by the lexicon. This inverse mapping procedure is the detailed in Section 8.2.

The functions `translate` and `generate` are only introduced in order to be able to manipulate lexicons and *composed lexicons*, in a transparent way and are described in Section 8.1.3.

8.1.3 Composed lexicons

The predefined class `ComposedLexicon` allows to manipulate several lexicons composed together in a transparent way. The class only contains a set of lexicons (of class `Lexicon`) and two functions:

```
class ComposedLexicon
  Collection lexicons
  Collection translate(LambdaTerm term, Signature origin, Signature target)
  Collection generate(Symbol type, Signature origin, Signature target)
```

The function `translate` takes as argument a λ-term from the signature `origin` and translates it into one or several λ-terms of the signature `target`. The preconditions of the function are:

- The `origin` and `target` signatures must be the object signature or the abstract signature of at least one lexicon in the set `lexicons`.

[5]The names "realize" and "parse" come from the ACG implementation of the Sémagramme team (LORIA).

- The argument `term` is a λ-term built on the signature `origin`.
- There exists a *path* between the origin and target signatures.

A path here is defined as a sequence of signatures, where two consecutive signatures are connected by a lexicon of the set `lexicons`. So a path must have at least two members, and each arc in the path represents either a mapping operation or an inverse mapping operation[6]. For instance, Let `lex1` be a lexicon with abstract signature `sig1` and object signature `sig2` and `lex2` be a lexicon with abstract signature `sig1` and object signature `sig2` (so that `lex1` and `lex2` share their abstract signature `sig1`). Now let `lexComp` be the composed lexicon containing the set `{ lex1, lex2 }`. Then the possible paths in `lexComp` are:

```
{
    [sig1, sig1], [sig2, sig2], [sig3, sig3],
    [sig1, sig2], [sig2, sig1], [sig2, sig3], [sig3, sig2],
    [sig1, sig2, sig3], [sig3, sig2, sig1]
}
```

Which correspond to the following usages of the function `translate` of `lexComp`:

`lexComp.translate(t, sig1, sig1)` // $identity(t)$
`lexComp.translate(t, sig2, sig2)` // $identity(t)$
`lexComp.translate(t, sig3, sig3)` // $identity(t)$
`lexComp.translate(t, sig2, sig1)` // $\mathscr{L}_1(t)$
`lexComp.translate(t, sig1, sig2)` // $\mathscr{L}_1^{-1}(t)$
`lexComp.translate(t, sig2, sig3)` // $\mathscr{L}_2(t)$
`lexComp.translate(t, sig3, sig2)` // $\mathscr{L}_2^{-1}(t)$
`lexComp.translate(t, sig1, sig3)` // $\mathscr{L}_2(\mathscr{L}_1^{-1}(t))$
`lexComp.translate(t, sig3, sig1)` // $\mathscr{L}_1(\mathscr{L}_2^{-1}(t))$

The path is computed automatically from the arguments of the function each time the function is called, and an error is thrown if more than one path is possible[7]. To summarize, the function translate provides a convenient way to manipulate sets of lexicons, but delegates most of the logic to the mapping and inverse mapping functions (i.e. the function `realize` and `parse` of the class `Lexicon`). The job of the function `translate` is to compute the path between the origin and target signatures and to transfer the results of one operation to the next operation in the path.

[6] An arc of a signature to itself is considered as the identity operation. Except from this kind of basic loop, no loops are allowed in a path.

[7] This is a limitation of the implementation. The user should be allowed to specify a path when an ambiguity exists.

The function `generate` is a shorthand which allows to compose the function `Signature::generate` (the function `generate` of the class `Signature`) with the function `translate`. The preconditions of the function `generate` are the same as the precondition of the function `translate`, except that the first argument is now a type which must belong to the `origin` signature. A call to `lex.generate(type, origin, target)`, where `lex` is a (composed) lexicon is strictly equivalent to `lex.translate(origin.generate(type), origin, target)` (assuming that `translate` is overloaded in order to allow collections of λ-terms for its first argument). The purpose of this function is to generate text from the result of the document planning phase.

8.1.4 Summary

Together, the predefined classes `LambdaTerm`, `Constant`, `Variable`, `Abstraction`, `Signature`, `Lexicon` and `ComposedLexicon` form the ACG kernel implemented in the Yseop technology. These classes allow to build grammars and to perform document planning as described in Chapter 7, and microplanning (see Section 8.3). The objects instantiating these classes can be defined in the code base, in which case they form a static definition of the output texts, but they can also be created dynamically either by using the `new` keyword, like in JAVA or C++, or by definitions given in the input of the NLG system (the Yseop technology allows to complete the code base at runtime using the XML input). This gives us all the flexibility needed in order to handle the different kinds of inputs and contexts introduced in Chapter 7; i.e. it is possible to define structures statically, and to either complete, restrict or modify these static definitions at runtime, depending on the input. This flexibility comes mainly from the fact that the kernel is embedded in a general purpose programming language (YML).

While the predefined classes described so far are available in the YML language, the functions they provide are all implemented as C++ functions in the virtual machine. As explained earlier, the critical components of the Yseop technology, which are available through predefined classes, benefit from a low level optimized implementation, in order to balance the fact that YML is an interpreted language. There are ten functions implemented in C++ in the virtual machine:

```
LambdaTerm::betaReduce
Signature::generate
Signature::checkType
Signature::checkTerm
Lexicon::realize
Lexicon::parse
Lexicon::translate
Lexicon::generate
```

```
ComposedLexicon::translate
ComposedLexicon::generate
```

Section 8.2 explains in more details the functions `Signature::generate`, `Lexicon::parse` and `ComposedLexicon::translate`. The other functions do not pose any particular problem to implement. While being a bit involved, the function `LambdaTerm::betaReduce` is heavily dependent on the internal representation of λ-terms, which have been left aside in this thesis[8]. The functions `Signature::checkType` and `Signature::checkTerm` are trivial recursive membership checks. The function `Lexicon::realize` only involves simple substitutions and β-reductions. The functions `Lexicon::translate` and `Lexicon::generate` are introduced for compatibility reasons and are basically overloads of the functions of the class `ComposedLexicon`. Finally, `ComposedLexicon::generate` is a utility function which can be reduced to a composition of the functions `Signature::generate` and `ComposedLexicon::translate`.

8.2 Generation and transformation of structures

8.2.1 Example grammar

In order to illustrate the following explanations, I will use a toy grammar example. Here and in the rest of the chapter, I use a more compact mathematical notation for the λ-terms, signatures and lexicons, but all definitions can also be defined using the predefined classes of Section 8.1.

Let's say we have four signatures: Σ_{logics}, $\Sigma_{derivations}$, Σ_{trees} and $\Sigma_{strings}$, with the following constants:

$$\Sigma_{logics} = \{c_{love} : e \to e \to t,$$
$$c_{John}, c_{Mary} : e\}$$

$$\Sigma_{derivations} = \{d_{love} : NP \to NP \to S,$$
$$d_{John}, d_{Mary} : NP\}$$

[8] See Section 8.1.1. The β-reduction process has simple and well known specification. The implementation uses term graphs, and the basic idea of the implementation of the β-reduction is to merge the nodes of the two input graphs to produce a new graph which represents the result of the β-reduction of the input λ-terms.

$$\Sigma_{trees} = \{S_2, VP_2 : tree \rightarrow tree \rightarrow tree,$$
$$NP_1 : tree \rightarrow tree,$$
$$loves, John, Mary : tree\}$$

$$\Sigma_{strings} = \{\text{“}loves\text{”}, \text{“}John\text{”}, \text{“}Mary\text{”} : o \rightarrow o\}$$

And that we have three lexicons: a lexicon $\mathscr{L}_{der\text{-}log}$ with abstract signature $\Sigma_{derivations}$ and object signature Σ_{logics}, $\mathscr{L}_{der\text{-}tag}$ with abstract signature $\Sigma_{derivations}$ and object signature Σ_{trees}, and $\mathscr{L}_{tag\text{-}str}$ with abstract signature Σ_{trees} and object signature $\Sigma_{strings}$, these three lexicons being defined as follows:

$$\mathscr{L}_{der\text{-}log} = \{S : t,$$
$$NP : e,$$
$$d_{love} : c_{love},$$
$$d_{John} : c_{John},$$
$$d_{Mary} : c_{Mary}\}$$

$$\mathscr{L}_{der\text{-}tag} = \{S, NP : tree,$$
$$d_{love} : \lambda xy.S_2\ x\ (VP_2\ loves\ y),$$
$$d_{John} : NP_1\ John,$$
$$d_{Mary} : NP_1\ Mary\}$$

$$\mathscr{L}_{tag\text{-}str} = \{tree : o \rightarrow o,$$
$$S_2, VP_2 : \lambda xyz.x\ (y\ z),$$
$$NP_1 : \lambda x.x,$$
$$loves : \text{“}loves\text{”},$$
$$John : \text{“}John\text{”},$$
$$Mary : \text{“}Mary\text{”}\}$$

Figure 8.2 shows $\mathscr{L}_{comp}$, the composition of the three lexicons $\mathscr{L}_{der\text{-}log}$, $\mathscr{L}_{der\text{-}tag}$ and $\mathscr{L}_{tag\text{-}str}$. Since we are in the context of a NLG framework, we are only interested here in transforming λ-terms built on the signature Σ_{logics} to λ-terms built on the signature $\Sigma_{strings}$. This translation corresponds to the path $[\Sigma_{logics}, \Sigma_{derivations}, \Sigma_{trees}, \Sigma_{strings}]$ of $\mathscr{L}_{comp}$, which itself corresponds to the three consecutive transformations $\mathscr{L}^{-1}_{der\text{-}log}$,

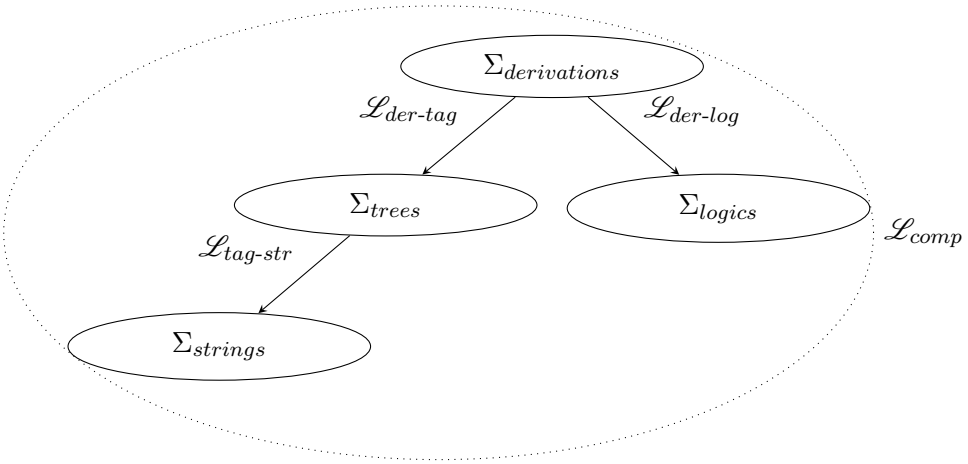


FIGURE 8.2: Example composition of three lexicons: $\mathscr{L}_{der\text{-}log}$, $\mathscr{L}_{der\text{-}tag}$ and $\mathscr{L}_{tag\text{-}str}$. The lexicon $\mathscr{L}_{comp}$ represents the composition of these three lexicons in an instance of the class `ComposedLexicon`. The path of the composed lexicon which is used to generate text from a logical sentence l is $[\Sigma_{logics}, \Sigma_{derivations}, \Sigma_{trees}, \Sigma_{strings}]$ and corresponds to the operation $\mathscr{L}_{tag\text{-}str}(\mathscr{L}_{der\text{-}tag}(\mathscr{L}^{-1}_{der\text{-}log}(l)))$.

$\mathscr{L}_{der\text{-}tag}$ and $\mathscr{L}_{tag\text{-}str}$. For instance, we can translate the λ-term $c_{love}\ c_{John}\ c_{Mary}$ to a λ-term built on $\Sigma_{strings}$ using $\mathscr{L}_{comp}$ as follows:

$$\begin{aligned}\mathscr{L}_{comp}.translate(c_{love}\ c_{John}\ c_{Mary}, \Sigma_{logics}, \Sigma_{strings}) &= \mathscr{L}_{tag\text{-}str}(\mathscr{L}_{der\text{-}tag}(\mathscr{L}^{-1}_{der\text{-}log}(c_{love}\ c_{John}\ c_{Mary}))) \\ &= \lambda z.\text{“}John\text{”}(\text{“}loves\text{”}(\text{“}Mary\text{”}\ z))\end{aligned}$$

Sections 8.2.2 and 8.2.3 explain in details the operation of inversion of a lexicon, corresponding to the function `parse` of the lexicon $\mathscr{L}_{der\text{-}log}$ of our example (noted $\mathscr{L}^{-1}_{der\text{-}log}$ above). Section 8.2.4 gives more details on the inner workings of the function `ComposedLexicon::translate`, here corresponding to the composition $\mathscr{L}_{tag\text{-}str}(\mathscr{L}_{der\text{-}tag}(\mathscr{L}^{-1}_{der\text{-}log}(\dots)))$. Finally, Section 8.2.5 describes the function `Signature::generate`.

8.2.2 Datalog prover

The implementation of the inversion of a lexicon (function `Lexicon::parse`) follows the algorithm given in (Kanazawa, 2007), which reduces the inversion of a lexicon to Datalog querying[9]. This Section gives a quick introduction to Datalog and describes the Datalog prover implemented.

[9]This method is limited to abstract categorial grammars of order two (the constants of the root signature of the grammar can have a type of order at most two), with almost-linear λ-terms (the image of a constant by a lexicon cannot contain duplicated variables, unless the type of the variable is atomic). Therefore the implementation is also limited to almost-linear second order ACG. The method can be generalized to any ACG using linear logic programming (De Groote, 2015).

8.2.2.1 Quick introduction to Datalog

Datalog (see for instance Abiteboul et al., 1995), is a logic programming language, or *deductive database*, which is a subset of Prolog (Clocksin and Mellish, 2003). In logic programming, the program is defined as a database of facts and implication rules, which can be *queried*, in order to retrieve existing facts or deduce new ones.

A *Datalog program* is divided between an *extensional database* and an *intentional database*. An intentional database is a set of rules of the form

$$p_0(\vec{v}_0) \text{ :- } p_1(\vec{v}_1), \ldots, p_n(\vec{v}_n)$$

where $p_0, \ldots, p_n$ are predicates of fixed arity and $\vec{v}_0, \ldots, \vec{v}_n$ are tuples of variables. A predicate together with its argument is called an *atom*, the atom on the left hand side of a rule (here $p_0(\vec{v}_0)$) is called the *head* of the rule and the atoms on the right hand side of a rule (here $p_1(\vec{v}_1), \ldots, p_n(\vec{v}_n)$) the *body* of the rule. The individual atoms in the body are called *subgoals*. A rule may have no body, in which case it is called a *bodyless rule*.

An extensional database is a set of *ground facts* or *extensional facts* of the form

$$p(\vec{c})$$

where p is is a predicate of fixed arity and $\vec{c}$ a tuple of constants[10]. Predicates which appear in the head of a rule are called *intensional predicates*. The other predicates are called *extensional predicates*.

A *query* is a conjunction of facts (grounded or not). A query q is said to be derivable from a program P if $P \vdash q$ (i.e. if q can be deduced from P). A *derivation* of P given q is a proof of q in P (i.e. the sequence of rules and facts which proves the query). A Datalog solver takes a query and a program, checks if the query is derivable from the program, and returns true if it is the case and false otherwise. If the query is indeed derivable from the program, it also usually gives the range of values that the variables of the query can take in order to be derivable from the program (i.e. the different values that the variables can take in every possible derivations of P given q). A typical toy example is the ancestor program, whose intentional database contains the rules:

$$ancestor(X, Y) \text{ :- } parent(X, Y).$$
$$ancestor(X, Y) \text{ :- } parent(X, Y), ancestor(Z, Y).$$

[10] In the following, unless stated otherwise, *facts* always refer to ground facts.

These rules recursively define the notion of ancestor as one's parent, or ancestor of one's parent ($parent(X, Y)$ represents the fact that Y is a parent of X, and $ancestor(X, Y)$ the fact that Y is an ancestor of X). An associated extensional database might be:

$$parent(pierre, paul).$$
$$parent(paul, jacques).$$

Given this program, we might for instance have the following queries:

$$\text{?- } ancestor(pierre, jacques).$$
$$\text{?- } ancestor(paul, pierre).$$
$$\text{?- } ancestor(pierre, X).$$

The first query returns *true* since *jacques* is the grand-father of *pierre*, but the second one returns $false$. The last query introduces a variable X which holds for any possible ancestor of *pierre*. Since there are values of X for which a derivation of the program exists, the query returns *true*. Additionally, we can get the the set of values of X for which the program has a derivation: $X = \{paul, jacques\}$ (i.e. X can be the father or grand-father of *pierre*).

8.2.2.2 Implementation

Datalog solvers only return *true* or $false$ given a query (plus the bindings of the variables of the query). However, in our case we also need to get the actual derivations which prove the query (see Section 8.2.3). So rather than a Datalog solver, we actually need a *Datalog prover*. Since there are no C++ Datalog prover which can easily be integrated in an existing proprietary software, I have implemented a Datalog prover from scratch. It uses a depth-first search algorithm (backward chaining) and an independent solver, which holds the equality constraints generated by the unification process (see below).

Algorithm 2 is a slightly simplified version of the implemented proving procedure[11]. The main function takes a list of goals (initially, the list of atoms in the query), a database and a solver. The algorithm is recursive: it first checks the stopping condition which is that no more goals need to be proven, in which case it returns the list of rules and facts which have been expanded until this point. If there are still goals which need to be proven, it tries to prove the next goal (i.e. atom) in the goal stack. A goal can be extensional or intentional, depending on its predicate. For extensional goals, the algorithm tries

[11] This algorithm stops as soon as it finds a valid derivation. The implemented algorithm continues until all possible valid derivations are found or no more derivation can be found.

Algorithm 2: The Datalog prover main algorithm.

```
function prove(goals, solver, database)
    if goals is empty then
        return current derivation
    end
    currentGoal ← goals.top
    goals.pop()
    if currentGoal is extensional then
        for fact ← nextFact(currentGoal, database) do
            if unify(currentGoal, fact, solver) then
                branch(goals, solver, database)
            end
        end
        backtrack()
    else
        for rule ← nextRule(currentGoal, database) do
            if unify(currentGoal, rule, solver) then
                goals.append(rule.body)
                branch(goals, solver, database)
            end
        end
        backtrack()
    end
    return false
end
```

to unify (see below) the goal with all grounded facts with the same predicate in the database. For each successful unification, the algorithm recursively branches on the rest of the goals. When all possible facts have been tested (i.e. no proving fact could be found), the algorithm backtracks to a previous state in order to continue the search. The procedure for intentional goals is similar. The only difference is that the algorithm now tires to unify the head of the rules with the same predicates than the goals, and adds the body of the rule to the pending goals if the unification is successful. Overall, this algorithm is a classical depth first search algorithm. The part specific to Datalog is the unification algorithm, which feeds a constraint satisfaction solver with new constraints while the search goes on.

The unification procedure applies on two atoms: the current goal, and a ground fact, or the head of a rule, depending on whether the current goal is extensional or not. It returns true if the two atoms can be unified and false otherwise. Two atoms can be unified if they have the same predicate, and all their arguments (variables and/or constants) can be unified two by two (the first argument of the first atom with the first argument of the second atom, the second argument of the first atom with the second argument of

the second atom, etc.). The rules for the unification of two arguments, depending on their type are the following:

- Two constant unify if they are equal.
- A constant c and a variable x unify if the constraint $c = x$ (added to the solver) is satisfied.
- A variable x and a variable y unify if the constraint $x = y$ (added to the solver) is satisfied.

So the unification mechanism produces constraints, which are added to the solver, which in turn checks that all constraints are compatible (for instance, if we have the constraints $x = 1$ and $y = 2$ in the solver and we try to add the constraint $x = y$ during an unification, then all constraints cannot be satisfied at the same time and the unification fails). The solver keeps the constraints in memory during the search[12], so the effect of Algorithm 2 is actually to build a constraint satisfaction problem (only containing equality constraints), while unifying atoms and branching in the search tree.

As it is usually presented, the unification mechanism does not include an external solver. Using an external solver is only used in the case of constraint logic programming, where we have the possibility of adding different constraints and build more complex constraint satisfaction problems. While the current implementation does not support other constraints than equality constraints, I expect that using ACG with types with features would require such capacity. Therefore the implementation has been built using an architecture fit for constraint logic programming in this perspective.

Since the implementation is custom made and the time for building it was relatively short, many parts are implemented using naive approaches. In particular, the equality constraint solver simply holds a collection of constraints and browses it each time a new constraint is added. A (much) better way would involve an indexed graph structure. Overall, the Datalog prover is usable in a prototype application, but would require some work in order to be adapted to a production environment. As we will see in Chapter 9, this situation makes it a bit difficult to evaluate with precision the performances that we should expect of the ACG kernel in realistic situations. However the tests carried out are encouraging, even with the shortcomings of the current implementation.

[12]The constraints are local to a derivation, so when the algorithm backtracks, it also removes the constraints added during the last step.

8.2.3 Inversion of a lexicon

The inverse image of a λ-term by a lexicon can be obtained using a reduction of the problem to Datalog querying (Kanazawa, 2007). The principle is to build an intentional database from the lexicon and an extensional database from the input λ-term. The resulting program can then be queried in such a way that the derivations proving the query correspond exactly to the inverse images of the input λ-term (each valid derivation is isomorphic to a λ-term over the abstract signature of the lexicon if and only if the image of this λ-term by the lexicon is the λ-term that we want to inverse).

8.2.3.1 Reduction of a lexicon to an intentional database

A lexicon $\mathscr{L}$ is reduced to an intentional database as follows. For each constant c in the abstract signature, we build a rule π_c from the type of c, the image of c by $\mathscr{L}$, $\mathscr{L}(c)$, and the *principal type* of $\mathscr{L}(c)$[13]. The principal type of a term is the most general type that can be associated with a λ-term. When we compute the principal type of $\mathscr{L}(c)$, we consider the constants and variables as free variables[14]. This operation gives us typing judgements on the λ-term and its constants and variables. For instance, using the example of Section 8.2.1, the typing judgements obtained from $\lambda xy.c_{love}\ x\ y$ are:

$$\begin{aligned} c_{love} &: \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \\ x &: \alpha_1 \\ y &: \alpha_2 \\ c_{love}\ x\ y &: \alpha_3 \end{aligned}$$

Using these typing judgements, we can build a Datalog rule as follows:

- For each atomic type in the type of c, and for each constant in $\mathscr{L}(c)$, build an atom, using a predicate named after this atomic type or this constant.
- Fill each atom with variables using the following procedure:
  - Take the typing judgement of the constant or variable associated with the atom (for the return type of the interpretation type, use the typing judgement of the whole term).

[13] In the case where we have multiple images for a constant, a rule is created for each image of the constant. Another way to say it is that we duplicate the abstract constant in order to have pairs of abstract constant/object λ-term, and use these pairs in order to build the rules of our program.

[14] The λ-term must be in η-long form. In our case, the typing judgement can be obtain directly from the term graph representation of a λ-term, see (Kanazawa, 2011).

  - For each atomic type in this typing judgement, add a variable to the predicate. This operation is done from left to right, and the variables should be coherent between the different predicates (for instance, the atomic type α_1 should always be replaced by the same variable X_1).

- The head of the rule is the atom associated with the return type of the type of c. The other atoms go in the body of the rule.

For example, let's build the intentional database associated with the example lexicon $\mathscr{L}_{\text{der-log}}$. This lexicon maps the abstract constant d_{love} of type $NP \to NP \to S$ to the λ-term $\lambda xy.c_{love}\ x\ y$ (i.e. the η-long form of the constant c_{love}). The rule associated with this constant contains four atoms: the atoms $NP()$, $NP()$ and $S()$ built from the atomic types of the type of d_{love}, and the atom $c_{love}()$ from the constants of $\mathscr{L}_{\text{der-log}}(d_{love})$. These atoms are filled with variables, using the typing judgements obtained from $\lambda xy.c_{love}\ x\ y$ described above: the first $NP()$ atom (corresponding to the variable x in $\lambda xy.c_{love}\ x\ y$) is filled with the variable associated with the type α_1 (X_1), the second $NP()$ atom (corresponding to the variable y) is filled with the variable associated with the type α_2 (X_2) and the atom $S()$ associated with the return type is filled with the variable associated with the type α_3 (X_3). Finally, the atom $c_{love}()$ is filed using the type $\alpha_1 \to \alpha_2 \to \alpha_3$. Putting all the atoms together gives us the rule:

$$S(X_3) \text{ :- } c_{love}(X_1, X_2, X_3), NP(X_1), NP(X_2).$$

Note that the atoms associated with the constants (here $c_{love}(X_1, X_2, X_3)$) are put before the other atoms in the body of the rule. This is important since the Datalog prover uses a depth first search algorithm and needs to avoid recursion as much as possible (predicates associated with constants are always extensional predicates, which cannot have recursive definitions). Applying the same logic to the other mappings of our example lexicon $\mathscr{L}_{\text{der-log}}$ gives us the intentional database:

$$\begin{aligned} S(X_3) &\text{ :- } c_{love}(X_1, X_2, X_3), NP(X_1), NP(X_2). \\ NP(X_1) &\text{ :- } c_{John}(X_1). \\ NP(X_1) &\text{ :- } c_{Mary}(X_1). \end{aligned}$$

To summarize, the reduction process could be interpreted as follows. In order to get the inverse image of a λ-term by a lexicon, we need to take into account both the structure of the λ-terms built over the object signature, and the types of the constants of the abstract signature, which constrain the possible combinations of the λ-terms built over the object

signature. The reduction to Datalog combines these two constraints into Datalog rules. The principal type of a λ-term gives us the information about its structure, and it is combined with the type of an abstract constant which gives us the information about the allowed combinations.

8.2.3.2 Reduction of an input λ-term to an extensional database

The reduction of the input λ-term to an extensional database follows approximately the same process as the reduction of an abstract constant and its image to a Datalog rule. First we need to associate a type with the input λ-term (corresponding to the type of the abstract constant in the reduction described previously). By default, we take the *distinguished type* of $\mathscr{L}$[15]. Using the distinguished type, we can apply the same procedure than in Section 8.2.3.1 for building a set of atoms, with the following differences:

- the atoms are ground facts, so instead of variables, the atoms are filled with constants (e.g. integers).
- Instead of building a rule, we take all the atoms built from constants in order to make an extensional database, which is associated with the previously built intentional database.
- Since the distinguished type is always an atomic type, there is only one atom left after building the database. This atom is used as a query.

Let S be the distinguished type of our example lexicon $\mathscr{L}_{\text{der-log}}$, which represents sentences. Using this type to reduce the λ-term $c_{love}\ c_{John}\ c_{Mary}$ to a set of atoms, we obtain the following extensional database:

$$c_{love}(1, 2, 3).$$
$$c_{John}(1).$$
$$c_{Mary}(2).$$

And the query:

$$?\text{-}\ S(3).$$

There is a slight complication when the input λ-term contains more than once the same constant. Since the reduction to Datalog rules of Section 8.2.3.1 sees duplicate variables

[15] This type can also be given as a parameter of the function `Lexicon::parse`. This allows for instance to interpret input concepts using different syntactic types (e.g. noun phrase or sentence).

with atomic type as a unique variable, we need also to see duplicate constants with atomic type as a unique constant when we reduce the input λ-term. Therefore in this case the computation of the principal type is modified to see duplicate constants with atomic type as one single constant.

8.2.3.3 From derivations to inverse images

The reductions described in Sections 8.2.3.1 and 8.2.3.2 give us a program and a query. Running the program against the query gives us a set of program derivations. Kanazawa (2007) shows that, under the condition that the program and query have been built from a lexicon $\mathscr{L}$ and input λ-term using the procedure given in the previous sections, *the set of derivations of the program is isomorph to the set of inverse images of the input λ-term, where each derivation corresponds exactly to one of these inverse images.* The inverse images are recovered by substituting the rules in the derivations by constants of the abstract signature of $\mathscr{L}$. For instance, let's continue with our input λ-term $c_{love}\ c_{John}\ c_{Mary}$. The intentional database, extensional database and query are:

$$
\begin{aligned}
intentional &: S(X_3) \text{ :- } c_{love}(X_1, X_2, X_3), NP(X_1), NP(X_2). && \pi_1 \\
&\quad NP(X_1) \text{ :- } c_{John}(X_1). && \pi_2 \\
&\quad NP(X_1) \text{ :- } c_{Mary}(X_1). && \pi_3 \\
extensional &: c_{love}(1, 2, 3). && f_1 \\
&\quad c_{John}(2). && f_2 \\
&\quad c_{Mary}(3). && f_3 \\
query &: \text{?- } S(3).
\end{aligned}
$$

Figure 8.3 shows the derivation of a solution to the query and the associated inverse image. First we prune the leaves of the derivation (i.e. the facts), then we substitute the rules by the abstract constants which have been used to build them. The result is an intermediary internal structure, which is a tree of λ-terms, called a *decomposition* of the input λ-term in terms of the abstract constants of the lexicon. The inverse image associated with a decomposition is obtained simply by applying each node of the decomposition to its children and β-reducing the resulting expression.

To summarize, the function `Lexicon::parse` has three main steps:

1. Build an intentional database from the lexicon mapping.
2. Build an extensional database from the input λ-term. This steps also gives us a query.

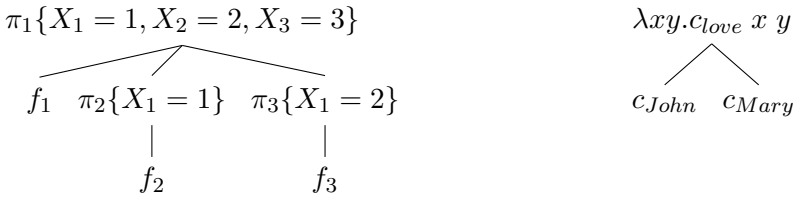


FIGURE 8.3: A program derivation (on the left) and its associated decomposition (on the right). The derivation contains the sequence of rules and facts which have been unified during the search. For the rule nodes π_1, π_2 and π_3, the final result of the unification of the variables is indicated. On the right, the decomposition is obtained by pruning the leaves of the derivation and substituting the rule nodes by the abstract constants which have been used to build them.

3. Run the program against the query, and convert the derivations of the program into λ-terms built on the abstract signature of the lexicon. The resulting set of λ-terms is returned by the function.

The first step does not depend on the input of the function `Lexicon::parse` and is cached by the instances of the class `Lexicon` (i.e. the intentional database is only computed once, during the first call to the function `parse` of a particular instance of the class `Lexicon`). An input λ-term may have an infinity of inverse images by a lexicon. In this case the function `Lexicon::parse` throws an error[16].

8.2.4 Transferring and choosing solutions

The job of the function `ComposedLexicon::translate` is to compose different transformations (mapping or inverse mapping). This operation involves the transfer of the results of a transformation to another one. The basic workflow of the function `ComposedLexicon::translate` is the following:

1. Compute the path from the origin signature to the target signature (see Section 8.1.3). This gives us a sequence of transformations (each element in the sequence corresponds to a call to either `Lexicon::realize` or `Lexicon::parse`).

2. Apply the first tranformation in the sequence to the input λ-term. Since the implementation allows to define a constant with several images by a lexicon, both `Lexicon::realize` and `Lexicon::parse` may return a collection of λ-terms. Therefore the result of the first transformation can always be considered as a collection of λ-terms.

[16] Allowing an infinite number of inverse images complexifies the implementation and the added complexity could not be justified by concrete use cases.

3. Apply the second transformation in the sequence (if any) to all elements of the previously computed collection of λ-terms. This involves multiple calls to either `Lexicon::realize` or `Lexicon::parse`, each one of these calls returning a collection of λ-terms. All the collections thus computed are concatenated. The collection of λ-terms resulting from this concatenation is the result of the second transformation in the sequence.

4. The last step is repeated for each transformation left in the sequence, using the results of the previous transformation in the sequence, until the last transformation is reached. The result of the last transformation is the result of the function `ComposedLexicon::translate`.

8.2.4.1 Merging lexicons

Some sequences of transformation can be merged into a single transformation. Indeed, when we have two consecutive calls to the function `Lexicon::realize` or two consecutive calls to the function `Lexicon::parse`, we can create a new lexicon, whose function `realize` or `parse` is the composition of the two consecutive calls. For instance, in our example we have the composition $\mathscr{L}_{tag\text{-}str}(\mathscr{L}_{der\text{-}tag}(\mathscr{L}^{-1}_{der\text{-}log}(\dots)))$. The lexicons $\mathscr{L}_{der\text{-}tag}$ and $\mathscr{L}_{tag\text{-}str}$ can be merged into a lexicon $\mathscr{L}_{der\text{-}str}$ as follows:

$$\begin{aligned}\mathscr{L}_{der\text{-}str} = \{&S, NP : o \to o,\\ &d_{love} : \lambda xyz.x\ (\text{“}loves\text{”}\ (y\ z)),\\ &d_{John} : \text{“}John\text{”},\\ &d_{Mary} : \text{“}Mary\text{”}, \}\end{aligned}$$

Using this new lexicon gives us the composition of transformations: $\mathscr{L}_{der\text{-}str}(\mathscr{L}^{-1}_{der\text{-}log}(\dots))$. Merging lexicons allows to precompute intermediate results, and only needs to be done once for all calls of the function `ComposedLexicon::translate` (in the case of two consecutive inverse mappings, it also allows to build only one extensional database instead of several ones). This optimization is done systematically, and the new lexicons are chached by the instances of the class `ComposedLexicon`. Note that the optimization applies recursively to sequences of more than two similar transformations. By applying this optimization, we ensure that we have the following property: in a sequence of transformations, a mapping (i.e. call to the function `Lexicon::realize`) is alway followed by an inverse mapping (i.e. call to the function `Lexicon::parse`), and an inverse mapping is always followed by a mapping.

8.2.4.2 Choosing a solution

In practice, we only need to generate a single text, so we need to choose among the different solutions returned by the functions `Lexicon::realize`, `Lexicon::parse` and `ComposedLexicon::translate`. Each of these function implements a choice mechanism which is active by default (i.e. by default these functions only return a single λ-term). The functions `Lexicon::realize` and `Lexicon::parse` simply choose at random among all the solutions, using a uniform distribution[17]. The function `ComposedLexicon::translate` has a different behaviour, which depends on the sequence of transformations used. There can be two kinds of composition:

- A mapping (`Lexicon::realize`) followed by an inverse mapping (`Lexicon::parse`).
- An inverse mapping followed by a mapping.

In the second case, the results of the inverse mapping operation are passed to the mapping operation, which may therefore be called several times. Since the mapping operation is fast, this does not raise any particular problem, and we can simply gather all the results of the mapping operation and then choose at random among these results using a uniform distribution. However in the first case, calling repeatedly the inverse mapping operation using the results of a mapping operation is very expensive. Therefore the default behaviour of the function `ComposedLexicon::translate` is to "cut" the accumulation of solutions in this case. When there is a mapping followed by an inverse mapping, the function `ComposedLexicon::translate` first chooses at random a solution of the mapping operation using a uniform distribution, and then apply the inverse mapping operation to this unique solution only. In this case, the final solution is not chosen uniformly on the set of all possible solutions of the function `ComposedLexicon::translate`, but an independence assumption is made in order to avoid too much computation. In practice, this independence assumption usually corresponds to the independence assumption between the document planner and microplanner modules (see Section 8.3).

8.2.5 Generation of λ-terms

The last function to analyse is the function `Signature::generate`, which takes a type and generates λ-terms of this type on a signature. In practice, the purpose of this

[17] In a complex scenario, we might imagine using other distributions, as for instance a distribution based on some language model. In practice, most industrial applications do not require such model and we can consider all alternative solutions as equivalent. Moreover the variability of the text is often an important factor, and using a distribution which favours only a few solutions by default would be counter-productive (although it might be the right choice in some particular cases).

function is to generate document plans (see Section 8.1.2). It only generates second-order λ-terms and throws an error if an infinite number of λ-terms of the given type can be built. Internally, the function is based on the same Datalog prover used for the inversion of the lexicon. The basic idea is a simplification of the reduction of the inversion of the lexicon to Datalog querying, and the function follows the same main steps:

1. Build an intentional database.
2. Build an extensional database.
3. Query the program thus built and convert the derivations of the program to λ-terms.

Since we only have to combine constants without any constraint coming from a mapping to λ-terms in an object signature, the construction of the program is much simpler. The only constraint on the combinations of λ-terms comes from their type. For each constant c of type α in the signature, we add a rule π_c to the intentional database using the following procedure:

- Create an atom for each atomic type in α and an atom for c. The arity of all predicates is zero.
- Create a rule by using the atom associated with the return type of α as the head, and the other atoms as the body of the rule.

The extensional database only contains an atom for each constant in the signature (again using predicates of arity zero). The input type is used to create an atom that is used as query. For example the intentional database, extensional database and query built when calling the function `generate(t)` on the signature Σ_{logics} is:

$$
\begin{array}{rll}
\mathit{intentional}: & t \text{ :- } c_{love}, e, e. & \pi_1 \\
& e \text{ :- } c_{John}. & \pi_2 \\
& e \text{ :- } c_{Mary}. & \pi_3 \\
\mathit{extensional}: & c_{love}. & f_1 \\
& c_{John}. & f_2 \\
& c_{Mary}. & f_3 \\
\mathit{query}: & \text{?- } t. &
\end{array}
$$

And the result of the call is the set of λ-terms:

$$\{c_{love}\ c_{John}\ c_{Mary},$$
$$c_{love}\ c_{John}\ c_{John},$$
$$c_{love}\ c_{Mary}\ c_{John},$$
$$c_{love}\ c_{Mary}\ c_{Mary}\}$$

Using a Datalog prover in order to list second-order λ-terms built on a signature is probably a bit too complex, given simplicity of the task (we do not need the unification mechanism of Datalog). However it allows to reuse the same indexation mechanism and the same search strategy in both `Signature::generate` and `Lexicon::parse`, which simplifies the implementation and is easier to optimize and maintain. Moreover, choosing to use a solver for the function `Signature::generate` might be justified in the case of types with features, which will probably need to be implemented at some point.

This concludes the detailled description of the functions `Lexicon::parse`, `ComposedLexicon::translat` and `Signature::generate`, which constitute the core functionalities of the implemented ACG based NLG framework. The focus of the implementation of this kernel is efficiency (although much is yet to be done on that point) and concision. However, a lot of linguistic knowledge is still needed in order to use the kernel for practical applications. As explained in Section 5.1, the position taken in this thesis is that ACG should be used as a kernel technology upon which other abstractions can be built in order to provide a simple and usable tools as possible for the developers of NLG systems. For instance, in Chapter 7, I have shown that we may use a schema-like representation as an overlay abstraction over an ACG kernel. Before building overlay abstractions, a simpler approach is to provide linguistic resources in the form of predefined grammars, which can be used in all applications, or in specialized fields. Such resources have been developed along with the ACG kernel and are presented in Section 8.3.

8.3 Linguistic resources

The prototype implementation provides signatures and lexicons corresponding to document planning and microplanning for English and French. The main reusable resource developed in this context is an English grammar based on X-TAG (XTAG Research Group, 2001). The other resources have been developed mainly for testing purposes and are application specific, although some have been thought as libraries which could be adapted to other applications.

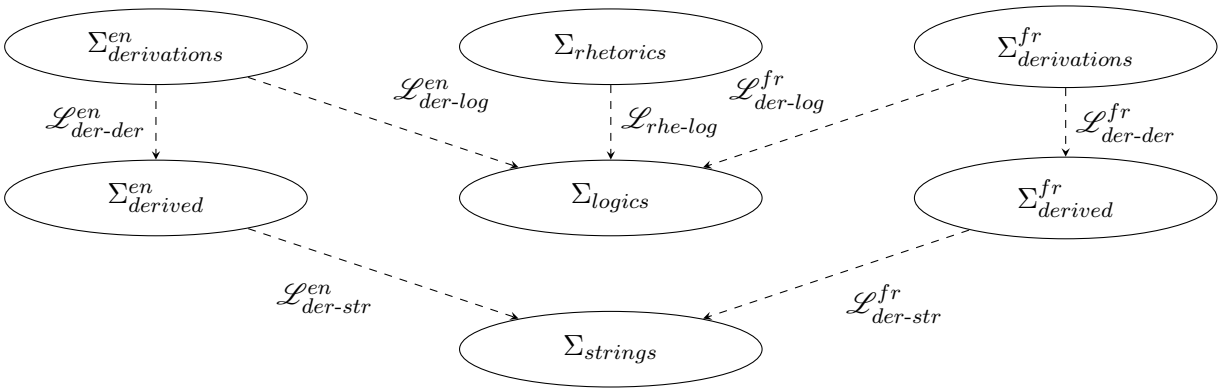


FIGURE 8.4: The signatures and lexicons provided along with the ACG kernel.

Figure 8.4 shows all the signatures and lexicons. The signature $\Sigma_{rhetorics}$ is meant for describing document plans based on RST relationships as defined in Section 7.4. The set of atomic types of $\Sigma_{rhetorics}$ contains a single atomic type τ. The constants of the signature, as well as the lexicon $\mathscr{L}_{rhe\text{-}log}$ are defined on a per application basis (see Chapter 9 for an example application using this signature). The signature Σ_{logics} defines logical sentences at the semantic level. The details of this signature are exposed in Section 8.3.1. The signatures $\Sigma^{en}_{derivations}$, $\Sigma^{en}_{derived}$ and the lexicon $\mathscr{L}^{en}_{der\text{-}der}$ represent the X-TAG grammar. These objects and the associated resources are described in Section 8.3.2. The French grammar, represented by the signatures $\Sigma^{fr}_{derivations}$, $\Sigma^{fr}_{derived}$ and the lexicon $\mathscr{L}^{fr}_{der\text{-}der}$ is a custom made toy grammar, built on the same model than their English counterparts only for the purpose of testing a multilingual setup. A true French grammar still needs to be developed. The signature $\Sigma_{strings}$ is the signature defined in Section 6.2.3, and the lexicons $\mathscr{L}^{en}_{der\text{-}str}$ and $\mathscr{L}^{fr}_{der\text{-}str}$ map the derived trees of their abstract signature as described in the same section. Finally, examples of the English syntax-semantics interface represented by the lexicon $\mathscr{L}^{en}_{der\text{-}log}$ are given in Section 8.3.3. In addition to these signatures and lexicons, three composed lexicons have been created for testing purposes (see Chapter 9):

- $\mathscr{L}^{en}_{micro}$, which contains the lexicons $\mathscr{L}^{en}_{der\text{-}log}$, $\mathscr{L}^{en}_{der\text{-}der}$ and $\mathscr{L}^{en}_{der\text{-}str}$ and can be used to translate logical sentences to strings, thereby performing microplanning tasks for the English language.
- $\mathscr{L}^{en}_{doc}$, which contains the same lexicons than $\mathscr{L}^{en}_{micro}$ plus the lexicon $\mathscr{L}_{rhe\text{-}log}$. It is used to translate document plans to strings in English.
- $\mathscr{L}^{fr}_{doc}$, which contains the lexicons $\mathscr{L}_{rhe\text{-}log}$, $\mathscr{L}^{fr}_{der\text{-}log}$, $\mathscr{L}^{fr}_{der\text{-}der}$ and $\mathscr{L}^{fr}_{der\text{-}str}$. It is used to translate document plans to strings in French.

These resources are by no means complete, and a lot is yet to be done in order to provide robust general purpose resources for microplanning and document planning. However

they allowed to conduct a few tests, described in Chapter 9, in order to show the validity of the general approach in realistic situations.

8.3.1 Semantics

The semantic resources were built in a particular context. Since the signature Σ_{logics} corresponds to the input of the microplanner, it means that one needs to build λ-terms on this signature in order to use the microplanner. Therefore, this signature is particularly "visible" from a user perspective (i.e. from the perspective of someone building a NLG system using the framework). In the long run, the goal here is to provide libraries of concepts for different specialized fields, in such a way that building an application in this domain does not require anything else but manipulating the concepts provided by the library in order to describe the output texts at a conceptual level. The constraint on the design of the semantic level is therefore to be as simple as possible, in order to require as little linguistic knowledge as possible (in other words, the usability requirement is very high). As a consequence, some expressivity has been sacrificed in order to simplify the signature Σ_{logics}, as compared for instance to Montague semantics.

Σ_{logics} contains only one atomic type c (for *concept*). Relations are represented as functions taking two arguments and action and state predicates (i.e. predicates textualized by verbs), take both their usual arguments (e.g. agent and patient) and their modifiers. For instance, take the following λ-term

$$c_{contrast}\ (c_{decrease}\ c_{turnover}\ (c_{in}\ c_{december}))(c_{improve}\ c_{turnover}\ (c_{in}\ c_{2015})\ (c_{to}\ c_{600{,}000}))$$

This conceptual representation may be textualized as: "The turnover decreased in December, but it improved during 2015 to $600,000." (see Section 8.3.3 for the complete definition of the constants and their textualisations). The constant $c_{contrast} : c \to c \to c$ represents the contrast relationship (textualized by the connector "but"). The constants $c_{decrease} : c \to c \to c \to c$ and $c_{improve} : c \to c \to c \to c$ are both textualized by an intransitive verb and take one main argument, plus a time modifier and a value modifier. Modifier concepts, such as the constant c_{in} representing a time period modifier, are textualized by prepositions and take the concept textualized by the rest of the modifying clause as their argument. Ultimately, the conceptual representation used in the tested applications follows closely the derivation tree of their textualisation (see Section 8.3.2).

Note here that in the example above, the constant $c_{decrease}$ only takes two arguments instead of three, as its type suggests. In order to improve readability, the implementation includes a mechanism called *default λ-terms*, which allows to skip unused arguments when building a λ-term on a particular signature. This mechanism works as follows:

in the signature used to build our λ-term (here Σ_{logics}), we can declare a map which associates each atomic type of the signature to a default λ-term. When a λ-term is built on this signature, the constructor can use these default λ-terms in order to complete missing arguments using the default λ-term associated with the type of this missing argument[18]. This feature can only be used where we can deduce the signature on which the λ-term is built from the context (for instance if the λ-term is declared in a lexicon, or as an argument of a `ComposedLexicon::translate` function). In Σ_{logics}, the atomic type c is mapped to the default λ-term c_{id}, which has an empty textualisation. The complete λ-term built from the example above is therefore (the constants c_{in} and c_{to} take two arguments, see Section 8.3.3):

$$c_{contrast}\ (c_{decrease}\ c_{turnover}\ (c_{in}\ c_{december}\ c_{id})\ c_{id})\ (c_{improve}\ c_{turnover}\ (c_{in}\ c_{2015})\ (c_{to}\ c_{600{,}000}\ c_{id})$$

Another mechanism used to improve readability and ease of use is the mechanism of named arguments (see Section 8.1.1). Each argument of a concept is associated with a name, which can be used to refer to this argument when building a λ-term. This allows not to rely solely on the order of the arguments, which can be hard to remember (especially if a lot of modifiers are possible). As an example, the λ-term $c_{decrease}\ c_{turnover}\ (c_{in}\ c_{december})$ will most often be written as follows in the code base[19]:

```
decrease({
  QUANTITY: turnover,
  TIME: in(december),
  VALUE: to(dollars600000)
})
```

The named arguments mechanism is combined with the default λ-term mechanism, in order to complete the missing arguments. Together they allow to manipulate concepts with less effort than the usual λ-term manipulation.

A last mechanism provided at the semantic level is the `thing` function. This function is used to dynamically create placeholders from a surface string representation. This is used when some parts of the conceptual representation are only known at runtime (see Chapter 7). For instance, in the example above, the constants $c_{decrease}$, c_{in} and c_{to} represent relatively general concepts, while the constants $c_{turnover}$, $c_{december}$ and $c_{600{,}000}$ are very specific and represent values which are usually given in the input of the application. For this kind of values, we can use the function `thing` in order to build the constants from the input information as follows:

[18] In the implementation, only atomic type can have default λ-terms, but this could be extended to complex types as well.

[19] I use here a JAVA-like syntax with a function taking a map as argument. The syntax of YML allows to use named arguments by default, so the brackets are not necessary in practice.

```
decrease({
  QUANTITY: thing("the turnover"),
  TIME: in(thing("December")),
  VALUE: to(thing("600,000 dollars"))
})
```

The strings `"turnover"`, `"december"` and `"600,000 dollars"` are input values of the NLG system. These input values are integrated into the conceptual representation using the function `thing` which creates new λ-terms and mappings in such a way that the λ-term returned by the function is a constant of type c with the string given as a parameter as its textualisation. Of course, this mechanism is not ideal, and a lot still needs to be done in order to have a smooth integration of input information into conceptual representations. Placeholders should be created at compilation time. Also, the function `thing` only takes a string. Ideally, there should be a function for each different kind of input. In this example for instance, we have a date (December), and an integer value with a unit ($600,000), which should have their own integration mechanism, without having to first textualise them. A good target mechanism would be to allow any kind of object as arguments of a constant (and more generally a λ-term), and automatically convert the objects into constants using their type information (i.e. whether it is an object of type `Date` or `Integer`). This possibility has been left for future developments.

Definitions at the semantic level have been used in two test applications (see Chapter 9). A special effort has been made for one of these applications, in order to build a small library of about forty concepts[20], reusable in all applications in the domain of business intelligence (Den Os, 2015). These concepts have been aggregated from the analysis of half a dozen existing application in the domain. However to this day the resulting library has only been tested on a single application (though a very complex one), so some work is probably still to be done in order to produce a stable library.

8.3.2 English grammar

The signatures $\Sigma^{en}_{derivations}$, $\Sigma^{en}_{derived}$ and the lexicon $\mathscr{L}^{en}_{der\text{-}der}$ follow the ACG representation of TAG defined in Section 6.2.2: $\Sigma^{en}_{derivations}$ contains syntactic types corresponding to substitutions and adjunctions ($\{S, S_a, NP, NP_a, VP, VP_a, V, V_a, \dots\}$), and constants corresponding to specific words or expressions. $\Sigma^{en}_{derived}$ only contains the atomic type τ, and constants for the different possible nodes of a syntactic tree ($\{S_1, S_2, NP_1, NP_2, N_1, VP_2, \dots\}$).

[20] Most entities, such as dates, numbers, etc., are application specific, and even specific to a particular input. Therefore most of the reusable concepts correspond to actions such as “increase” or “decrease” and modifiers.

The abstract constants of $\mathscr{L}^{en}_{der\text{-}der}$ are mapped to (λ-terms representing) syntactic trees built on $\Sigma^{en}_{derived}$.

The predefined English grammar is a set of *tree templates* built on $\Sigma^{en}_{derived}$. A tree template is a tree where the leaf corresponding to the anchor of the tree has been replaced by a variable. A tree template can be instantiated by applying it on a leaf node (i.e. a constant of type τ). For instance, one of the tree templates for an intransitive verb is:

$$\lambda x\ np_1\ s_a\ vp_a.s_a\ (S_2\ np_1\ (vp_a\ (VP_1\ (V_1\ x))))$$

This template can be instantiated by applying it on the constant $t_{improved} : \tau$ representing a past participle anchor for the verb "to improve", resulting in the tree:

$$\lambda np_1\ s_a\ vp_a.s_a\ (S_2\ np_1\ (vp_a\ (VP_1\ (V_1\ t_{improved}))))$$

The grammar has been adapted from X-TAG (Den Os, 2015)[21]. It contains about one thousand tree templates, divided into 57 verbal tree families and 123 non verbal isolated trees. The grammar is defined using a metagrammar (Candito, 1999), using the tool XMG (Crabbé et al., 2013). This tool outputs a grammar in XML format, which is then translated into YML λ-terms using a custom converter written in JAVA.

While the metagrammar contains information about features, this information is ignored by the converter. This is a limitation of the actual implementation, as types with features have not been implemented yet. The result is that some phenomenons, in particular morphological ones like agreements and tense, are not handled by the YML version of the grammar. This didn't pose a problem for the example application used to test the prototype. In those cases the tense is fixed, and there were no agreements that could not be hard coded. However it is not sufficient in the general case, and in particular when generating French, which has more agreement rules than English. While the ideal approach would be to include all the features in the grammar, an alternative solution can be used. The signature $\Sigma_{strings}$ contains constants which are associated with YML objects[22]. Usually, these objects are strings, but we may use any other kind of object. For instance, one could use a textualisable reference, in order to apply a referring expression generation algorithm in post treatment. In general, the "string" which results from calling the ACG module can be considered as a linked list of objects, which can then be processed by other modules down the line. This can be used in

[21] The grammar follows closely the X-TAG definitions. The adaptation concern some technical features, plus a few trees for discourse level syntactic trees. The metagrammar, as well as the JAVA converter and the semantic library of the last section have been created by Den Os (2015).

[22] All constant are associated with a symbol which identifies them. This symbol can be any YML object.

particular to apply agreement rules. This shows that although the ACG module does not copes with all the phenomenons we might need yet, the fact that it is integrated in a larger NLG frameworks allows to delegate some problems to existing functions in a relatively simple way.

In an application, the grammar is used by creating constants in the signature $\Sigma^{en}_{derivations}$ and linking them to tree templates instantiated with an anchor. For instance, an application may define the constant $d_{improved} : NP \rightarrow S_a \rightarrow VP_a \rightarrow S$ with $\mathscr{L}^{en}_{der\text{-}der}(d_{improved}) = \lambda np_1\ s_a\ vp_a.s_a\ (S_2\ np_1\ (vp_a\ (VP_1\ (V_1\ t_{improved}))))$. So using the grammar still involves a lot of linguistic knowledge, as one need to define the vocabulary and instantiate the tree templates of the grammar that correspond. In the tested applications, the vocabulary has been created from scratch. However, one could create a general purpose dictionary from an existing linguistic resource such as the Lefff (Sagot, 2010), which could then be used in any application. This possibility has been left for future research.

In order to ease the creation of the vocabulary of an application and to integrate input information, a few dynamic mechanisms have been added to the grammar. First, a few function allow to instantiate several tree templates at once by grouping them in families. For instance, we have the function `transitiveVerb`, which takes a string, and returns a collection tree templates (for instance active and passive forms) instantiated with the input string. This mechanism is not ideal, and has been implemented temporarily in order to compensate for the lack of types with features. Indeed, there is no particular reason to define tree families dynamically, and a better mechanism would be to use types with features in order to defined static parametrized sets of tree templates, which could then be specialized for different usages. Another dynamic mechanism, which is more justified, is the possibility of creating constants and mappings dynamically, in order to incorporate input information into the grammar. There are two functions used for this purpose in practice: `createAnchor` and `createNP`. The first function takes a string and returns a constant of the signature $\Sigma^{en}_{derived}$ of type τ, where the image of this constant by $\mathscr{L}^{en}_{der\text{-}str}$ is a constant of type $o \rightarrow o$ associated with the input string. This function is used to dynamically create anchors from input information, or in the case of the dynamic tree families described above from user specifications. The function `createNP` is a wrapper of `createAnchor` which takes a string returns a constant d of the signature $\Sigma^{en}_{derivations}$ where $\mathscr{L}^{en}_{der\text{-}der}(d) = createAnchor(s)$ (where s is the input string). This function is in turn used by the function `thing` described in the Section 8.3.1 on semantics. Together, the three functions `thing`, `createNP` and `createAnchor` represent the dynamic aspect of the grammar, and allow to include input information into the ACG module.

8.3.3 Text variability

Building a NLG system using the linguistic resources described so far includes two separate tasks:

- First, one needs to define the concepts used in the application. This involves declaring the right constants at the semantic level and at the derivation level and adding the right mappings in the lexicons. This task is subject to automation or semi-automation in several ways (e.g. by defining a predefined vocabulary or library of concepts).
- Second, one needs to associate λ-terms built on the the signature $\Sigma_{semantics}$ with the different messages of the application (for now we consider that the document planning phase is done by an external module). This step is done dynamically, using the NLG framework language.

Ideally, the ACG definitions allow a faithful representation of the different messages of the application, while associating many textual variations with these representations. However in practice such situation would require very high level concepts. The choice made here (though it is by no means a definitive one), is to use a relatively low level semantic representation. Therefore the semantic level only allows limited textual variations for a particular concept. This implies that some of the effort for defining textual variations may be reported on a higher level, which correspond here to the second task, where the NLG system builds the semantic representation of a particular message (i.e. the NLG system may build several "synonymous" semantic representation for a particular message in order to improve variability). While this situation is not ideal, it seems that in practice most applications have simple enough needs in terms of output variability, for it to be a good compromise between expressivity and simplicity. As an illustration of some possibilities of textual variation using the ACG definition of the concepts, let's use the λ-term shown in Figure 8.5 from the business intelligence domain, corresponding to the meaning of "In December the turnover decreased, but it improved to 600,000 dollars during the year.".

In this example, we have six concepts defined statically: `contrast`, `decrease`, `improve`, `in`, `during` and `to`, that we shall note $c_{contrast}$, $c_{decrease}$, $c_{improve}$, c_{in}, c_{during} and c_{to} respectively. The other concepts are built dynamically using the function `thing`. Note that here this function takes as parameter the object `turnover` which is not a string. This illustrates how one can mix referring expression generation with the ACG module. Here the object `turnover` is an object of an existing YML framework which is recognized by the referring expression generation module as an entity. Therefore it will be processed

```
contrast(
  decrease({
    QUANTITY: thing(turnover),
    TIME: in(thing("December")),
  }),
  improve({
    QUANTITY: thing(turnover),
    TIME: during(thing("the year")),
    VALUE: to(thing("600,000 dollars"))
  })
)
```

FIGURE 8.5: Example microplanner input from the business intelligence domain. The object `turnover` is an YML object recognized processed by the existing NLG framework as a reference to an entity, and is used to perform referring expressions generation, after the execution of the ACG microplanner.

by this module wherever it is positioned in the text after the ACG module has done its job, resulting either in the textualisation "the turnover" or "it". In details, given a string s (or any object that can be textualized), the function `thing` creates the following constants and mappings:

$$
\begin{aligned}
&c_s : c \to c \to c \\
&d_s : NP_a \to N_a \to NP \\
&t_s : \tau \\
&s_s : o \to o \\
&\mathscr{L}^{en}_{\text{der-log}}(d_s) = c_s \\
&\mathscr{L}^{en}_{\text{der-der}}(d_s) = \text{NXN}(t_s) \\
&\mathscr{L}^{en}_{\text{der-str}}(t_s) = s_s
\end{aligned}
$$

Where NXN $= \lambda a\ np_a\ n_a.np_a\ (NP_1\ (n_a\ (N_1\ a)))$ is the noun phrase tree template of the grammar[23]. Figure 8.6 shows the definitions associated with the concepts `contrast`, `decrease`, `in`, and `to` (the definitions for the concept `improve` are similar to the ones for the concept `decrease` and the definitions for the concept `during` are similar to the definitions for the concept `in`). These definitions illustrate different kinds of variations. The concepts $c_{contrast}$ and $c_{decrease}$ both have several inverse image by the lexicon

[23]The names of the tree follow the X-TAG notation. Note here that according to the grammar, when the noun phrase contains a determinant, the determinant should be anchoring another tree and then be adjoined to the tree NXN. This can be done relatively easily by separating the determinant and the noun in two strings anchoring two trees combined together (the combination of the two trees forms a new tree template, which has two anchors; named arguments are very useful for building new tree templates from existing ones). However the input information may provide strings already containing both the determinant and the noun, in which case we simply treat it as a noun as in the example.

$\mathscr{L}^{en}_{der\text{-}log}$, representing synonymous wordings. The textualisations of $c_{decrease}$ are "perfect" synonyms because they all use the same tree template returned by the function `intransitiveVerb`[24] (see the tree template in Section 8.3.2). The textualisations of $c_{contrast}$ however, use two different tree templates. The function `conjunction` returns the tree template $\lambda a\ s_1\ s_2.S_4\ (s_1\ t_{comma}\ a\ s_2)$, which concatenates two sentences with a comma and a conjunction and the function `adverbSentence` does the same thing but with a period and an adverb. The concept c_{in} uses a different kind of variation, playing on the position of the complement (left or right). The function `sentenceModifier` returns two tree templates, one adjoining a complement on the left of a sentence node and the other one the right. For the concept c_{to}, the function `prepositionalComplement` returns a tree template adjoining a complement on the right of a verbal phrase node.

To summarize, we have three different wordings for the concept $c_{decrease}$, two for the concept $c_{contrast}$, two different positions for the concept c_{in} and only one for the concept c_{to}. Additionally, not shown on the Figure 8.6, we have three different wordings for the concept $c_{improve}$ ("improved", "bettered" and "progressed") and two positions for the concept c_{during}. The possibilities multiply out to give 72 textual variant for our original logical sentence. More interesting variations are possible by adding a variant to the concept $c_{contrast}$ as follows:

$$
\begin{aligned}
&d_{despite} : NP \rightarrow S \rightarrow S \\
&\mathscr{L}^{en}_{der\text{-}log}(d_s) = c_{contrast} \\
&\mathscr{L}^{en}_{der\text{-}der}(d_s) = \texttt{commaSentenceModifier}(\text{“}despite\text{”})
\end{aligned}
$$

This variant allows to generate sentences like: "Despite the decrease of the turnover in December, it improved to 600,000 dollars during the year". in order to use this variant, one needs first to add variants to the concepts $c_{decrease}$, c_{in} and to the function `thing` in order to be able to express them either as a noun phrase or as a modifier of a noun phrase. For this purpose we add the following constants to the signature $\Sigma_{derivations}$: $d_{the\_decrease} : NP \rightarrow NP_a \rightarrow NP_a \rightarrow NP$, $d_{in\_np} : NP \rightarrow NP_a \rightarrow NP_a$ and $d_{to\_np} : NP \rightarrow NP_a \rightarrow NP_a$ and map them to their respective tree templates (the function `thing` is modified in order to return concepts with two variants, one being a noun phrase and the other a noun phrase modifier with the preposition "of"). Using three different wordings for the textualisation of the concept $c_{decrease}$ (e.g. "the decrease", "the reduction", "the diminution"), we arrive at a total of 90 variants for our example conceptual sentence. Figure 8.7 shows two of these possible variants, along

[24] As indicated before, in practice there should be several trees in a family, and the function `intranstiveVerb` should allow to select different subsets of the family. For now, only the variants of passive and active forms for the transitive verbs have been tested.

`contrast`
$c_{contrast} : c \to c \to c$
$d_{but} : S \to S \to S$
$d_{however} : S \to S \to S$
$\mathscr{L}^{en}_{der\text{-}log}(d_{but}) = c_{contrast}$
$\mathscr{L}^{en}_{der\text{-}log}(d_{however}) = c_{contrast}$
$\mathscr{L}^{en}_{der\text{-}der}(d_{but}) = \texttt{conjunction}(\text{“}but\text{”})$
$\mathscr{L}^{en}_{der\text{-}der}(d_{however}) = \texttt{adverbSentence}(\text{“}however\text{”})$
`decrease`
$c_{decrease} : c \to c \to c \to c$
$d_{decreased} : NP \to S_a \to VP_a \to S$
$d_{lowered} : NP \to S_a \to VP_a \to S$
$d_{diminished} : NP \to S_a \to VP_a \to S$
$\mathscr{L}^{en}_{der\text{-}log}(d_{decreased}) = c_{decrease}$
$\mathscr{L}^{en}_{der\text{-}log}(d_{lowered}) = c_{decrease}$
$\mathscr{L}^{en}_{der\text{-}log}(d_{diminished}) = c_{decrease}$
$\mathscr{L}^{en}_{der\text{-}der}(d_{decreased}) = \texttt{intransitiveVerb}(\text{“}decreased\text{”})$
$\mathscr{L}^{en}_{der\text{-}der}(d_{lowered}) = \texttt{intransitiveVerb}(\text{“}lowered\text{”})$
$\mathscr{L}^{en}_{der\text{-}der}(d_{diminished}) = \texttt{intransitiveVerb}(\text{“}diminished\text{”})$
`in`
$c_{in} : c \to c \to c$
$d_{in} : NP \to S_a \to S_a$
$\mathscr{L}^{en}_{der\text{-}log}(d_{in}) = c_{in}$
$\mathscr{L}^{en}_{der\text{-}der}(d_{in}) = \texttt{sentenceModifier}(\text{“}in\text{”})$
`to`
$c_{to} : c \to c \to c$
$d_{to} : NP \to VP_a \to VP_a$
$\mathscr{L}^{en}_{der\text{-}log}(d_{to}) = c_{to}$
$\mathscr{L}^{en}_{der\text{-}der}(d_{to}) = \texttt{prepositionalComplement}(\text{“}to\text{”})$

FIGURE 8.6: Example definition for the microplanner. The functions `conjunction`, `adverbSentence`, `intransitiveVerb`, `sentenceModifier` and `prepositonalComplement` are tree family functions. They return one or several trees templates instantiated with the given parameter. These functions create constants in the signatures $\Sigma_{derived}$ and $\Sigma_{strings}$ corresponding to the anchors of the trees.

with their corresponding intermediary λ-terms built on $\Sigma^{en}_{derivations}$ and $\Sigma^{en}_{derived}$. The kind of variation shown in this example (expressing a concept either as a sentence or as a noun phrase) is typically difficult to model correctly using template based approaches, but have a natural expression using a grammar based NLG framework.

Other variations are possible, such as the variation between the active and passive form for a transitive verb. However we are rapidly limited by the fact that the semantic representation is very close to the level of derivation trees, and do not allow rewritings more complex than basic words reordering. Also the syntax-semantic interface does not handle recursivity, and problems arise when concepts have many possible modifiers. In this case, the solution adopted here is to artificially augment the number of arguments of the tree templates in order to be able to receive the different modifiers[25].

Overall, the possible variations still allow a good variability, especially when compared with more "templatized" systems, where variability based on words reordering is already considered complex (though still possible). When using linguistic resources, once identified the possible variations can be applied very easily to many situations. Two people with advanced linguistic knowledge, but no prior knowledge about ACG have used the linguistic resources in order to generate sentences using the ACG kernel. The main feedbacks are that it takes much to learn the basics, but that it is rather efficient once it is done. Note that we have only used a small fraction of the grammar, and a lot is yet to be done both in exploring the possibilities offered by linguistic resources in practice and the methodology and tools for adapting the linguistic resources to particular applications.

[25] The problem here is very specific to the way the semantic level has been designed, and comes from the fact that the modifiers of the concepts are not built in a recursive manner, but all apply directly to the main predicate. This choice has been made in order to simplify the construction of λ-terms at the semantic level, but it implies a lot of rigidity. An alternative is to use Montague semantics such as described in (Pogodalla, 2009).

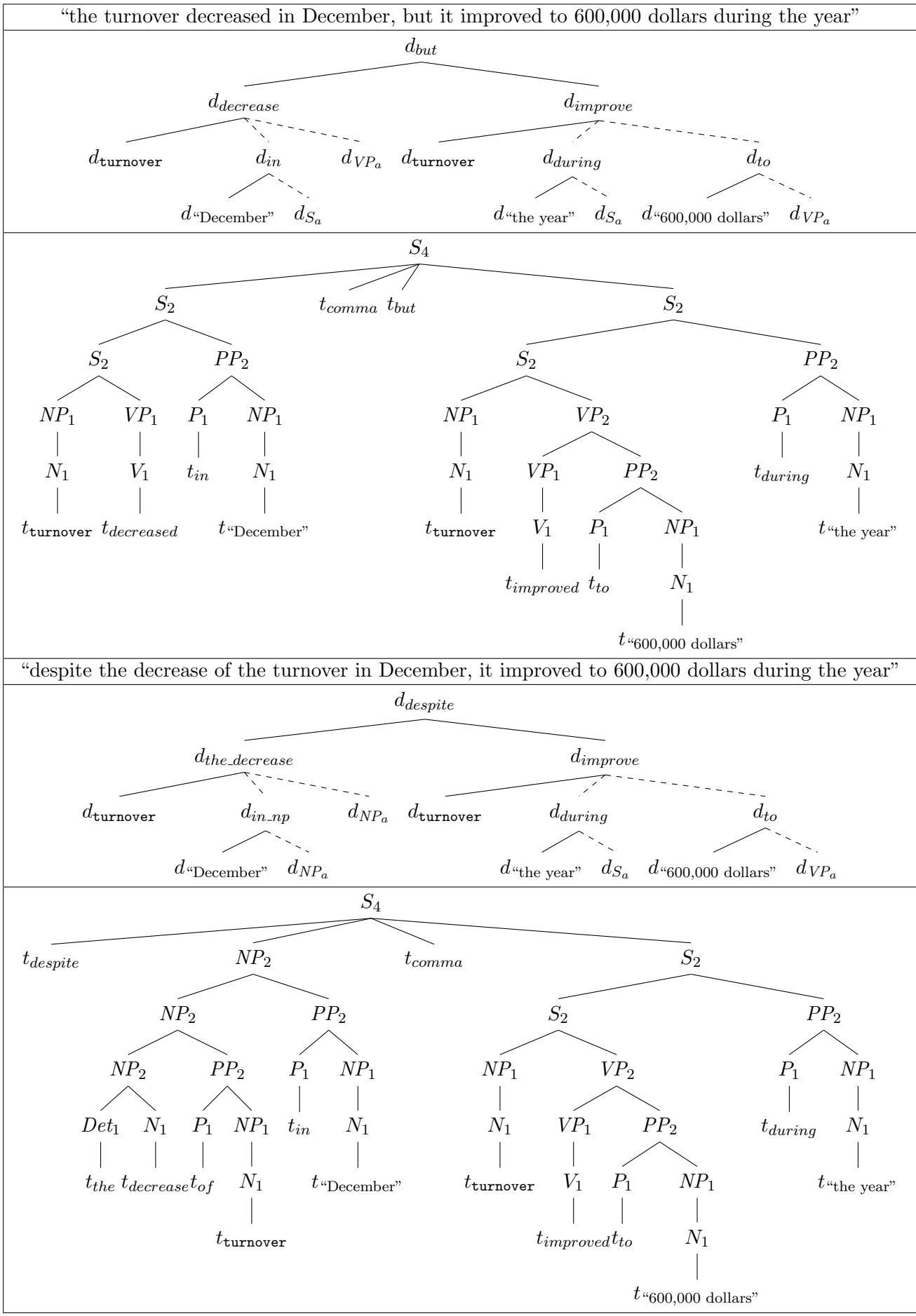


FIGURE 8.7: The derivation tree and derived tree of two possible output variants of the inpur λ-term of Figure 8.5. They represent λ-terms built on $\Sigma^{en}_{derivations}$ and $\Sigma^{en}_{derived}$. The strings shown for each example are the output of the micropanner after being processed by the referring expression generation module.

Chapter 9

Results and perspectives

The ACG based NLG framework presented in Chapter 8 has been evaluated in realistic situations, in order to validate that the approach is indeed a good candidate for being the core technology of a general purpose NLG framework. The evaluation is presented in Section 9.1. Sections 9.2 and 9.3 present the two applications which have been used for evaluating the implementation and the conclusions drawn from this evaluation. Section 9.4 presents the perspectives for future research, and Section 9.5 concludes this thesis.

9.1 Evaluation

Evaluating a NLG framework is different from evaluating a NLG system. The evaluation of a NLG system usually relies on the comparison between automatically generated text and hand written ones against a particular goal or metric. For instance, Reiter et al. (2003) compare automatically generated text and hand written ones as to their effectiveness for persuading people to stop smoking. This kind of task based or goal based evaluation is specific to a domain (here public healthcare), and involves humans in real world experiments or in more controlled environments (Portet et al., 2009). Alternatively, human ratings can be used in order to measure metrics such as readability, coherence, etc. (Hunter et al., 2012). Machine learning based NLG tend to use automated metrics measurements, such as BLEU (Papineni et al., 2002) or METEOR (Lavie and Agarwal, 2007).

All of these evaluation methods can only be used on a particular NLG system. They measure how well a NLG system achieves its purpose, which is to generate human-like, useful texts. On the other hand, the purpose of a NLG framework is to allow users

to build such NLG systems easily. It is generally possible to achieve a similar NLG system using different NLG frameworks, the difference being in the amount of effort needed to achieve this result[1]. Therefore the evaluation of NLG systems, which only evaluates the resulting NLG system, is not very helpful for evaluating and comparing NLG frameworks.

It is not easy to evaluate a NLG framework, as it is expected to handle many different scenarios, and a large part of the advantages it might have over another framework in terms of simplicity are somewhat based on subjective appreciations, and fuzzy concepts such as the amount of knowledge needed in order to use the framework effortlessly. The metrics used for evaluating a NLG framework often come from the software design world. For instance, Reiter and Dale (2000) defend a particular modular architecture as both correct (i.e. including all the tasks needed to perform NLG) and efficient, and Mellish et al. (2004) describe their approach using software architecture concepts such as genericity and flexibility. The difficulty here lies in the fact that different authors will often rely on different sets of software quality metrics. In order to keep the evaluation as neutral as possible, the choice made in this thesis is to rely on a standard set of software quality metrics introduced by Meyer (1988), and introduced in Section 5.1, namely: correctness, extendibility, reusability, compatibility, robustness, ease of use, efficiency and verifiability.

Correctness, the ability of the software to perform the task it has been created for, can be interpreted in the context of a NLG framework as expressivity. Indeed, the broad specification of a NLG system is to be able to define NLG systems for any kind of application, and therefore to be able to express many different situations. We know, from the research done in formal linguistics, that the framework will be able to represent pretty much anything we might need to represent in practice. The sentences generated in commercial applications are usually relatively common ones in terms of syntactic construction, therefore the goal in this thesis is not to show that we can represent complex linguistic phenomena using ACG. We have already seen some simple examples of modelisation of output texts in Chapter 8, and we know that more elaborate representations can be used if ever needed. On this point, an ACG based NLG framework does not particularly distinguish itself from a "classical" NLG framework[2].

[1] Some NLG frameworks may be intrinsically limited, or have a poor expressivity, which may impact the NLG systems built from it. However any useful NLG framework has enough expressivity to represent pretty much anything that might be needed by a realistic NLG system.

[2] Here and in the following, a "classical" NLG framework means a framework based on an imperative programming style, with the power of expressivity of a general purpose programming language, and following (more or less) the architecture from (Reiter and Dale, 2000). The classical NLG framework archetype, also including the actual Yseop technology, is used to compare the advantages and drawbacks of an ACG based NLG framework.

Extendibility here means the ease with which a NLG system built using the implemented framework can be modified, whenever there is a change in the specifications of the system. In practice, extendibility comes from a good modular design. An ACG based NLG framework forces a modular design, by using levels of abstractions as its basic component. In this aspect, it therefore results in NLG systems which are easily extendible. The same property also ensures a good reusability. The principle of using linguistic resources in order to describe the output texts ensures that most of the definitions (at least the ones up the syntactic level) are not application specific and can be reused in other applications (for instance the vocabulary). The modularity of an ACG based NLG framework gives it an advantage in terms of extendibility and reusability over a more classical NLG framework. A classical NLG framework also use a modular design, but the fact that all levels of abstraction use the same underlying representation pushes the modularity of an ACG based NLG framework beyond what can be done with a classical NLG framework. However it is not clear yet, as to how one could better benefit from this advantage in terms of processes and development tools.

It is not very meaningful to evaluate the compatibility and robustness properties in our particular context. For these properties, we rely on the fact that the framework has been implemented in an existing technology, which already has them, as any other industrial software. A similar remark can be made about the verifiability requirement, which mostly depends on the test environments and debugging tools made available by the existing technology.

Now we come to the properties which usually cause problems in linguistically motivated systems: ease of use and efficiency. There is no way around the fact that an ACG based NLG framework is hard to use in practice. Manipulating λ-terms in order to define linguistic knowledge requires a long formation. The bet made in this thesis, is that we can use ACG as a kernel technology, and build layers of abstraction over this kernel in order to allow users to develop NLG systems without ever having to manipulate any λ-term and with having as little linguistic knowledge as possible. This goal is far from being achieved yet, and would require much more time and resources than what has been allowed for this work. For the time being, the objective is to allow people with sufficient knowledge (i.e. with a formation in computational linguistics or computer science) to use the "raw" framework in order to evaluate if it can be used in a useful way for practical application. In other words, we are not at the point where we want to compare the ACG based NLG framework to the existing technology[3], but at the point

[3] As an example evaluation, one could develop the same application with the implemented framework and with the existing technology, and compare the time spent for developing the application, the complexity of the resulting code, etc.. A full evaluation would require several participants and metrics for the productivity and simplicity of the resulting code.

where we want to know if the implemented framework has any chance of competing with it. For this purpose, we want to check that there are no crippling problems associated with the ACG based NLG framework and in particular:

- We want to check that we can use the ACG based NLG framework in realistic applications, without involving too many workarounds and edge case scenarios, and without specializing the linguistic resources so much that we loose the possibility of reusing them from one application to the other. On the other hand, we also need to check that if the linguistic resources do not allow to easily represent part of the text of the application, we always have a way to fall back on alternative solutions.
- A crippling problem which often comes along linguistically motivated systems is the efficiency problem. We need to make sure that we can generate text fast enough for the framework to be usable in real time applications.

Sections 9.2 and 9.3 present two setups which have been used to check whether the implemented framework satisfies the above conditions. The answer to these questions do not come in a simple yes or no answer, but are subject to interpretation. This comes from the following reasons: about the efficiency of the framework, there are known optimizations which can be implemented and can change in a significant manner the results of the tests. The expectations about the efficiency of the framework as it is implemented is that it should be slower than the existing technology, but not so much slower that it would seem impossible to make it fast enough with the known possible optimizations. A similar remark applies to the ease of use of the framework. There may be difficulties and workarounds needed in order to build a realistic application with the framework, but we should take into account the possible evolutions of the framework and the linguistic resources which could solve these problems[4].

9.2 Business intelligence

For the first evaluation, we use an existing application in the business intelligence domain. The application receives the data of a graphical representation along with data analysis results, and generates a description of the graph. Figure 9.1 shows an example descriptive text along with a graph. This application is a Yseop product and forms a good basis for testing the speed of the implemented framework, as it is a real time

[4]Checking that the framework does not contain any inherent crippling problem is also a way experimenting in order to find what would be the most useful evolutions of the framework, so we are not in a context where we want to assess the value of a definitive, stable version of the software.

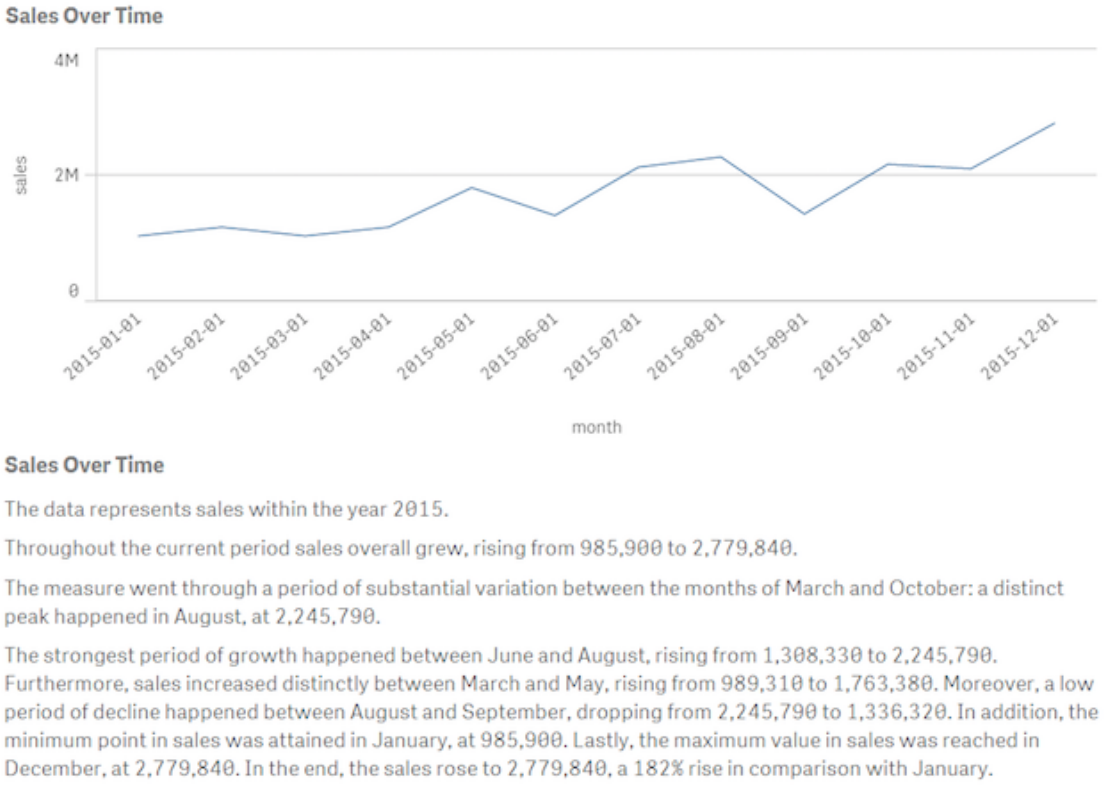


FIGURE 9.1: Business intelligence application example. The data of the graph on top is analysed by the NLG system in order to produce the descriptive text below it.

application with response time constraints. The general workflow of the application is the following:

- First the input data is analysed using a numerical analysis and pattern detection module, which detects events like maximums, peaks, stagnations, etc..
- The detected events are then filtered and organized into a document plan by a document planning module.
- Finally the events are textualized by a microplanning (and realisation) module.

The evaluation is only on the last step, the microplanning module, which involves the most computation for the text generation part of the system (the document planner for this application is relatively simple). The test consists in comparing the execution speed of the existing NLG system and the implemented prototype for generating the text corresponding to one particular document plan. A possible text corresponding to the chosen document plan is the following:

This graph shows sales by country within the month range 1 - 12.

Overall Trend *Sales fluctuated, oscillating between 154,134 and 364,471. All the contributors followed the same trend as the total of all sales. Do note that the lowest trough took place in 9, at 154,134. A sharp peak took place in 10, at 334,918. Lastly, the strongest period of decline happened between 8 and 9, from 360,927 to 154,134.*

Breakdown per country Now that we have looked at the overall trend, let's look at each country separately.
The United States' sales frequently vary. A notable trough took place in 9, at 86,531. A sharp peak took place in 10, at 258,498. Lastly, the minimum point in sales was reached in 11, at 80,434.
Canada's sales represented 13.15 % of the total. The minimum point in sales was reached in 7, at 22,986. The strongest period of decline happened between 5 and 7, from 43,986 to 22,986. Lastly, the longest period of growth happened between 7 and 12, rising from 22,986 to 48,312.
Mexico's sales represented 8.18 % of the total. The lowest trough took place in 2, at 16,834. The highest peak took place in 8, at 29,434. Lastly, the strongest period of decline happened between 8 and 10, from 29,434 to 19,438.
The last country (Nicaragua) only accounts for 7.31 % of the total.

The sentences in italic font are sentences dynamically generated, the other ones being static, and are the only ones concerned by the test. There is a total of 19 dynamic sentences. The relative order of the sentences does not change between two consecutive calls, and each sentence has between 2 and 10 possible variations (with a mean of 4 possible variations per sentence). For the evaluation, the semantic representation of each of these sentences has been created by hand (more precisely, the semantic representation of the messages corresponding to these sentences). The composed lexicon $\mathscr{L}^{en}_{micro}$ (see Figure 8.4) is used in order to translate these semantic representations into text. So the test NLG systems performs 19 calls to the function `ComposedLexicon::translate`, with the origin signature being Σ_{logics} and the target signature being $\Sigma_{strings}$. The time spent by the test NLG system is compared in Figure 9.2 with the time spent by the existing NLG system for generating the same text (modulo local variations) from a document plan[5]. The time spent by the implemented framework is further divided between the time spent in the function of inversion of a lexicon (`Lexicon::parse`), in the mapping function (`Lexicon::realize`) and in the rest initialization phase (here I count the creation of the the database for the inversion of the lexicon as an initialization

[5]All the tests have been realized on a Ubuntu 16.04 LTS virtual machine, with an Intel i7-6700HQ CPU (2.6 GHz) and 8 Go RAM.

BI system	ACG framework
1102 ms	570 ms

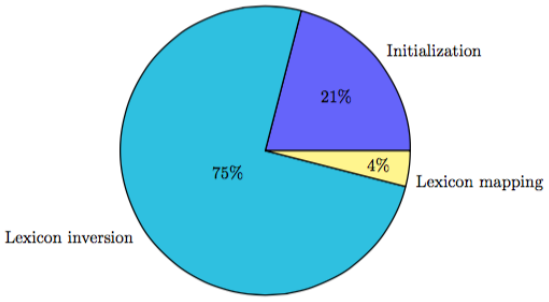


FIGURE 9.2: Results of the performance test of the framework. The time spent for generating 19 sentences is compared between the existing NLG system for business intelligence and the implemented framework. The pie chart shows the decomposition of the time spent by the ACG framework between the initialization of the system (creation of objects, composition of lexicons, creation of the databases for the inversion of a lexicon, etc.), the operation of inversion of a lexicon and the mapping operation.

step, since it is done only once for all calls to the function `Lexicon::parse`). For the inversion of the lexicon step, the mean number of derivations found by the Datalog prover is 2.5 (ranging between 1 and 6).

Surprisingly, the result of the evaluation is that the implemented framework is about twice faster than the existing technology on this particular test. However this result should be taken with precaution, as several factors can explain it. There are several ways of doing the same thing using the existing Yseop technology, and there could be inefficiencies in the way the application has been built which can explain a large part of the difference in performance. On the other hand, there is still room for improvement for the ACG framework. If we look in more details the time spent by the implemented framework on different operations, we find that the operation of inversion of a lexicon takes three quarters of the overall time spent. This comes as no surprise, since this operation involves the Datalog prover, which is the component with the larger margin for improvement. The initialization phase, which includes the creation of the databases takes also some time (about a fifth). Part of the initialization effort can certainly be reported during the compilation phase, so there is also room for improvement here. The current estimation is that we could probably cut the response time of the ACG framework by two or three by applying the right optimizations.

While being subject to caution, the result of this test is still very positive. Overall, what this result shows for sure is that the ACG framework is definitely in the same order of magnitude than the existing technology when it comes to microplanning, and that it is elligible for being used in production application (as it is the case with the business intelligence application used for this test). This clears the common apprehension that

linguistically motivated systems cannot compete with more "pragmatic" ones. The next step would be to build a benchmark of applications in order to test more systematically the ACG framework against the existing technology, while making sure that both are as optimized as they can.

Another interesting "side effect" of this test, is that it has been realized by an employee of Yseop with no particular knowledge about ACG (but with formation in computational linguistics). His task was to create the constants of the signatures Σ^{en}_{logics} and $\Sigma^{en}_{derivations}$, build the lexicons $\mathscr{L}^{en}_{der\text{-}log}$ and $\mathscr{L}^{en}_{der\text{-}der}$ using the provided TAG for english (see Section 8.3.2), and to build the input λ-terms for generating the output shown above. This was the occasion of checking the (subjective) difficulty of using the framework. The result is that it needs a lot of bootstrap knowledge, but becomes quite efficient after enough effort has been invested into it. In practice, it seems that there are still a few workarounds which need to be used in order to use the framework in realistic applications. In particular, the tree templates of the grammar often need to be adapted in order either to add arguments (the problem comes from the rigidity of the semantic representation, see Section 8.3.3) or to combine tree templates together in order to have coarser grained syntactic representations[6]. Research is still to be done in order to find the appropriate processes and tools for easing the creation of linguistic resources.

9.3 Argumentation

The purpose of the second evaluation is to explore the usage of the ACG framework for building a complete application, including document planning and multilingual microplanning. For this evaluation, an existing application has been rebuilt using the implemented framework and the linguistic resources presented in Section 8.3. The application that has been rebuilt is a car selling application, which recommends cars for potential buyers on a second hand car selling web site. This application has been chosen because it seemed to justify the use of the document planning technique described in Section 7.4. The car selling application uses a recommender system in order to suggest cars to a user. The general workflow of the application is as follows:

- The user is prompted through a web based interface for information about itself (age, number of children, favourite color and brand, number of kilometers per year, etc.).

[6] fine grained representations add unnecessary complexity for applications with relatively poor syntactic complexity. In this case, noun phrases could always be manipulated as a whole without having to manipulate nouns and determiners separately.

- The information is stored in a user model, along with additional information that the system can infer from the context.
- The user model is used to filter a database of cars. The filtering is done as a multi-objective optimization problem, by attributing a set of scores to each vehicle. Each score can be positive or negative and is based on a combination of a user need (from the user model) and a characteristic of the vehicle. The scores are combined through a weighted sum and the vehicles are sorted according to the total score. The ten best cars are then selected.
- Each one of the ten best cars is passed to the NLG module, along with its scores. Each car description is considered as independent from the others. A short introductory text is added to introduce the ten cars, including information such as "We have selected 5 Renault, 4 Peugeot and 1 Ferrari.". Here I focus on the description of a single vehicle.

More precisely, the input of the NLG module responsible for the description of a vehicle is modelled as a set of *arguments*. An argument is defined by four elements:

- A symbol identifier representing the argument, generally named after a user need, such as "preferred color" or "required option".
- A characteristic of the vehicle, such as its color or its price (or eventually an inferred property, such as the difference between the price of the vehicle and the budget of the user).
- A score between -1 and 1, which is one of the scores used by the recommender system to rank the different cars.
- A topic symbol, which is used to infer rhetorical relationships between the arguments.

The set of all possible arguments has been defined by the developper of the application. The set of arguments associated with a particular vehicle is computed using rules such as "if the user asked for a black car and the car is black, then add the argument {"Chosen color", color, +0.2, APPEARANCE}", where "Chosen color" is the identifier of the argument, color is the characteristic of the vehicle, +0.2 is the score of the argument and APPEARANCE the topic symbol. These rules are applied either by the filtering algorithm, which then uses the score of the arguments to rank the vehicles, or just at the end of the filtering phase, for the arguments which should not be used by the filtering algorithm (e.g. "this car is the best of the selection", such arguments have no vehicle

	Id	Characteristic	Score	Topic
1	“Price below budget”	price	+0.8	ECONOMIC
2	“Chosen color”	color	+0.2	APPEARENCE
3	“Easy resell”	manual gearbox	+0.3	RESELL
4	“Easy resell”	popularity	+0.3	RESELL
5	“Powerful for category”	horsepower	+0.1	POWER
6	“Missing option”	option sunroof	-0.2	OPTIONS
7	“Bonus option”	option alloy wheels	+0.2	OPTIONS
8	“Bonus option”	option reversing radar	+0.2	OPTIONS
9	“Bonus option”	integrated GPS	+0.2	OPTIONS
10	“Guaranty”	guaranty	+0.1	DEFAULT
11	“Availability”	availability	+0.1	DEFAULT
12	“Best of selection”	none	+0.1	DEFAULT

FIGURE 9.3: Example input of the NLG module of the car selling application. An argument can be reused several times on different vehicle characteristics, in which case the arguments can be aggregated together to be expressed through a single argument.

characteristic associated with them). For each car selected during the filtering phase, the systems calls the NLG module with the set of arguments computed for this car as argument. Figure 9.3 shows a complete example of an input of the NLG module.

Internally, the NLG module is decomposed into three submodules: initialization, document planning and microplanning.

9.3.1 Initialization

The initialization phase creates the constants and mapping necessary for document planning using the input information.

Each argument is represented by a constant dynamically created in the signature $\Sigma_{rhetorics}$. Additionally, constants for the rhetorical relationships are created using the method described in Section 7.4, and the method described in the same section for semantic aggregation is used in order to aggregate arguments when possible, in order to build a richer text[7]. The image by $\mathscr{L}_{rhe\text{-}log}$ of the constants associated with the arguments are defined statically, for each type of argument (i.e. each argument identifier) and the aggregation of several arguments are mapped to the conjunction of the images of their components. In details, the constants of the signature $\Sigma_{rhetorics}$ and the lexicon $\mathscr{L}_{rhe\text{-}log}$ are completed using the following procedure:

- First, new arguments are created, which represent the aggregation of several arguments (see Section 7.4.2.2). Two arguments can be aggregated when they have

[7]For the sake of conciseness, these methods are not reproduced here. The NLG module follows closely the procedures described in the mentioned section.

the same identifier. For instance, the two arguments "Easy resell" of Figure 9.3, are aggregated into a new argument.

- Then rhetorical relationships are created. The application uses one type of rhetorical relationship, the *contrast* relationship[8], which has two nucleus. A contrast relation is created for every pair of arguments which have the same topic but where one of the arguments has a positive score and the other a negative score. The purpose of the contrast relation is to balance negative arguments with positive ones. The notion of topic ensures that the arguments in the contrast are not on subjects too different from each other, in order to avoid weird transitions (for instance balancing an argument on the price with an argument on the color of the vehicle). The topic associated with each argument has been defined by the developer of the application, based on domain knowledge.

- For each argument and rhetorical relationship, one or several constants are created in the signature $\Sigma_{rhetorics}$, following the procedure described in Section 7.4. Each constant is mapped to a λ-term in the signature Σ_{logics}. This mapping is defined statically for each possible argument in the application (i.e. for each argument identifier symbol) and for each kind of rhetorical relationship (here only the contrast relation). So in our case, all constants representing the contrast relationship are mapped to the same λ-term in the signature Σ_{logics}, and several constants representing arguments with the same identifier may also be mapped to the same λ-term in the signature Σ_{logics} (for instance, all constants representing arguments with the identifier "Bonus option" in Figure 9.3).

- The constants are divided into buckets, where all constants in a bucket are either connected to another one in the bucket through a rhetorical relationship (whose constant is also in the bucket) or have the same identifier as another argument in the bucket. The separation into buckets ensures that a spanning rhetorical tree can be built for each separate bucket, as it is generally not possible to build a spanning rhetorical tree for all the input arguments.

- Finally each bucket is used in turn as the configuration of the document planner in order to generate the text for this bucket.

This procedure creates a lot of constants for the input of Figure 9.3. For the sake of concision, the following only shows the constants of the signature $\Sigma_{rhetorics}$ generated

[8]The decision to use this unique rhetorical relation has been made after an analysis of a small corpus of texts written by humans for the application. The fact that only one useful rhetorical relation could be found is somewhat problematic for the evaluation of the ACG framework, but the fact that the application needs semantic aggregation makes it complex enough that it is still useful for the purpose of evaluating the implemented framework.

from the arguments 6, 7, 8 and 9[9]:

$A_6 : \tau[sp = \{A_6\}, p = \{A_6\}]$

$A_7 : \tau[sp = \{A_7\}, p = \{A_7\}]$

$A_8 : \tau[sp = \{A_8\}, p = \{A_8\}]$

$A_9 : \tau[sp = \{A_9\}, p = \{A_9\}]$

$A_{7,8} : \tau[sp = \{A_7, A_8\}, p = \{A_7, A_8\}]$

$A_{7,9} : \tau[sp = \{A_7, A_9\}, p = \{A_7, A_9\}]$

$A_{8,9} : \tau[sp = \{A_8, A_9\}, p = \{A_8, A_9\}]$

$A_{7,8,9} : \tau[sp = \{A_7, A_8, A_9\}, p = \{A_7, A_8, A_9\}]$

$Contrast_1 : \tau[sp = x, p = \{A_6\}] \rightarrow \tau[sp = y, p = \{A_7\}] \rightarrow \tau[sp = x \cup y, p = \{A_6, A_7\}]$

$Contrast_2 : \tau[sp = x, p = \{A_6\}] \rightarrow \tau[sp = y, p = \{A_8\}] \rightarrow \tau[sp = x \cup y, p = \{A_6, A_8\}]$

$Contrast_3 : \tau[sp = x, p = \{A_6\}] \rightarrow \tau[sp = y, p = \{A_9\}] \rightarrow \tau[sp = x \cup y, p = \{A_6, A_9\}]$

$Contrast_4 : \tau[sp = x, p = \{A_6\}] \rightarrow \tau[sp = y, p = \{A_7, A_8\}] \rightarrow \tau[sp = x \cup y, p = \{A_6, A_7, A_8\}]$

$Contrast_5 : \tau[sp = x, p = \{A_6\}] \rightarrow \tau[sp = y, p = \{A_7, A_9\}] \rightarrow \tau[sp = x \cup y, p = \{A_6, A_7, A_9\}]$

$Contrast_6 : \tau[sp = x, p = \{A_6\}] \rightarrow \tau[sp = y, p = \{A_8, A_9\}] \rightarrow \tau[sp = x \cup y, p = \{A_6, A_8, A_9\}]$

$Contrast_7 : \tau[sp = x, p = \{A_6\}] \rightarrow \tau[sp = y, p = \{A_7, A_8, A_9\}] \rightarrow \tau[sp = x \cup y, p = \{A_6, A_7, A_8, A_9\}]$

9.3.2 Document planning and microplanning

The document planning and microplanning phases are done by calling the function `generate` of either the lexicon $\mathscr{L}_{doc}^{en}$ or $\mathscr{L}_{doc}^{fr}$, depending on whether the output language is English or French respectively. The origin target parameter of the function `generate` is $\Sigma_{rhetorics}$, which is used to generate document plans, and the target signature parameter is $\Sigma_{strings}$. For instance, the constants shown above can only be combined to give a single valid rhetorical tree in the signature $\Sigma_{rhetorics}$:

$$Constrast_7(\{A_6\}, \{A_7, A_8, A_9\}) : \tau[sp = \{A_6, A_7, A_8, A_9\}, prom = \{A_6, A_7, A_8, A_9\}]$$

Since the current implementation doesn't support features, this rhetorical tree is actually represented as a set of rhetorical trees, representing the possible permutations of the leaves. Each rhetorical tree is mapped to a λ-term built on the signature Σ_{logics}, one of

[9] constants built from the argument x are noted A_x. Constant representing the aggregation of two arguments x and y are noted $A_{x,y}$ For concision, the feature *prom* has been shortened here to p. In order to avoid unnecessary memory consumption, constants for the aggregation of several arguments (for instance $A_{7,8}$) are created only when they may be used in a rhetorical relationship. The rest of the procedure follows exactly the one described in Chapter 7.

At 27 499€, which is more than 2 500€ below your budget, the price of this Peugeot 508 is unbeatable. It is gray, your favourite color. It has a manual gearbox and it is popular among our clients, which is best if you want to resell. This car has a powerful engine (160 hp) for a sedan, which suits well a vigorous driving. The sunroof is missing, but it has a built-in GPS, a reversing radar and alloy wheels. It has a 24 months guaranty and is available immediately. Of all our vehicles, this offer is the best suited to your needs

À 27499€, soit plus de 2500€ en dessous de votre budget, le prix de cette Peugeot 508 est imbattable. Elle est grise, votre préférence. Elle a une boîte manuelle et elle jouit d'une bonne popularité parmi nos autres clients, ce qui est meilleur en cas de revente. Elle est dotée d'une puissante motorisation (160 ch) pour une berline, qui conviendra bien à une conduite énergique. Elle n'a pas le toit ouvrant demandé, cependant elle a un GPS intégré, un radar de recul ainsi que des jantes alliage. Elle est garanti 24 mois et est disponible immédiatement. De tous nos véhicules, cette offre est la plus adaptée à vos besoins.

FIGURE 9.4: Examples of output texts for the car selling application in English and French.

which is:

$$CONTRAST\ (BE\ SUNROOF\ MISSING)$$
$$(HAS\ CAR\ (AND\ ALLOY\_WHEELS\ (AND\ GPS\ REVERSING\_RADAR))) : t$$

The other possible λ-terms built on the signature Σ_{logics} for this example are only permutations of the constants $ALLOY\_WHEELS$, GPS and $REVERSING\_RADAR$. The concepts used to build the logical sentences follow the guidelines presented in Section 8.3 and are mapped to derivation trees in the same way than described in the same section. Figure 9.4 shows an example output text generated by processing the input of Figure 9.3 through the document planner and microplanner. Since the miroplanning phase was not very important for this evaluation, the textualisation of the different concepts has been left to the minimum, and each output sentence has only one possible variant.

9.3.3 Results

The results of this evaluation are somewhat mixed. The fact that only one rhetorical relationship is used to build the document plans makes the evaluation a bit weak. On the other hand, it shows how a real world application actually uses the framework, and the fact that semantic aggregation is needed makes it more interesting. There were no particular problem encountered while building the linguistic resources for this evaluation (other than the ones already encountered in the first evaluation), which confirms that the framework is suited for building realistic applications. The performances are also rather good, with the document planning phase taking around one hundred milliseconds

in the same conditions than the first evaluation. Overall, the objective to show that the implemented framework can be competitive with the existing technology both in terms of execution speed and expressivity is achieved. However, more tests need to be performed in order to explore the usage of the ACG framework for document planning, and the evaluation performed here have only scratched the surface on that matter.

9.4 Perspectives

There are many ways in which the prototype NLG framework presented in this thesis can be improved. These improvement can be classified in two broad categories: evolutions of the ACG kernel, written in C++, and new linguistic resources.

9.4.1 Evolution of the ACG kernel

On the short term, several optimizations can be made:

- The Datalog prover uses a backward chaining algorithm. Given the fact that we often need all possible derivations for a particular query, it would make more sense to use a forward chaining algorithm, which would be more efficient.
- The unification algorithm has a naive implementation and needs to be rewritten.
- Other optimizations such as answer set programming need to be studied.

Aside from the possible optimizations, several major modifications to the framework are possible. First, implementing types with features seems to be a very important step in order to make the framework more usable in practice. The basic idea here would be to modify the type system so as to include features, and upgrade the Datalog prover so as to perform constraint logic programming, thereby including the different kinds of constraints introduced by the type system. Another interesting evolution, would be the ability to specify a probabilistic distribution over the elements defined by a grammar. This could be done for instance by adding weights to the constants, and use a ranking algorithm in order to select the most probable output texts. The weights can also be used in heuristic search, thereby improving the speed of the generation when the space of possible output texts is huge. Finally, a more technical, while probably very important evolution would be to have a better integration of the ACG module with the rest of the application, by allowing automatic conversion of objects into λ-term based on their type in the NLG framework language.

9.4.2 New linguistic resources

The semantic level described in Section 8.3.1 has many limitations, and it would probably be a good idea to provide a more theoretically sound resource for the syntax-semantic interface at some point. A research which has only been mentioned in this thesis, but which is very relevant is the G-TAG theory (Danlos, 2000), which describes the syntax-semantic interface and is specialized for text generation. A big advantage of this theory is that it has already been adapted to ACG (Danlos et al., 2014, Maskharashvili, 2016). The reason for not using a full fledged theory for the syntax-semantic interface in this thesis has to do with the ease of use requirement and the wish to provide access to the semantic level to end users, so some experimentations are still to be done in order to compare the cost-benefit ratio of this approach (though it seems pretty clear that it will be needed at some point).

On the side of the creation of new linguistic resources, the most obvious evolution is to add syntactic grammars for different languages, and in particular the languages already supported by the existing technology: French, Spanish, German and Japanese. Associated with these syntactic grammars, the creation of a dictionary of words for each language seems also to be an interesting path. This kind of dictionary can be created automatically using rich lexicons such as Lefff (Sagot, 2010). Going even further up in the levels of abstraction, the creation of resources for the semantic level should be considered. At this level, the only solution is to specialize the libraries to a particular domain of application (e.g. reporting, business intelligence, sales, etc.).

Another interesting path for the creation of new linguistic resources is to go down in the levels of abstraction in order to cover the realisation module. First the syntactic grammars can be augmented with features in order to cover morphological phenomenon. Second, resources can be created for generating strings from different data types. In practice, the textualisation of specific data types, such as integers, dates, floating numbers, etc., represents an important part of the complexity of the realisation module. Having linguistic resources dedicated to them would unify the text generation process and probably simplify the maintenance of the realisation module as well. Finally, new linguistic resources can be created or the existing ones modified in order to cope for the referring expression generation module. Since referring expression generation usually involves long range dependencies, it is not very clear how the linguistic resources should be adapted in order to include this module efficiently. Research also need to be done in order test different means of representation for the document planning module and to explore how the content selection module could be represented using the ACG framework.

For the long term, we may ask ourselves how ACG could be combined with machine learning approaches. Building linguistic resources for a symbolic artificial intelligence is indispensable given the current knowledge and the requirements imposed on the technology, but it can only take us so far. At some point, machine learning seems unavoidable in order to scale to multiple domains and to get closer to human level performances. A first step could be to build linguistic resources directly from corpora, or to adapt them to specific domains automatically from a small corpus. It is interesting to note that the recent advances in machine learning use a representation made of "theory neutral" or generic levels of abstraction (e.g. layers in deep neural nets), which are not so far from what we get by abstracting levels of abstractions out of symbolic computation system. Bridging the gap between symbolic systems and numerical ones is probably one of the important challenge of the AI field in the next decade.

9.5 Conclusion

The initial goal of the thesis was to extend the Yseop technology in order to include both microplanning and document planning techniques, while satisfying a number of constraints:

- The solution needs to be integrated into an existing NLG framework.
- It must satisfy industrial standards of software quality, and in particular in terms of efficiency and ease of use (other important properties include expressivity, reusability and maintainability).
- A particular attention should be paid to the management of external resources, such as linguistic resources.

Chapters 2, 3 and 4 were dedicated to the state of the art in NLG. Many different techniques and architectures exist, both for document planning and microplanning, however there are not so many systems whose goal is to provide a framework for the creation of NLG systems. Most of the existing NLG systems follow what we could call the standard NLG architecture, described in (Reiter and Dale, 2000) (another notable concurrent attempt being the RAGS project Mellish et al., 2004). While being certainly suitable for the project at hand, the standard architecture suffers from several drawbacks (mainly fuzziness, limited reusability and a modularization which introduces different kinds of low level data structures). Chapter 5 introduced a promising alternative found in the formalism of abstract categorial grammars, to deal with some drawbacks of the standard architecture. One of the main advantages of ACG over the standard architecture

is its ability to represent different formalisms with a unique low level data type, thereby unifying the different aspects of the generation process. Another important aspect is its theoretical efficiency and its ability to manipulate existing linguistic resources.

In the second part, I detailed the realization of the idea of using ACG as the kernel of a NLG framework. Chapter 6 introduced the definitions for ACG and the representation of TAG in ACG. The Chapter 7 was dedicated to bridging the gap between the ACG formalism and the different aspects of NLG in the context of a NLG framework, and in particular document planning. This analysis constitutes one of the contributions of this thesis, and has shown how ACG can be used in the context of a NLG framework (especially how the distinction between static and dynamic definitions impact ACG), in particular for performing document planning: I have shown how schemas could be represented, and how to perform RST based document structuring using ACG. In Chapter 8, I presented in details the implementation of an ACG based NLG framework in the existing Yseop technology. I also presented the linguistic resources developed for this project. The description of the implementation is the second contribution of this thesis, and gives insights in how the ACG formalism meets the constraints of industrial software. Finally, in Chapter 9, I presented an evaluation of the implementation against standard software quality properties. The main results can be summarized as follows:

- The ACG formalism has a good modular architecture "by design", which results in a good expressivity, reusability and maintainability of the system. This was one of the main reasons for choosing ACG in the first place.
- The current implementation is complex to use, but it allows to build realistic applications in an efficient way. The idea to build layers of abstraction over the ACG framework in order to allow end users to use it effortlessly is still pending, and the resources allowed to this work did not permit to go further in that direction. This constitutes a limitation in this work, as the ease of use requirement is not met. This however was expected, and the result, though mixed, is still positive as the framework meets its purpose in a satisfying manner, and has proven to be a good basis for further research.
- The framework is faster than the existing technology on standard microplanning tasks. This result was not expected and is very positive, though it is to be taken with caution, as there are still a lot of uncertainty about the overall efficiency of the framework (the framework still need to be optimized, and benchmark tests to be performed).

Overall, the conclusion of this thesis is very positive. The initial goals have been (mostly) met and the risk taken by taking a non-standard approach to practical NLG seems to

have paid off, sometimes even more than what was expected. The results are sufficiently promising to justify to put even more efforts into it, and prompt future work in this vein.

Bibliography

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

Gabor Angeli, Percy Liang, and Dan Klein. A simple domain-independent probabilistic approach to generation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 502–512. Association for Computational Linguistics, 2010.

Douglas E. Appelt. Planning English referring expressions. *Artificial intelligence*, 26(1): 1–33, 1985.

Henk P. Barendregt and Erik Barendsen. Introduction to lambda calculus. *Nieuw archief voor wisenkunde*, 4(2):337–372, 1984.

John A. Bateman. Enabling technology for multilingual natural language generation: the kpml development environment. *Natural Language Engineering*, 3(01):15–55, 1997.

Dave Beckett and Brian McBride. RDF/XML syntax specification (revised). *W3C recommendation*, 10, 2004.

Marcel Bollmann. Adapting SimpleNLG to german. In *Proceedings of the 13th European Workshop on Natural Language Generation*, pages 133–138. Association for Computational Linguistics, 2011.

Kalina Bontcheva. Generating tailored textual summaries from ontologies. In *The Semantic Web: Research and Applications*, pages 531–545. Springer, 2005.

Nadjet Bouayad-Agha, Gerard Casamayor, and Leo Wanner. Content selection from an ontology-based knowledge base for the generation of football summaries. In *Proceedings of the 13th European Workshop on Natural Language Generation*, pages 72–81. Association for Computational Linguistics, 2011.

Nadjet Bouayad-Agha, Gerard Casamayor, Simon Mille, Marco Rospocher, Horacio Saggion, Luciano Serafini, and Leo Wanner. From ontology to nl: Generation of multilingual user-oriented environmental reports. In *Natural Language Processing and Information Systems*, pages 216–221. Springer, 2012.

Thomas Bouttaz, Edoardo Pignotti, Chris Mellish, and Peter Edwards. A policy-based approach to context dependent natural language generation. In *Proceedings of the 13th European Workshop on Natural Language Generation*, pages 151–157. Association for Computational Linguistics, 2011.

Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

Charles Callaway. Multilingual revision. In *EACL 2003*, page 15, 2003.

Charles B. Callaway and James C. Lester. Dynamically improving explanations: A revision-based approach to explanation generation. In *IJCAI (2)*, pages 952–958. Citeseer, 1997.

Marie-Hélène Candito. *Organisation modulaire et paramétrable de grammaires électroniques lexicalisées*. PhD thesis, Université Paris 7, 1999.

Giuseppe Carenini and Johanna D. Moore. Generating and evaluating evaluative arguments. *Artificial Intelligence*, 170(11):925–952, 2006.

William Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

Jose Coch. Overview of Alethgen. In *Demonstrations and Posters of the Eighth International Natural Language Generation Workshop (INLG'96)*, pages 25–28, 1996.

Benoit Crabbé, Denys Duchier, Claire Gardent, Joseph Le Roux, and Yannick Parmentier. Xmg: extensible metagrammar. *Computational Linguistics*, 39(3):591–629, 2013.

Robert Dale. Cooking up referring expressions. In *Proceedings of the 27th annual meeting on Association for Computational Linguistics*, pages 68–75. Association for Computational Linguistics, 1989.

Robert Dale and Ehud Reiter. Computational interpretations of the Gricean maxims in the generation of referring expressions. *Cognitive science*, 19(2):233–263, 1995.

Hercules Dalianis. Aggregation in natural language generation. *Computational Intelligence*, 15(4):384–414, 1999.

Laurence Danlos. *The linguistic basis of text generation.* Cambridge University Press, 1987.

Laurence Danlos. G-TAG: A lexicalized formalism for text generation inspired by tree adjoining grammar. *Tree Adjoining Grammars: Formalisms, Linguistic Analysis, and Processing. CSLI Publications*, 2000.

Laurence Danlos. D-STAG: a formalism for discourse analysis based on SDRT and using synchronous TAG. In *Formal Grammar*, pages 64–84. Springer, 2011.

Laurence Danlos and Fiammetta Namer. Morphology and cross dependencies in the synthesis of personal pronouns in romance languages. In *Proceedings of the 12th conference on Computational linguistics-Volume 1*, pages 139–141. Association for Computational Linguistics, 1988.

Laurence Danlos, Bertrand Gaiffe, and Laurent Roussarie. Document structuring à la sdrt. In *Proceedings of the Eigth European Workshop on Natural Language Generation*, pages 11–20, Toulouse, France, 2001. CNRS, Institut de Recherche en Informatique de Toulouse and Université des Sciences Sociales.

Laurence Danlos, Frédéric Meunier, and Vanessa Combet. EasyText: an operational NLG system. In *Proceedings of the 13th European Workshop on Natural Language Generation*, pages 139–144. Association for Computational Linguistics, 2011.

Laurence Danlos, Aleksandre Maskharashvili, and Sylvain Pogodalla. An ACG analysis of the G-TAG generation process. In *INLG 2014-8th International Natural Language Generation Conference*, pages 35–44. Association for Computational Linguistics, 2014.

Dana Dannélls. Improving information access to cultural content through discourse strategies. In *Proceedings of the eleventh in a series of international scientific Conferences on Advances in Artificial Intelligence held bi-annually by the Italian Association for Artificial Intelligence (AI* IA). Reggio Emilia, Italy*, 2009.

George B. Dantzig. *Linear programming and extensions.* Princeton university press, 1998.

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

Philippe De Groote. Tree-adjoining grammars as abstract categorial grammars. In *TAG+ 6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks*, pages 145–150, 2002.

Philippe De Groote. Abstract categorial parsing as linear logic programming. In *Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015)*, pages 15–25, Chicago, USA, July 2015. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W15-2302.

Rodrigo De Oliveira and Somayajulu Sripada. Adapting SimpleNLG for brazilian portuguese realisation. *INLG 2014*, page 93, 2014.

Seniz Demir, Sandra Carberry, and Kathleen F. McCoy. A discourse-aware graph-based content-selection framework. In *Proceedings of the 6th International Natural Language Generation Conference*, pages 17–25. Association for Computational Linguistics, 2010.

Lorie Den Os. Génération automatique de textes d'analyse financière avec une grammaire FB-LTAG. Master's thesis, Université Paris VII, 2015.

Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. Hyperedge replacement, graph grammars. *Handbook of Graph Grammars*, 1:95–162, 1997.

Michael Elhadad. FUF: The universal unifier user manual version 5.2. Columbia University. *Computer Science Department, June*, 1993.

Michael Elhadad and Jacques Robin. An overview of surge: A reusable comprehensive syntactic realization component. Technical report, Technical Report 96-03, Ben Gurion University, Dept. of Computer Science, Beer Sheva, Israel, 1996.

Michael Elhadad, Jacques Robin, and Kathleen McKeown. Floating constraints in lexical choice. *Computational Linguistics*, 23(2):195–239, 1997.

Jerome Feder. Plex languages. *Information Sciences*, 3(3):225–241, 1971.

Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Dimitrios Galanis and Ion Androutsopoulos. Generating multilingual descriptions from linguistically annotated OWL ontologies: the NaturalOWL system. In *Proceedings of the Eleventh European Workshop on Natural Language Generation*, pages 143–146. Association for Computational Linguistics, 2007.

Claire Gardent. Generating minimal definite descriptions. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 96–103. Association for Computational Linguistics, 2002.

Claire Gardent and Eric Kow. A symbolic approach to near-deterministic surface realisation using tree adjoining grammar. In *45th Annual Meeting of the Association for Computational Linguistics-ACL 2007*, pages 328–335. Association for Computational Linguistics, 2007.

Albert Gatt and Ehud Reiter. Simplenlg: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation*, pages 90–93. Association for Computational Linguistics, 2009.

H Paul Grice, Peter Cole, Jerry Morgan, et al. Logic and conversation. *1975*, pages 41–58, 1975.

Barbara J. Grosz, Scott Weinstein, and Aravind K. Joshi. Centering: A framework for modeling the local coherence of discourse. *Computational linguistics*, 21(2):203–225, 1995.

Karin Harbusch and Gerard Kempen. Generating clausal coordinate ellipsis multilingually: A uniform approach based on postediting. In *Proceedings of the 12th European Workshop on Natural Language Generation*, pages 138–145. Association for Computational Linguistics, 2009.

Eduard H. Hovy. Aggregation in natural language generation. In *In the Proceedings of the Fourth European Workshop on Natural Language Generation*. Citeseer, 1993.

Eduard H. Hovy and Elisabeth Maier. Parsimonious or profligate: how many and which discourse structure relations? Technical report, DTIC Document, 1992.

James Hunter, Yvonne Freer, Albert Gatt, Ehud Reiter, Somayajulu Sripada, and Cindy Sykes. Automatic generation of natural language nursing shift summaries in neonatal intensive care: Bt-nurse. *Artificial intelligence in medicine*, 56(3):157–172, 2012.

Kentaro Inui, Takenobu Tokunaga, and Hozumi Tanaka. Text revision: A model and its implementation. In *Aspects of automated natural language generation*, pages 215–230. Springer, 1992.

Srinivasan Janarthanam and Oliver Lemon. Adaptive generation in dialogue systems using dynamic user modeling. *Computational Linguistics*, 40(4):883–920, 2014.

Aravind K. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? 1985.

Sylvain Kahane. The meaning-text theory. *Dependency and Valency. An International Handbook of Contemporary Research*, 1:546–570, 2003.

Makoto Kanazawa. Parsing and generation as datalog queries. In *ACL*, volume 7, pages 176–183, 2007.

Makoto Kanazawa. Parsing and generation as datalog query evaluation. *To appear*, 2011.

Martin Kay. Functional grammar. In *Annual Meeting of the Berkeley Linguistics Society*, volume 5, pages 142–158, 1979.

Martin Kay. Chart generation. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 200–204. Association for Computational Linguistics, 1996.

Ioannis Konstas and Mirella Lapata. Unsupervised concept-to-text generation with hypergraphs. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 752–761. Association for Computational Linguistics, 2012.

Ioannis Konstas and Mirella Lapata. A global model for concept-to-text generation. *Journal of Artificial Intelligence Research*, 48(1):305–346, 2013a.

Ioannis Konstas and Mirella Lapata. Inducing document plans for concept-to-text generation. In *EMNLP*, pages 1503–1514, 2013b.

Gerasimos Lampouras and Ion Androutsopoulos. Using integer linear programming for content selection, lexicalization, and aggregation to produce compact texts from OWL ontologies. In *14th European Workshop on Nat. Lang. Generation, 51st Annual Meeting of ACL*, pages 51–60. Citeseer, 2013.

Irene Langkilde-Geary and Kevin Knight. Halogen statistical sentence generator. *Proceedings of the ACL-02 Demonstrations Session, Philadelphia*, 2002.

Alex Lascarides and Nicholas Asher. Segmented discourse representation theory: Dynamic semantics with discourse structure. In *Computing meaning*, pages 87–124. Springer, 2007.

Alon Lavie and Abhaya Agarwal. METEOR: An automatic metric for MT evaluation with high levels of correlation with human judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation*, pages 228–231. Association for Computational Linguistics, 2007.

Benoit Lavoie and Owen Rambow. A fast and portable realizer for text generation systems. In *Proceedings of the fifth conference on Applied natural language processing*, pages 265–268. Association for Computational Linguistics, 1997.

Percy Liang, Michael I. Jordan, and Dan Klein. Learning semantic correspondences with less supervision. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*, pages 91–99. Association for Computational Linguistics, 2009.

Saad Mahamood and Ehud Reiter. Generating affective natural language for parents of neonatal infants. In *Proceedings of the 13th European Workshop on Natural Language Generation*, pages 12–21. Association for Computational Linguistics, 2011.

Saad Mahamood, William Bradshaw, Ehud Reiter, and NLG Arria. Generating annotated graphs using the nlg pipeline architecture. *INLG 2014*, page 123, 2014.

François Mairesse and Marilyn A. Walker. Controlling user perceptions of linguistic style: Trainable generation of personality traits. *Computational Linguistics*, 37(3): 455–488, 2011.

William C. Mann and Sandra A. Thompson. Rhetorical structure theory: Toward a functional theory of text organization. *Text-Interdisciplinary Journal for the Study of Discourse*, 8(3):243–281, 1988.

Tomasz Marciniak and Michael Strube. Discrete optimization as an alternative to sequential processing in nlg. In *Proceedings of the 10th European Workshop on Natural Language Generation (ENLG 2005), Aberdeen, UK*, 2005.

Daniel Marcu. Building up rhetorical structure trees. In *The Proceedings of the Thirteenth National Conference on Artificial Intelligence*, volume 2, pages 1069–1074, Portland, Oregon, August 1996. American Association for Artificial Intelligence.

Daniel Marcu. From local to global coherence: A bottom-up approach to text planning. In *The Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 629–635, Providence, Rhode Island, July 1997. American Association for Artificial Intelligence.

Aleksandre Maskharashvili. *Discourse Modelling with Abstract Categorial Grammars*. PhD thesis, Université de Lorraine, 2016.

David D. McDonald. Natural language generation: complexities and techniques. Technical report, DTIC Document, 1986.

Deborah L. McGuinness, Frank Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

Kathleen R. McKeown. Discourse strategies for generating natural-language text. *Artificial Intelligence*, (27):1–41, 1985.

Chris Mellish and Jeff Z. Pan. Natural language directed inference from ontologies. *Artificial Intelligence*, 172(10):1285–1315, 2008.

Chris Mellish, Mike Reape, Donia Scott, Lynne Cahill, Roger Evans, and Daniel Paiva. A reference architecture for generation systems. *Natural Language Engineering*, 10 (3-4):227–260, 2004.

Frédéric Meunier. *Implantation du formalisme G-TAG.* PhD thesis, Université Paris VII, 1997.

Bertrand Meyer. *Object-oriented software construction.* Prentice hall New York, 1988.

Johanna D. Moore and Cécile L. Paris. Planning text for advisory dialogues: Capturing intentional and rhetorical information. *Computational Linguistics*, (19):651–695, 1993.

Jean-François Nogier and Michael Zock. Lexical choice as pattern matching. *Knowledge-Based Systems*, 5(3):200–212, 1992.

Mick O'Donnell, Chris Mellish, Jon Oberlander, and Alistair Knott. Ilex: an architecture for a dynamic hypertext generation system. *Natural Language Engineering*, 7(03): 225–250, 2001.

Daniel S. Paiva and Roger Evans. Empirically-based control of natural language generation. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 58–65. Association for Computational Linguistics, 2005.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

Cécile Paris, Nathalie Colineau, Andrew Lampert, and Keith Vander Linden. Discourse planning for information composition and delivery: A reusable platform. *Natural Language Engineering*, 16(01):61–98, 2010.

Theodosios Pavlidis. Linear and context-free graph grammars. *Journal of the ACM (JACM)*, 19(1):11–22, 1972.

Paul Piwek and Kees Van Deemter. Constraint-based natural language generation: A survey. Technical report, Technical Report 2006/03, Computing Department, The Open University, 2006.

Detlef Plump. Term graph rewriting. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, 2:3–61, 1999.

Sylvain Pogodalla. Ambiguïté de portée et approche fonctionnelle des TAG. In *Traitement Automatique des Langues Naturelles-TALN 2007*, pages 325–334, 2007.

Sylvain Pogodalla. Advances in abstract categorial grammars: Language theory and linguistic modeling. esslli 2009 lecture notes, part ii. 2009.

Alain Polguère. La théorie sens-texte. *Université de Montréal URL: http://olst. ling. umontreal. ca/pdf/PolgIntroTST. pdf*, 1998.

François Portet, Ehud Reiter, Albert Gatt, Jim Hunter, Somayajulu Sripada, Yvonne Freer, and Cindy Sykes. Automatic generation of textual summaries from neonatal intensive care data. *Artificial Intelligence*, 173(7-8):789–816, 2009.

Owen Rambow. Domain communication knowledge. In *Fifth International Workshop on Natural Language Generation*, pages 87–94, 1990.

Ehud Reiter. Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible? In *Proceedings of the Seventh International Workshop on Natural Language Generation*, pages 163–170. Association for Computational Linguistics, 1994.

Ehud Reiter. An architecture for data-to-text systems. In *Proceedings of the Eleventh European Workshop on Natural Language Generation*, pages 97–104. Association for Computational Linguistics, 2007.

Ehud Reiter. Method and apparatus for configurable microplanning, September 15 2015. US Patent 9,135,244.

Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

Ehud Reiter, Roma Robertson, and Liesl M. Osman. Lessons from a failure: Generating tailored smoking cessation letters. *Artificial Intelligence*, 144(1-2):41–58, 2003.

Jacques Robin. *Revision-Based Generation of Natural Language Summaries Providing Historical Background*. PhD thesis, Columbia University, 1994.

Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

Charlotte Roze. *Vers une algèbre des relations de discours*. PhD thesis, Université Paris Diderot - Paris 7, 2013.

Benoît Sagot. The LEFFF, a freely available and large-coverage morphological and syntactic lexicon for French. In *7th international conference on Language Resources and Evaluation (LREC 2010)*, 2010.

Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with Gecode. http://www.gecode.org/doc/4.2.0/MPG.pdf, 2010. (Online, accessed 19 July 2016).

James Shaw. Clause aggregation using linguistic knowledge. Proceedings of the Joint 17th International Conference on Computational Linguistics 36th Annual Meeting of the Association for Computational Linguistics (COLING-ACL'98), 1998.

Pierre-Luc Vaudry and Guy Lapalme. Adapting SimpleNLG for bilingual English-French realisation. In *Proceedings of the 14th European Workshop on Natural Language Generation*, pages 183–187, 2013.

Leo Wanner, Bernd Bohnet, Nadjet Bouayad-Agha, François Lareau, and Daniel Nicklaß. MARQUIS: Generation of user-tailored multilingual air quality bulletins. *Applied Artificial Intelligence*, 24(10):914–952, 2010.

Michael White, Rajakrishnan Rajkumar, and Scott Martin. Towards broad coverage surface realization with CCG. In *Proc. of the Workshop on Using Corpora for NLG: Language Generation and Machine Translation (UCNLG+ MT)*, 2007.

XTAG Research Group. A lexicalized tree adjoining grammar for English. Technical Report IRCS-01-03, IRCS, University of Pennsylvania, 2001.

Michael R. Young and Johanna D. Moore. Dpocl: A principled approach to discourse planning. In *Proceedings of the Seventh International Workshop on Natural Language Generation*, pages 13–20. Association for Computational Linguistics, 1994.

Jin Yu, Ehud Reiter, Jim Hunter, and Somayajulu G. Sripada. A new architecture for summarising time series data. *ITRI-04-01 INLG04 Posters: Extended*, page 49, 2004.