Vrije Universiteit Brussel
Faculteit Wetenschappen en
Bio-ingenieurswetenschappen
Departement Computerwetenschappen

Generalisation and Specialisation Operators for Computational Construction Grammar and their Application in Evolutionary Linguistics Research

Proefschrift voorgelegd tot het behalen van de graad van doctor in de wetenschappen aan de Vrije Universiteit Brussel te verdedigen door

Paul VAN EECKE

Promotoren:
Prof. dr. em. Luc Steels
Prof. dr. Katrien Beuls

Brussel, 4 oktober 2018

2018 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic & Scientific Publishers nv)
Keizerslaan 34
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@aspeditions.be
www.aspeditions.be

ISBN 978 90 5718 825 1
NUR 980
Legal deposit D/2018/11.161/111

Jury Members

Chairman	Prof. dr. Coen De Roover *Vrije Universiteit Brussel, Belgium*
Secretary	Prof. dr. Geraint Wiggins *Vrije Universiteit Brussel, Belgium*
External members	Prof. dr. Luc De Raedt *KU Leuven, Belgium*
	Prof. dr. Laura Kallmeyer *Heinrich-Heine-Universität Düsseldorf, Germany*
	Prof. dr. Freek Van de Velde *KU Leuven, Belgium*
Promotors	Prof. dr. em. Luc Steels *Vrije Universiteit Brussel, Belgium*
	Prof. dr. Katrien Beuls *Vrije Universiteit Brussel, Belgium*
Internal member	Prof. dr. Bart Jansen *Vrije Universiteit Brussel, Belgium*

Abstract

The natural languages that underlie human communication are remarkably expressive, robust and well-adapted to the communicative needs of their users. However, the question of how these languages have emerged and through which mechanisms they continue to evolve remains heavily debated. A common methodology for studying this question is to simulate the emergence and evolution of language using agent-based models. In these models, a population of autonomous agents, which are either physical robots or software entities, participates in a series of communicative interactions, known as *language games*. Each game is played by two agents in the population, one being the speaker and the other being the hearer. The game involves a scripted, communicative task, which either succeeds or fails. At the end of the game, the speaker provides feedback to the hearer, so that learning can take place. The goal of the models is to determine the exact mechanisms that need to be present in the individual agents, so that a communication system with human language-like properties can emerge and evolve.

While agent-based models have within the language game paradigm most extensively been used to study concept learning and vocabulary formation, they have more recently also been successfully applied to experiments on the emergence and evolution of grammar. In these models, the agents need to be equipped with a computational grammar formalism that supports robust and flexible language processing, including mechanisms for inventing and adopting grammatical structures. This dissertation presents three major contributions to the field of research that studies the modelling of the emergence and evolution of grammar.

The first contribution consists in the implementation of a new, higher-level notation for Fluid Construction Grammar (FCG). FCG is an advanced computational grammar formalism that is often used in evolutionary linguistics experiments. The new notation represents grammatical structures in a more intuitive way and abstracts away from low-level implementation details. This facilitates the use of FCG in language evolution

experiments and the new notation has indeed already become FCG's standard notation.

The second contribution introduces powerful mechanisms for generalising and specialising grammatical constructions. The impasse that arises when agents are faced with utterances that they cannot process can often be overcome by adapting constraints that block the application of existing grammatical constructions. Previous experiments relied on ad hoc ways to detect and adapt these constraints. Here, I extend FCG with three general mechanisms: (i) an anti-unification based operator that finds the blocking constraints and their least general generalisations, (ii) a hierarchical type system that can capture these generalisations is a fine-grained way, and (iii) a pro-unification operator that imposes additional constraints on a construction, specialising it to specific cases.

The third contribution consists in a case study that demonstrates how the representations and mechanisms introduced above can be incorporated in an actual agent-based experiment. The experiment that I present here studies how early syntactic structures can emerge and evolve in a population of agents. In particular, it models how shared word order patterns can come into place and reduce the referential ambiguity of the language. The experiment makes use of the type hierarchy system to capture the association strength between words and slots in the word order patterns, and relies on the anti-unification operator to expand the coverage of existing patterns to novel words. The experiment shows that a coherent and efficient word order system rapidly emerges in a population of agents that are equipped with these general, local mechanisms.

Samenvatting

De natuurlijke talen die aan de basis liggen van menselijke communicatie zijn opvallend expressief, robuust en aangepast aan de communicatieve noden van de taalgebruikers. De vraag hoe deze talen ontstaan zijn en zich voortdurend verder ontwikkelen, blijft echter onderwerp van hevige discussie. Een moderne methode om deze vraag te bestuderen, bestaat erin om het ontstaan en de evolutie van taal te simuleren door middel van agentgebaseerde modellen. Tijdens deze simulaties neemt een populatie van autonome agenten - dit kunnen fysieke robots of softwareagenten zijn - deel aan een reeks communicatieve interacties, die *taalspelen* genoemd worden. Elk taalspel wordt gespeeld door twee agenten in de populatie, de ene in de rol van spreker en de andere in de rol van hoorder. Het spel verloopt volgens een vast script en omvat een communicatieve taak die beide agenten tot een goed einde proberen te brengen. Na elk spel geeft de spreker feedback aan de hoorder over de uitkomst van de taak, zodat leren mogelijk wordt. Het doel van de modellen is om de exacte mechanismes te bepalen die in de individuele agenten aanwezig moeten zijn, opdat een communicatiesysteem dat bepaalde kenmerken van menselijke taal vertoont, kan onstaan.

Hoewel het taalspelparadigma in het verleden het vaakst werd gebruikt om het onstaan van concepten en vocabularia te bestuderen, wordt het sinds kort ook met succes toegepast om het ontstaan en de evolutie van grammatica te simuleren. Om deze simulaties te kunnen uitvoeren, moeten de agenten uitgerust worden met een computationeel grammaticaformalisme dat snelle, robuuste en flexibele taalverwerking ondersteunt en dat mechanismes voor het uitvinden en het leren van grammaticale structuren bevat. Dit proefschrift levert drie belangrijke bijdragen aan het onderzoeksveld dat het modelleren van de evolutie van grammatica bestudeert.

De eerste bijdrage betreft de implementatie van een nieuwe notatie voor Fluid Construction Grammar (FCG). FCG is een geavanceerd computationeel grammaticaformalisme dat vaak in evolutionaire taalkunde-experimenten gebruikt wordt. De nieuwe notatie stelt grammaticale regels en structuren op een intuïtievere manier voor, en zorgt ervoor

dat bepaalde technische aspecten automatisch achter de schermen behandeld worden. Dit maakt het gebruik van FCG in evolutie-experimenten heel wat makkelijker, wat ervoor gezorgd heeft dat deze nieuwe notatie ondertussen de standaardnotatie voor FCG geworden is.

De tweede bijdrage introduceert krachtige mechanismes om grammaticale constructies te generaliseren en te specialiseren. De impasse die ontstaat wanneer agenten geconfronteerd worden met talige uitingen die ze niet kunnen verwerken, valt vaak op te lossen door de condities die de toepassing van een bestaande constructie tegenhouden, aan te passen. In vorige experimenten werden ad-hoc methodes gebruikt voor het detecteren en aanpassen van deze condities. Hier wordt FCG uitgebreid met drie algemene mechanismes: (i) een anti-unificatiegebaseerde operator die de blokkerende condities en hun meest specifieke generalisaties bepaalt, (ii) een hiërarchisch typesysteem dat toelaat deze generalisaties op een fijnmazige manier te vatten, en (iii) een pro-unificatieoperator die bijkomende condities toevoegt aan een constructie, zodat deze enkel in specifieke gevallen kan toepassen.

De derde bijdrage bestaat uit een gevalstudie die toont hoe de representaties en mechanismes die in de eerste twee bijdragen geïntroduceerd worden in een concreet agentgebaseerd experiment toegepast kunnen worden. De gevalstudie onderzoekt hoe primitieve syntactische structuren kunnen ontstaan in een populatie van agenten. Meer specifiek modelleert het experiment hoe gedeelde woordvolgordepatronen kunnen ontstaan en de referentiële ambiguïteit van de taal reduceren. Het experiment maakt gebruik van het hiërarchisch typesysteem om de associatiesterkte tussen woorden en kavels in de woordvolgordepatronen te vatten, en gebruikt de anti-unificatieoperator om de dekking van bestaande patronen naar nieuwe woorden uit te breiden. Het experiment toont aan dat een coherent en efficiënt woordvolgordesysteem snel kan ontstaan in een populatie van agenten die uitgerust zijn met deze algemene, lokale mechanismes.

Acknowledgements

The scientific adventure that has now resulted in the completion of my doctoral dissertation started around 5 years ago, at a summer school in Cortona, Italy. There, the organiser of the school, Luc Steels, introduced me to his ambitious research program and to his vibrant team of multitalented researchers. I immediately knew that I wanted to join his team and a few months later, I was thrilled to be able to start my PhD under his supervision. During my PhD, I could pursue my research interests and ideas in a lively and stimulating research environment, with plenty of opportunities to meet excellent researchers from many fields and to actively participate in the scientific process. Thank you, Luc, for all the opportunities you have created, and for all the trust you have placed in me. It is due to your vision and inspiration that I have become the researcher who I am today.

My deepest gratitude also goes to Katrien Beuls, who became the second promotor of my dissertation. Thank you, Katrien, for all the inspiring discussions about the fundamentals and future of our research, and for the highly productive days of pair programming. Thank you also for your precious advice and unconditional support, in particular during the last phase of my PhD; it means a lot to me. I have always enjoyed working with you, and I am happy that we will be able to continue to do so for the foreseeable future.

I am also indebted to the other members of my PhD jury: prof. dr. Coen De Roover, prof. dr. Geraint Wiggins, prof. dr. Laura Kallmeyer, prof. dr. Luc De Raedt, prof. dr. Freek Van De Velde and prof. dr. Bart Jansen. Thank you for all the work you have put into reading and assessing this dissertation. Your inspiring questions, comments and suggestions have not only improved the quality of this document, but will also have a positive impact on my future research.

I would also like to thank Remi van Trijp for teaching me the ropes of the scientific process, for introducing me into the computational construction grammar and evolutionary linguistics communities, and for providing valuable and detailed comments on

every chapter of this dissertation. Thank you, Remi, I appreciate it a lot.

A large part of my PhD research has been conducted at Sony Computer Science Laboratories (CSL) Paris. I am grateful to all my CSL colleagues for their support, whether scientific, administrative or technical. I would in particular like to thank François Pachet, former director of CSL; Sophie Boucher, the lab manager; Nicolas Duval and Pratik Bhoir, the system administrators, and Peter Hanappe, one of the finest researchers I have ever met. At CSL, I was fortunate enough to share an office for more than 3 years with Miquel Cornudella, an excellent Lisp teacher and a true friend. I would also like to thank Timotée Neullas, Emmanuel Deruty, David Colliaux, Stéphane Rivaux and Michael Anslow, for the many interesting and entertaining discussions we had.

My gratitude also goes to all my VUB colleagues, in particular to Jens Nevens and Roxana Rădulescu for their valuable feedback on my work, to Frederik Himpe for solving all my technical problems, and to Lara Mennes and Brigitte Beyens for their administrative support. I would also like to thank Emília Garcia Casademont for sharing her insights on the emergence of phrase structure, Raquel Fernández and Robert van Rooij for hosting me during a two-month research visit at the Institute for Logic, Language and Computation (ILLC) and Michael Rovatsos for professionally coordinating the ESSENCE project, on which I was employed.

I am especially grateful to all the present and former members of the Centre for Computational Linguistics, in particular to Frank Van Eynde, Vincent Vandeghinste, Liesbeth Augstinus and Jonas Doumen. You are the reason that I studied computational linguistics and artificial intelligence, and that I was well prepared to carry out this research project.

I would also like to thank my family for their continuous support throughout my education and scientific career. Thank you for providing me with the freedom and opportunities to pursue my dreams and for supporting all the decisions that I made. My gratitude also goes to my T&L, 'harde kern' and other friends, you know who you are.

Finally, I would like to express my warmest appreciation to the person who stood by me at every point during this journey, who encouraged me to seize every scientific opportunity that presented itself, who weighed in with crucial advice at a number of key choice points, and without whom the end result would not have been the same. Thank you, Lynn, for everything.

The research reported in this dissertation has been financially supported by the Marie Curie Initial Training Network ESSENCE (grant agreement no. 607062), with additional funding from Sony Computer Science Laboratories Paris.

Contents

Chapter 1

Introduction

1.1 Introduction

The natural languages that underlie human communication are remarkably expressive, robust and well-adapted to the communicative needs of their users. While the origins of these languages have fascinated the scientific community for many years, the exact mechanisms through which they have emerged and continue to evolve remain heavily debated. A modern methodology for investigating these mechanisms consists in simulating the emergence and evolution of language using agent-based models. In these models, a population of autonomous, artificial agents, which are either physical robots or software entities, participates in a series of situated, communicative interactions,

called *language games*. Each game is played by two agents in the population, one agent in the role of speaker and the other in the role of hearer. The game involves a scripted communicative task, for example establishing joint attention to a particular object in the scene. In order to fulfil the task, the speaker will need to convey information to the hearer using language, and the hearer will need to perform an action. At the end of the game, the speaker provides feedback to the hearer about the outcome of the task, so that learning can take place.

The language game paradigm allows studying the exact mechanisms that need to be present in the individual agents, so that a communication system that exhibits human language-like properties can emerge and evolve. The results of the experiments do not only contribute to the linguistic debate on the origins of language, but also provide insight into how artificial systems, in which large populations of agents develop their own robust, flexible and adaptive communication system, can be built.

Previous studies have mainly focused on the application of the language game paradigm to concept learning and vocabulary formation, which has led to a good understanding of the mechanisms that are involved (Steels, 2011b). In a smaller number of studies, the same paradigm has also successfully been applied to the emergence and evolution of grammar, but the understanding of the mechanisms that are involved there is still much more limited. A major challenge in setting up these experiments is to equip the agents with a computational formalism that allows for robust and flexible language processing and that includes the appropriate mechanisms for inventing and adopting grammatical structures. This dissertation aims to advance the state of the art in this domain of research by introducing improved representations and more general and powerful invention and adoption mechanisms with which the agents can be endowed, and by presenting a first experiment in which a population of agents employs these representations and mechanisms for developing a shared language that makes use of syntactic patterns for improving its expressiveness and efficiency.

The remainder of this chapter is structured as follows. The first part presents an overview of the objectives and contributions of the dissertation (1.2). The second part discusses the potential impact of these contributions (1.3). Finally, the third part explains in detail the structure of this document (1.4).

1.2 Objectives and Contributions

The primary objective of this dissertation is to improve the representations and mechanisms that are currently used in agent-based models of language evolution, and extend

them with more powerful and general learning operators. This is done with the goal of establishing a general framework that provides powerful building blocks for conducting more advanced experiments on the emergence and evolution of grammar, and for designing intelligent systems in which a large number of agents needs to communicate using a robust, flexible and adaptive language. A secondary objective is to make use this framework to investigate how early syntactic structures can emerge and evolve in a population of artificial agents, and what advantages these structures bring to the language.

These two high-level objectives are tackled by five concrete contributions, which are inspired by three major subfields of artificial intelligence (AI): knowledge representation and reasoning (1.2.1), metacognition (1.2.2) and evolutionary computation (1.2.3).

1.2.1 Knowledge Representation and Reasoning

One of the central goals in AI is to find adequate computational representations that are able to capture information about relevant subdomains of the world in such a way that they can be used by intelligent systems to automatically solve complex tasks. In the case of language games, the complex task that the agents need to solve consists in comprehending and formulating utterances, i.e. mapping between utterances and a representation of their meaning. The knowledge that needs to be represented in order to be able to solve the task is captured in the form of a computational grammar. The first two contributions of this dissertation concern major advances in the representations that are used in Fluid Construction Grammar (FCG), the computational grammar formalism that is most widely used in evolutionary linguistics experiments.

- **A High-Level Notation for FCG.** I present the implementation of a high-level FCG notation. The new notation presents the information that is contained in constructions and transient structures in a clearer and more intuitive way, and is high-level in the sense that it handles low-level processing issues behind the scenes. The conceptual clarity, intuitiveness and high-level nature of the new notation facilitate the design and set-up of language evolution experiments and allows for an easier explanation of the grammars that are learnt.
- **A Type Hierarchy System for FCG Symbols.** All information that is captured in an FCG grammar is local to its constructions. This has important advantages when used in evolutionary experiments, where new constructions, features and categories can emerge at any moment, and existing ones constantly evolve. However, it has the disadvantage that the grammar cannot explicitly capture generalisations and systematic relations among features and categories that are used

in different constructions. In order to accommodate this issue, I introduce a type hierarchy system for FCG symbols. The system allows expressing hierarchical relations among the symbols that are used in the grammar, while preserving the open-endedness, dynamicity and fluidity of the formalism. I show that the fact that these hierarchical relations can be expressed allows capturing fine-grained generalisations that are useful for evolving and learning grammars.

1.2.2 Metacognition

According to the classical action-perception cycle, agents perceive the world, reason about it, and act upon it. This model is well suited to react to situations that were foreseen when the system was built. However, autonomous agents will inevitably encounter unforeseen situations, in which, in order to react appropriately, they do not only need to be able to reason about the world, but also need to be able to reason about their reasoning (Schmill et al., 2008). This problem solving capability plays a central role in AI and is commonly referred to as *metacognition* or *meta-level reasoning*[1]. Metacognition is often implemented by a double action-perception cycle, in which the first cycle perceives, reasons about and acts upon the world, and the second cycle monitors, reasons about and controls the reasoning itself (see e.g. the Soar system (Laird et al., 1987) and the MIDCA system (Cox et al., 2016)). In the case of language processing, the first cycle is concerned with applying the rules of the grammar, and the second cycle is concerned with monitoring this process, reasoning about the grammar rules, and if necessary, adapting the grammar (see e.g. Beuls et al. (2012)). Two contributions of this dissertation concern the metacognitive problem-solving capabilities that are needed for supporting the robust and flexible processing of adaptive languages. These capabilities are concretely implemented in FCG.

- **Integration of a Meta-Level Architecture.** Whether they were part of evolutionary experiments or not, FCG grammars have often been combined with a general meta-level architecture. It required however quite an effort and a detailed knowledge of the FCG processing engine and its codebase, in order to interlace FCG's routine processing layer with the meta-level monitoring and control offered by this architecture. Here, I present a tight integration of this general meta-level architecture in FCG, making it a standard feature that is easy to use with any FCG grammar. It is especially useful in combination with the general and powerful meta-level operators that are presented in the next contribution.
- **Meta-Level Generalisation and Specialisation Operators.** A meta-level ar-

[1]See Cox and Raja (2011) for a good introduction on metacognition.

chitecture is only as powerful as the operators that it has at its disposal. In previous evolutionary linguistics experiments, these operators were always very specific to the problem at hand, acting on predefined features and implementing very specific rules. Here, I present two general meta-level operators that can be applied for solving a much wider range of problems. The first operator concerns an anti-unification-based generalisation algorithm for FCG constructions. This operator allows generalising constructions, relaxing the necessary constraints, so that they can be applied to novel, unforeseen observations. One flavour of the algorithm creates new, generalised constructions, whereas another flavour of the algorithm incorporates the required generalisations into the type hierarchy of the grammar. This operator is particularly useful in evolutionary linguistics experiments for expanding the coverage and applicability of existing constructions. The second operator, called pro-unification, concerns an algorithm that specialises constructions towards observations by incorporating additional, observed constraints. This operator is particularly useful to capture observed agreement or word order regularities.

1.2.3 Evolutionary Computation

Like many techniques in AI, the mechanisms that are used to model the emergence and evolution of language are inspired by processes observed in biological evolution. In particular the concepts of variation, selection, self-organisation and emergent functionality play a central role in this field of research (Steels, 2012b). The last contribution of this dissertation concerns a concrete experiment that combines these concepts with the representations and meta-level operators described above, and shows how a language exhibiting early syntactic structures can emerge and evolve in a population of agents.

- **Agent-Based Experiment on the Origins of Syntax.** I present an agent-based experiment that models how early syntactic structures can emerge and evolve in a population of artificial agents, through processes of variation, selection and self-organisation. The experiment investigates different strategies that the agents can use for structuring their utterances, and compares the emerged languages in terms of expressiveness, efficiency and coherence. Apart from shedding light on the possible mechanistic origins of syntactic structures, the experiment demonstrates that the representations and operators introduced in this dissertation provide novel ways to elegantly model the mechanisms involved in the emergence and evolution of language.

1.3 Potential Impact

I see a potential impact of the contributions presented in this dissertation in at least three domains of research: computational construction grammar (1.3.1), evolutionary linguistics (1.3.2) and intelligent systems (1.3.3).

1.3.1 Computational Construction Grammar

Computational construction grammar is a subdiscipline of linguistics that "aims to operationalise the insights and analyses from construction grammar into concrete processing models" (Van Eecke and Beuls, 2018). These models can be used to validate the completeness and consistency of construction grammar analyses and test their accuracy and coverage on text corpora (Steels, 2017). The contributions presented in this dissertation are already having a concrete impact in this domain. The intuitiveness of the high-level notation that I have implemented facilitates the learning of FCG, the formalisation of linguistic intuitions and hypotheses, and the dialogue with the construction grammar community. This is reflected by the fact that the high-level notation has already become FCG's standard notation, and that it has already been used in multiple publications by different researchers for tackling various linguistic challenges (i.a. Marques and Beuls, 2016; Beuls et al., 2017; van Trijp, 2017; Cornudella Gaya, 2017). I hope that in the future, this will lead to FCG becoming a standard for documenting, validating, testing and exchanging construction grammar analyses, and I believe that the open-endedness of the formalism and the fact that it is largely theory-neutral make FCG well-suited to serve this purpose.

The meta-level operators for generalising and specialising constructions can be used to make grammars more flexible and robust against unexpected input, thus extending their coverage when applied to text corpora. They can also be used to study the mechanisms that are involved in other topics of interest in construction grammar and in which meta-level problem solving is of key importance, such as language acquisition and creativity (Van Eecke and Beuls, 2018).

1.3.2 Evolutionary Linguistics

The potential impact of my dissertation in the field of evolutionary linguistics resides mainly in the methodological innovations that are proposed. The general framework of representations and meta-level operators that is introduced, and demonstrated in the agent-based experiment on the origins of early syntax, will in the future lead to

more advanced experiments on the emergence and evolution of grammar. Building further on my contributions, many interesting experiments come within reach. Among other things, these experiments could (i) use the same mechanisms to extend the current experiment to include hierarchical and recursive structures that further extend the expressiveness of the language, (ii) combine the word order strategy used here with strategies for agreement marking, and (iii) model processes of grammaticalisation, in which lexical words specialise in a specific function and become grammatical markers. The results of these experiments will lead to a better understanding of the mechanisms through which grammatical structures can emerge and evolve and provide evidence for the hypothesis that these structures do not need to be innate but can emerge through communication.

1.3.3 Intelligent Systems

A third area of potential impact concerns applications in which intelligent agents need to communicate, either with humans or with each other. In many applications in which an intelligent agent needs to communicate with a human user, a very precise understanding of what the user says is necessary. This requires mapping the user's utterance to a meaning representation that can be used to reason within the domain of the application. An example of such an application is visual question answering. A human user asks a question about an image, for example taken by the camera of a robot. The system should try to understand the question, reason about the image, and formulate an answer. For example, if the user would ask "Do you see more dogs than cats?" about an image that contains 2 dogs and 3 cats, the system should be able to answer "No, I see more cats than dogs". We are currently designing a visual question answering system for the standard CLEVR dataset (Johnson et al., 2017), and it proves to be the case that the representations and mechanisms described in this dissertation are particularly useful for building such a system. An FCG grammar maps the question of the user to a functional program (e.g. (> (count (filter-object dog input-image) (count (filter-object cat input-image)))) in which the functions 'filter-object' and 'count' are implemented as deep neural networks. The functional program is then executed and an answer to the question is formulated. FCG is especially well-suited for being used in this kind of task, as the meaning representation that it uses can be specifically designed for the problem at hand. A next, exciting step in this project is to use the metacognitive capabilities that are presented in this dissertation to have an agent learn a grammar that maps between the natural language input and the functional programs that serve as a meaning representation. Once this system is in place, an experiment can be set up, in which a population of agents develops its own,

adequate grammar without the need for human input. A different project in which the high-level FCG notation and type hierarchy system have already been applied, explores how machine-readable instructions can be inferred from wikiHow recipes (Cangalovic, 2018).

1.4 Structure of the Dissertation

The remainder of this dissertation is structured as follows:

- **Chapter 2: Background.** This chapter sketches the background of this research project and situates it within the broader contexts of evolutionary and computational linguistics.
- **Chapter 3: Fluid Construction Grammar.** This chapter introduces FCG and focusses in particular on the contributions that were made in the context of this dissertation: the implementation of a high-level notation and the tight integration of a general meta-level architecture for problem solving and learning. This chapter introduces many concepts on which the following chapters build, but can also be read separately as a stand-alone introduction to FCG.
- **Chapter 4: A Type Hierarchy System for FCG Symbols.** This chapter presents an extension to FCG that allows capturing generality-specificity associations between symbols that occur in the constructions of a grammar. In a type-hierarchy, symbols can be declared to be subtypes or supertypes of other symbols and FCG's construction application machinery is adapted to take these (weighted) associations into account. The chapter discusses why such a system is desired, describes how the system is implemented and demonstrates different aspects of its use through examples.
- **Chapter 5: Generalising Constructions using Anti-Unification.** This chapter introduces an anti-unification-based generalisation operator for FCG constructions. It discusses why the operator is needed, describes how different flavours of the operator are implemented, and shows how it can be used to relax constraints that block the application of a construction, or to capture fine-grained generalisations in a type hierarchy.
- **Chapter 6: Specialising Constructions using Pro-Unification.** This chapter introduces an operator that constrains FCG constructions towards concrete observations. It discusses why such an operator is desired, describes its implementation and shows how it can be used to capture observed regularities, for

example in agreement or word order.

- **Chapter 7: Case Study: the Origins of Syntax.** This chapter introduces a case study in which the representations and mechanisms described in the previous chapters are applied in an agent-based experiment on the emergence of early syntactic structures. The case study shows that these representations and mechanisms allow a population of artificial agents to rapidly develop a shared language that makes use of basic syntactic structures for improving its expressiveness, coherence and efficiency.
- **Chapter 8: Conclusion.** This chapter summarises the contributions of this dissertation, reflects on its achievements, and formulates interesting paths to pursue in further research.

This dissertation is accompanied by an interactive web demonstration, which is accessible at `https://www.fcg-net.org/demos/vaneecke-phd`. The web demonstration shows most of the examples discussed in this dissertation in full detail, providing additional information on the specifics of how everything is implemented.

Chapter 2

Background

2.1 Introduction

This chapter sketches the broader research context in which this dissertation is embedded. It first discusses its objectives in relation to the state of the art in evolutionary linguistics (2.2) and then reviews the methods that are used from a computational linguistics angle (2.3). Both sections discuss the wider research context first, and

gradually narrow it down until the specific topics that are addressed in this dissertation are reached.

2.2 Evolutionary Linguistics

Evolutionary linguistics is the field of research that studies the origins of natural languages. This section introduces the main approaches that are taken in this field, the methodologies that are used, the results that have been obtained, and the limits of the current research.

2.2.1 Innate vs. Emergent Language

A long-standing debate in linguistics concerns the nature of linguistic structures. A first view claims that humans are born with a stable, universal grammar, which, as a consequence of its innateness, underlies all human languages (i.a. Chomsky, 1986; Roberts, 2017). According to this view, variations among languages are limited to different parametrisations of this universal grammar (Fanselow, 1993; Gianollo et al., 2008), and learning a language consists thus in finetuning a set of parameters (e.g. Fodor and Sakas, 2017). The alternative view argues that linguistic structures are not innate or a priori present in the human brain, but that language is a dynamic system that emerges through the communicative interactions of interlocutors (Hopper, 1987; Jasperson et al., 1994). The contributions presented in this dissertation are situated within a research program that investigates the latter view. This research program aims to build models that show that linguistic structures can emerge, evolve and propagate through the communicative interactions of members of the language community.

2.2.2 Three Complimentary Perspectives on Language Evolution

The evolution of natural languages can be studied from different points of view, which each have their own goals and methodologies. Steels (2012c, p. 1-2) identifies the following three perspectives:

- The *biological perspective* (i.a. Jablonka and Lamb, 2005; Fitch, 2010; Arbib, 2012; Dediu, 2007; Bickerton and Szathmáry, 2009) focuses on the human biological endowment. This perspective investigates how the cognitive functions that are needed in the brain for processing language are neurobiologically implemented, and how these neurobiological structures and processes are genetically

encoded. It also studies how, where and when the genetic basis underlying the human language capactity has evolved in the history of biological evolution. The methodology that is used to study language evolution from this perspective is the framework of Darwinian, genetic evolution that is used in evolutionary biology.

- The *social perspective* (i.a. Knight et al., 2000; Tomasello, 2003; Dor et al., 2014) focuses on the societal conditions that have influenced the evolution of language. These include (changes in) the size of communities, the social relations, the need to cooperate on complex tasks and the role of trust in establishing a (symbolic) communication system that can be used to deceive. The methodologies that are used to study language evolution from this perspective belong to the domains of anthropology and social science.

- The *cultural perspective* (i.a. Steels, 1999; Smith et al., 2003; Kirby et al., 2008; Christiansen et al., 2009; Steels, 2012c) focuses on how languages evolve over time as a consequence of their use in communication. The evolution of any unit of language can be the topic of investigation, including, but not limited to, sounds, concepts, words, semantic structures, morphological structures, syntactic structures and dialogue structures. The cultural perspective studies how these units can appear in a language, propagate, change (for example through grammaticalisation processes), erode and disappear. The methodologies that are used to study language evolution from this perspective belong to the domain of linguistics, and are inspired by concepts used in evolutionary biology (e.g. variation, selection, self-organisation, emergent functionality) and artificial intelligence (e.g. agent-based modelling, evolutionary computation).

These three perspectives are complementary and are highly dependent on each other. However, the fact that the processes of biological, social and cultural evolution take place at very different timescales makes it possible to study them in isolation. For example, processes of cultural evolution can be studied independently from processes of biological and social evolution, as long as they do not assume cognitive functions or social circumstances that conflict with the insights provided by these other perspectives.

While Steels (2012d, p. 1) points out that a complete theory of language should cover all three perspectives, further progress needs to be made in the subdomains that investigate these perspectives before a unified theory can be presented. This dissertation focuses uniquely on the cultural perspective on language evolution.

2.2.3 Cultural Language Evolution

The field of cultural language evolution studies how cultural transmission can explain the linguistic structures that are observed in natural languages. The research in this field is centred around two complimentary research questions. The first question is concerned with how the learning biases of language learners shape the structure of languages. This research question is studied through the *iterated learning* paradigm. The second question is concerned with how linguistic conventions arise through interaction and coordination, and is studied through the *language game* paradigm.

The Iterated Learning Paradigm

The iterated learning paradigm (i.a. Kirby, 2001; Smith et al., 2003) studies "*the process by which a behaviour arises in one individual through induction on the basis of observations of behaviour in another individual who acquired that behaviour in the same way*" (Kirby et al., 2014, p. 108). The paradigm provides a framework for investigating how language is transmitted across generations of language users, by which the learning mechanisms used by the learners give rise to certain structural features observed in human languages. An important concept in the theory is the *transmission bottleneck*. Learners only have access to a small amount of learning data for learning a complex and open-ended system, which leads to a pressure for systematicity and compositionality in language.

Experiments following this paradigm typically start with a random language, in the sense that its form-meaning mappings are not systematic nor compositional. This language is used 'internally' by a first-generation speaker and a subset of the language is 'externalised' through communication and observed by a second-generation learner. The learner generalises over these observations, reconstructing his own 'internal' language. The learner then becomes the speaker, and 'externalises' a subset of his internal language to a third-generation learner, who generalises over his observations and reconstructs his 'internal' language. This process is repeated over many generations and the experiments show that the language that emerges is systematic and compositional. Note that in these experiments, only the hearer imposes structure on the language, and that achieving communicative success does not play any role.

The iterated learning paradigm has been studied through three different methodologies. The initial experiments were conducted using *agent-based models*. These models focus mainly on the learner's biases involved in the emergence of systematic, compositional languages. These include biases towards generalisation (Kirby, 2001, 2002a), and

one-to-one meaning-signal mappings (Smith, 2002, 2004). The paradigm has also been investigated using mathematical models, which confirm that languages that are subject to iterated learning converge to the learning biases of the learners (Griffiths and Kalish, 2007). Finally, the iterated learning paradigm has also been tested in many experiments with human participants (Kirby et al., 2008). These experiments confirm that the regularisation and systematisation of languages that are the outcome of agent-based and mathematical models can also be replicated in the laboratory, and reveal the learning biases of humans.

Overall, the experiments that have been conducted within the iterated learning paradigm convincingly show that the learning biases of the learners can give rise to certain aspects of linguistic structure, in particular to the emergence of regular and compositional structures. The experiments do however not (have the goal to) account for the sharing, coordination and alignment of linguistic concepts and structures in a population within a single generation.

The Language Game Paradigm

The language paradigm (Steels, 1995, 1997, 2012d) shares the aim of the iterated learning paradigm to explain the origins of language in terms of cultural evolution, but proposes very different selectionist criteria. Instead of focusing on transmission constraints and learning biases, the language game paradigm attributes the emergence and evolution of linguistic structures to the pressures of communicative success, cognitive (processing) effort and social conformity.

Experiments in this paradigm are conducted with populations of agents that are situated in a physical or simulated world. The agents in the population repeatedly play language games. Typically, two agents from the population, a speaker and a hearer, participate in a game and carry out a communicative task. The speaker needs to convey information to the hearer using language. He uses re-entrance to monitor his conceptualisation and formulation process, i.e. he internally comprehends and interprets the utterance that he would formulate and determines whether he would have been able to complete the task based on this information. If necessary, the speaker will add the necessary words, meanings, categories, structures, or any other elements to his grammar (invention). The hearer comprehends and interprets the utterance and performs the task. Then, the speaker gives feedback about the outcome of the task. If the communicative interaction succeeded, the speaker and hearer typically reinforce the linguistic elements that they have used. If it failed, the speaker might punish the elements that he has used, and the hearer uses the feedback to learn. He makes hypotheses about the linguistic elements

that were used by the speaker and adds them to his grammar (adoption). The idea is that if the adequate invention, adoption and reinforcement mechanisms are in place, the population will eventually converge on a shared language.

The language game paradigm has been studied through agent-based models (i.a. the collection of papers presented in Steels, 2012c) and through mathematical models (i.a. Baronchelli et al., 2006; Liu et al., 2009). The experiments, which will be explained in more detail below, convincingly show that through local interactions and communicative pressures, populations of agents can globally coordinate on languages that feature many of the characteristics observed in human languages, such as shared semantic categories, vocabularies and case systems. The framework and case study that are presented in this dissertation are situated within the language game paradigm.

2.2.4 Language Games on the Emergence of Vocabularies

A substantial number of experiments have used the language game paradigm to study how populations of agents can reach conventional agreement on shared vocabularies without central control, central broadcasting or mind-reading. These experiments study variations on the 'naming game' (Steels, 1995), in which the agents in the population develop a language to refer to objects using unique (proper) names. These experiments have been grounded in physical robots (Steels and Loetzsch, 2012), a variety of alignment mechanisms has been studied (Wellens, 2012), scaling laws have been determined (Baronchelli et al., 2006), convergence proofs have been formulated (De Vylder and Tuyls, 2006), and the effects of social network structures (Liu et al., 2009), intrinsic motivation and active learning (Schueller and Oudeyer, 2016) have been investigated . The naming game has been so extensively studied that Steels (2011b, p. 350) concludes that "*the question how a set of conventions can become shared in a distributed population of autonomous individuals through a cultural process has been solved [in this context]*".

2.2.5 Language Games on the Emergence of Concepts

A next series of experiments go beyond vocabularies that consist of proper names, and study how a population of agents can coordinate on conceptual systems. In these experiments, the agents do not only invent, adopt and align a set of words, but also invent, adopt and align a set of semantic categories that serve as meanings to these words. For example, imagine that a speaker needs to draw the attention of a hearer to an object that is the largest object in the scene. The speaker does not have any

concepts or words yet and perceives the size of the objects on a continuous scale. He reasons that the size channel discriminates well this object from the other objects in the scene and that is has thus a high chance to lead to communicative success. He invents a new size category (let's call it `large`, although the agent will probably prefer `size-cat-1`) with the size of the object as initial value. He will now invent a word to express this semantic category and say the word. The hearer will of course not understand it, but after feedback, he will adopt the word and associate it either to a similar semantic category that he already has, or otherwise create a new one. One word might become associated to multiple categories and vice versa. Both words and semantic categories have a score assigned to them, which reflects how often they led to communicative success in the past. After each interaction, the score of the word is updated, the score of the category is updated, and the category itself is shifted towards the observation (e.g. if `large` had a value of 0.6 and the new observation had a size value of 0.7, the value of `large` might become 0.62).

Experiments on the emergence and evolution of concepts have been conducted in many domains, including color categories (Steels et al., 2005; Puglisi et al., 2008; Bleys et al., 2009), action categories (Steels and Spranger, 2008), spatial categories (Spranger, 2012) and semantic roles (van Trijp, 2008). The results of the experiments show that through communicative interactions, populations of agents cannot only coordinate on shared vocabularies, but also on shared conceptual systems that underlie these vocabularies. While the emergence and evolution of conceptual systems might not be a solved problem yet, these experiments have led to a relatively good understanding of the general concept formation and alignment mechanisms that are involved.

2.2.6 Language Games on the Emergence of Grammar

In a third wave of language game experiments, the complexity of the languages that emerge and evolve increases considerably, both from a semantic and a morpho-syntactic point of view. The topic of interest shifts from semantic categories and vocabularies towards semantic relations and grammatical structures. There are two main approaches to the emergence of grammar. The first approach investigates how complex compositional and hierarchical syntactic structures might arise to express complex compositional and hierarchical semantic structures, whereas the second approach studies how morpho-syntactic structures might arise to dampen the referential ambiguity of the language.

Inspired by cognitive and procedural semantics, the first approach especially focusses on the meaning side of the languages. The concepts that emerge and their semantic

representations now include predicates that express relations and that are combined compositionally and hierarchically. The syntactic structures co-evolve with the semantic structures and mirror their composition and hierarchy. Experiments following this approach have been conducted in the domains of color (Bleys and Steels, 2009; Bleys, 2016), spatial relations (Spranger et al., 2010; Spranger, 2016), quantifiers (Pauw and Hilferty, 2012; Pauw et al., 2013) and logical operators (Sierra-Santibáñez, 2014, 2018). These experiments investigate how different strategies for conceptualising relevant aspects of the world compare to each other in terms of (time to reach) communicative success, (time to reach) convergence and cognitive effort.

The second approach studies the morpho-syntactic strategies that languages use to minimize their referential ambiguity. These strategies are beneficial for the language users, as they increase the expressiveness of the language and minimize the cognitive effort that is needed to process utterances in this language. In order to achieve this, natural languages employ two main morpho-syntactic devices, namely markers, for example for case, gender or number, and word order. Both markers and word order indicate which words in an utterance belong together, which reduces the referential ambiguity of the language. Experiments have been conducted on the emergence and evolution of case markers (van Trijp, 2013; Lestrade, 2015a; van Trijp, 2016; Lestrade, 2016), the recruitment and erosion of agreement markers (Beuls and Steels, 2013; Vera, 2018) and the emergence of phrase structure (Steels and Garcia Casademont, 2015a,b; Garcia Casademont and Steels, 2016). These experiments confirm the hypothesis that the emergence of shared grammatical structures reduces the ambiguity of the languages, thus maximizing their expressiveness and minimizing the cognitive effort involved in their processing.

2.2.7 Limits of the Current Research

The overview of experiments presented above shows that the language game paradigm cannot only explain the emergence and self-organisation of vocabularies and conceptual systems, but can also be used to study the emergence of complex semantics and shared grammatical structures. While the emergence of shared vocabularies might be considered a solved problem by now, the research into the emergence and evolution of grammatical structures still leaves many questions unanswered. Important challenges are here to explain how language strategies such as case marking and phrase structure, as well as different conceptualisation strategies, emerge, how certain strategies become important for structuring a language, how they compete with other strategies, and how they might fade away again.

As the complexity of the agent-based models scales up enormously when moving from the emergence and evolution of vocabularies to the emergence and evolution of grammar, the tools that are used to build the models need to become more powerful as well. There are currently two released software packages in use that provide useful building blocks for setting up this kind of experiments.

The first software package is called *MoLe: Modelling Language Evolution* (Lestrade, 2017), and was used in the experiments on the emergence of case reported by Lestrade (2015a,b, 2016). MoLe is distributed as an R package[1] and was especially developed for studying case and argument structure, as reflected by its former name *WDWTW (who does what to whom)*. It includes the necessary building blocks for setting up multi-agent language games in which lexical items can be recruited as grammatical markers. MoLe does not include an advanced semantic processing engine, an elaborate language processing engine, and interfaces to physical robots or rich world models.

The second software package is called *Babel2* (Loetzsch et al., 2008) and was used in all other experiments on grammar evolution described above, except for the ones presented by Sierra-Santibáñez (2014, 2018); Vera (2018). Babel2 groups a number of software tools that can be used to set up a wide range of evolutionary linguistics experiments. It is implemented in Common Lisp and is distributed via its github page[2]. Babel2 includes an experiment framework for implementing multi-agent simulations and a monitoring and visualisation system for tracking and visualising both the details of individual language games and results that are aggregated over a series of games (Loetzsch et al., 2009). For conceptualising complex meanings, and interpreting meanings in relation to the world, Babel2 includes a powerful procedural semantics framework, called incremental recruitment language (IRL) (Spranger et al., 2012b). For mapping from a meaning representation to an utterance (comprehension), and vice versa (formulation), it includes Fluid Construction Grammar (FCG) (Steels, 2011a, 2017), a bidirectional computational grammar framework. Babel2 also includes a robot interface that connects these tools with physical robots, as well as a general meta-level architecture that can be used in the experiments to separate the routine processing abilities of the agents from their problem solving capabilities.

These two software packages have led to interesting experiments on the emergence and evolution of grammatical structures. However, more advanced experiments, in which different language strategies are combined, become tedious to implement using the existing systems and would benefit from more advanced tools to model the representation, invention, adoption, competition and alignment of grammatical structures in a more general way. In this dissertation, I aim to push the state of the art

[1] `https://CRAN.R-project.org/package=MoLE`
[2] `https://github.com/EvolutionaryLinguisticsAssociation/Babel2`

in this domain by introducing improved representations and more general invention, adoption and alignment mechanisms in Babel2's language processing component Fluid Construction Grammar.

2.3 Computational Linguistics

This section discusses the language processing techniques that are used in experiments on the emergence and evolution of grammar from a computational linguistics point of view. It first describes the properties that are required for a language processing component to be used in such experiments (2.3.1) and then discusses a selection of linguistic formalisms, focussing in particular on how they relate to these requirements (2.3.2).

2.3.1 Requirements

In order to be used as the language processing component in evolutionary linguistics experiments, a computational grammar formalism needs to satisfy a number of requirements. The formalism needs to (i) have sufficient expressive power, (ii) support bidirectional, semantic processing, (iii) have an efficient implementation, and (iv) integrate invention, adoption and alignment mechanisms that allow for robust, flexible and open-ended language processing.

Expressive Power When studying the emergence and evolution of natural languages, it is obvious that the expressiveness of the computational grammar formalism that is used needs to be sufficient to model at least the phenomena that are under investigation, and preferably also other phenomena that are observed in natural languages. These phenomena include for example free and fixed word orders, hierarchical and recursive structures, case and agreement marking systems, and long-distance dependencies.

Bidirectional, Semantic Processing Evolutionary linguistics experiments require a language component that is capable of semantic processing. The task of this component is not to accept or generate sentences that are licensed by a grammar, but to use a grammar to map between utterances and a representation of their meaning. It is crucial that the system is bidirectional, in the sense that the same grammar and processing

mechanisms are used for both comprehension (mapping from an utterance to a representation of its meaning) and formulation (mapping from a meaning representation to an utterance). This ensures that the grammatical structures that are learned by an agent in comprehension can immediately be used by this agent in formulation, and that the structures that are invented in formulation can also be comprehended. It is important that the grammar formalism allows for meaning representations that are expressive enough to model the meanings conveyed through natural language utterances.

Efficient Implementation The grammar formalism needs to have an efficient implementation that can be used in experiments in which utterances are formulated and comprehended thousands and thousands of times.

Flexible Language Processing In evolutionary linguistics experiments, each agent in the population gradually builds up his own grammar, based on the outcome of the communicative interactions in which he participates. This means that the grammar of an agent is not a static system, but a dynamic system that undergoes changes every time an utterance is comprehended or formulated. The grammar formalisms that are used need to reflect this dynamic nature of the language. They need to support mechanisms for introducing new linguistic elements into the grammar, for adopting linguistic elements introduced by other language users, and to model the competition between different elements that serve the same function.

2.3.2 Grammar Formalisms

A large variety of grammar formalisms has been developed in the computational linguistics literature. Based on the kind of structure that these formalisms attribute to the utterances of a language, they can be categorised into three groups: phrase structure grammars, dependency grammars, and construction grammars. The following sections briefly discuss these three groups of grammar formalisms with a special focus on the requirements described above.

Phrase Structure Grammars

Phrase structure grammars analyse utterances in terms of hierarchical constituency structures. A constituent is a group of (typically) adjacent words that function as a single unit in an utterance. The smallest constituents are individual words, and larger constituents span over multiple smaller ones. The constituents are organised into a

tree structure that spans over all the words of an utterance. Most phrase structure grammars spring from the generative grammar tradition pioneered by Chomsky (1956, 1957). The main aim of these grammars is to establish a set of rules that describes (recognises and generates) all correct sentences of a natural language, and only those.

Phrase structure grammars have often been computationally implemented using context-free grammars (CFGs). Early on, CFGs have been enhanced with systems for handling transformations that capture relations between sentences, for example the relation between the active and passive voice (Chomsky, 1957). Other formalisms enhance context-free grammars with features, allowing to drastically reduce the number of rewrite rules needed to model phenomena such as subject-verb agreement (Gazdar et al., 1985). CFGs do however not have sufficient expressive power to model all utterances observed in natural languages (Shieber, 1985), which has led to the development of more expressive formalisms.

One class of formalisms aims to extend the expressive power of CFGs in order to be able to model at least some of the non-context-free structures that are observed in natural languages, while retaining the property that they can be parsed in polynomial time. These formalisms include, in increasing order of expressiveness, tree-adjoining grammars (TAGs) (Joshi et al., 1975), linear context-free rewriting systems (LCFRSs) (Vijay-Shanker et al., 1987) and range concatenation grammars (RCGs) (Boullier, 2000). For an elaborate discussion of this class of formalisms, see Kallmeyer (2010). Like CFGs, these formalisms represent utterances as tree structures and aim to model all and only the correct sentences of a language. Apart from the relations that are captured in the tree structures, they do not aim to model the semantics of the sentences that they describe. This is the main reason why (extended) CFGs are usually not used in evolutionary linguistics experiments.

A different approach is taken by categorial grammars, such as combinatory categorial grammar (CCG) (Steedman, 2000). Categorial grammars attribute to each lexical item a syntactic type, which is either primitive (e.g. N and NP for noun and noun phrase) or complex (e.g. S\NP for an intransitive verb, meaning 'takes an NP on the left and returns an S'). Based on these syntactic types, utterances are analysed through deduction. Categorial grammars are well-suited for semantic parsing, as during the deduction process that builds up the phrase structure, lambda expressions capturing the compositional meaning of an utterance can be constructed in parallel. Generally, categorial grammars are used for parsing, rather than for production or generation. Categorial grammars have not often been used in evolutionary linguistics experiments, with the exception of Briscoe (2000) in the context of learning parameter settings for a universal grammar.

A third class of phrase structure grammars that has a very high expressive power groups constraint-based formalisms such as lexical-functional grammar (LFG) (Kaplan and Bresnan, 1982), head-driven phrase structure grammar (HPSG) (Pollard and Sag, 1994) and sign-based construction grammar (SBCG) (Boas and Sag, 2012). These formalisms use the unification of feature structures as a basic mechanism to derive the possible sentences of a language, their phrase structure and their semantics. These formalisms model language as a static constraint system, which limits their usability in experiments in which languages dynamically emerge and evolve.

Finally, Definite Clause Grammars (DCGs) (Pereira and Warren, 1980) are an extension of CFGs that is closely related to languages for logic programming, in particular Prolog. DCGs represent grammar rules as definite clauses, as used in computational logic. DCGs can be straightforwardly compiled into plain Prolog code, and have direct access to the full power of the language. They can be processed efficiently, can compute semantic structures and can be used both for parsing and for producing sentences. They have been used in interesting evolutionary linguistics experiments, including Kirby (2001, 2002a) on the emergence of compositionality, and Sierra-Santibáñez (2014, 2018) on the emergence and evolution of language systems for boolean coordination.

By design, phrase structure grammars attribute a central role to the order of the words in an utterance. This allows the implementation of efficient parsing algorithms, but has two major disadvantages when used in evolutionary linguistics experiments. The first disadvantage is that it is difficult to implement phenomena such as case or agreement marking independently from word order constraints. A second disadvantage is that the rules in phrase structure grammars are local and cannot access information captured in structures located more remotely in the tree structure.

Dependency Grammars

Dependency grammars (Tesnière, 1965) do not analyse utterances in terms of constituents, like phrase structure grammars, but in terms of the dependencies between the words that constitute the utterances. These dependencies are usually grammatical functions such as subject, direct object or modifier. The nodes in the parse trees are the words that occur in the utterances. The main verb is the root of the parse tree and arcs go from higher nodes to their dependents, for example from the main verb to the subject noun, and from the subject noun to its determiner. As the parse trees do not need to capture word order, dependency grammars can well be used for modelling free word order languages. Dependency grammars are most often learnt from annotated corpora and are only used in the parsing direction. The semantics that they model is

limited to the labels on the arcs in the dependency trees. For these reasons, dependency grammars are not used in evolutionary linguistics experiments. An elaborate discussion about dependency parsing is presented by Nivre (2006); Kübler et al. (2009).

Construction Grammars

While phrase structure grammars and dependency grammars analyse sentences in terms of constituency and dependency relations respectively, construction grammars do not choose a primary perspective based on a particular relation, but consider many different perspectives at the same time. The basic tenets of construction grammar, as laid out by Fillmore et al. (1988); Goldberg (1995); Kay and Fillmore (1999); Croft (2001); Goldberg (2006), are the following. First of foremost, construction grammars consider language as a collection of form-meaning pairings, called constructions. Constructions cut through the traditional layers of linguistic analysis, as they can combine phonological, morphological, syntactic, semantic, pragmatic and multi-modal information within a single construction. Construction grammars do not distinguish between a 'lexicon' and a 'grammar', but adhere to a lexicon-grammar continuum. The constructions that make up a grammar range from very concrete, for example mapping between a particular string or phonological form and its meaning, to very abstract, for example in the case of argument structure constructions. Constructions can span over multiple worlds or combine specific words with more abstract phrases. Construction grammars consider language as a dynamic system, of which the constructions and their entrenchment are in constant flux.

There have been multiple efforts to build computational systems for formalising and processing construction grammars. Embodied Construction Grammar (ECG) (Bergen and Chang, 2005; Feldman et al., 2009) aims to analyse language into conceptual schemas that parametrize mental simulations (Bergen and Chang, 2005). The ECG system only implements language comprehension and not production. Template Construction Grammar (TCG) (Barres, 2017) and Dynamic Construction Grammar (DCG) (Dominey et al., 2017) are neuro-computational approaches to construction grammar. DCG focusses on mapping regularities in word order to semantic roles and TCG focusses on the neural dynamics of language-vision interactions. Finally, Fluid Construction Grammar (FCG) (Steels, 2011a, 2017) implements the aforementioned tenets of construction grammar with a special focus on language as an open-ended, dynamic system.

FCG was especially designed to be used in evolutionary linguistics experiments. It performs bidirectional, semantic processing, has a very high expressive power, has an

efficient implementation, and integrates many features for modelling the invention, adoption and alignment of linguistic elements. FCG has been used in many evolutionary linguistics experiments, including experiments on the emergence and evolution of case marking (van Trijp, 2016), agreement marking (Beuls and Steels, 2013) and phrase structure (Garcia Casademont and Steels, 2016). The contributions presented in this dissertation are all integrated into the FCG platform. An elaborate discussion of FCG is presented in chapter 3 of this dissertation.

2.4 Conclusion

In this chapter, I have laid out the broader research context in which this dissertation is embedded. I have first situated the project within the field of evolutionary linguistics, in particular within the subfield that employs the language game paradigm to study language evolution from a cultural perspective. Previous work in this field has mainly focussed on the emergence of vocabularies and concepts, which has led to a good understanding of the mechanisms that are involved. More recently, the same paradigm has also successfully been applied in experiments that study the emergence and evolution of grammar. However, more advanced experiments would benefit from more general tools for representing, inventing, adopting and aligning grammatical structures.

Then, I have discussed the methods that are used in these experiments from a computational linguistics perspective. I have introduced the requirements that such experiments impose on the grammatical formalisms that they use. These requirements included sufficient expressive power, support for bidirectional semantic processing, an efficient implementation, and support for the integration of invention, adoption and alignment mechanisms. I have discussed a number of formalisms with respect to these requirements, with a special focus on those that have been used in previous evolutionary linguistics studies. I have introduced Fluid Construction Grammar as a computational construction grammar platform that was especially designed to fit these requirements and that has been extensively used in previous experiments. All contributions that will be presented in the rest of this dissertation are integrated in the FCG platform.

Chapter 3

Fluid Construction Grammar

3.1 Introduction

The framework that I present in this dissertation is integrated in Fluid Construction Grammar (FCG). While most contributions, in particular the type hierarchy system and meta-level operators, are designed as modular extensions to the existing system, other contributions have led to major improvements to the FCG system itself. The present chapter introduces the basics of FCG, on which the remaining chapters will build. It focuses in particular on my own contributions, including the implementation of a higher-level notation (3.2, 3.4), the tight integration of a meta-layer framework and library of diagnostics and repairs (3.5), and the design and implementation of an interactive web service and API (3.6).

The FCG source code including the framework of cognitive operators and all other contributions described in this dissertation, has been released under an Apache 2.0 open source license, and is available at `https://www.fcg-net.org/download`.

3.2 A High-level Notation for FCG

With the first implementation dating back to around 2002, and certain major components even to the late nineties, FCG has been in constant development over the course of the last 15 years. This has led to major improvements in the representation of constructions, the unificatioaengine, the stability of the overall system, the ease of use, and the visualisations and interfaces. When the current research project started, FCG was a mature technology that had proven its value both in evolutionary linguistics experimentsdag (see e.g. the collection of papers published by Steels (2012b)) and in case studies implementing challenging issues in existing natural languages (see e.g. Beuls (2012); van Trijp (2014)). The code had reached a relatively stable status, with the notation and basic design patterns documented by Steels (2011a) and more advanced computational issues discussed by Steels (2012a). I will refer to this stage in the development of FCG as FCG-2011, while I will refer to the current version simply as FCG.

The evolutionary experiments and case studies on existing natural languages that were set up using FCG-2011 provided new insights into how constructions are best represented. These insights led to a new notation that was initially used as a vehicle for explaining FCG grammars to the community in a clearer and more natural way. The new notation was introduced by Steels (2017), a paper of which the initial version was already drafted mid 2014. In the context of the current research project, I have built a

mapping between this new FCG notation and FCG-2011. The implementation of this mapping confirmed the validity and precision of the new notation, while also making major contributions to the notation itself. The mapping is fully operational and has replaced the FCG-2011 notation in the FCG release. While important parts of the FCG core still use FCG-2011 representations, the grammar designer interfaces with the FCG system using the new notation only. All input, such as construction inventories and their constructions, as well as all output, including visualisations of the construction application process, are presented in the new notation.

The new notation is more abstract and represents the information contained in constructions and transient structures in a more intuitive way. Moreover, it can be thought of as a higher-level formalisation, handling certain low-level instructions behind the scenes. The new notation considerably speeds up grammar development and makes FCG much easier to learn. Since its first release in June 2015, it has been used in multiple grammar evolution experiments (e.g. Cornudella et al. (2016); Garcia Casademont and Steels (2016)), case studies on different existing natural languages (e.g. Marques and Beuls (2016) for Portuguese clitics, Beuls et al. (2017) for Russian motion verbs, Beuls (2017) for Spanish verb morphology, Van Eecke (2017) for Dutch verb phrases), the first implemented broad-coverage construction grammar for English (van Trijp, 2017) and even a project on construction-based planning, plan recognition and plan prediction (Beuls et al., in preparation).

3.3 Language Processing as a Problem Solving Process

FCG implements language processing as a *problem solving process*. Research into problem solving has a long history in the field of artificial intelligence (AI), starting in the fifties with the work of Newell et al. (1957). Over time, solid theoretical foundations were laid out (Nilsson, 1971) and problem solving is still very influential in major subfields of AI (Russell and Norvig, 2009). In the field of computational linguistics, language processing has often been treated as a problem solving process. This is the case for both comprehension (Hobbs et al., 1993; Powers et al., 2003) and formulation (Appelt, 1985; Garoufi and Koller, 2010), as well as for language learning (Zock et al., 1988).

A problem can be defined by the following four components, as laid out by Russell and Norvig (2009: 70) for the 8-queens problem[1].

[1] The 8-queens problem consists in finding a configuration of 8 queens on a chessboard, with no queen being attacked by any other queen. The problem was originally proposed by chess composer Max Bezzel in 1848 and is an iconic example problem in AI handbooks.

- **State Representation** A representation of the state of the problem at a certain point in time. This consists of all relevant information about the problem that is known at that moment, in a data structure that allows efficient processing of the problem at hand. In the case of the 8-queens problem, this is a representation of the chessboard, with any configuration of 0 to 8 queens on the board.

- **Initial State** The state representation before the problem solving process has started. It is for this problem state that a solution needs to found. In the case of the 8-queens problem, this is a state representation with no queens on the board.

- **Operators** A set of actions that can be applied to a state representation. The operators transform a state representation into a different state representation. In the case of the 8-queens problem, only 1 operator is available, namely adding a queen to any empty square on the board.

- **Goal Test** A function that decides whether a problem state qualifies as a solution. In the case of the 8-queens problem, 8 queens should be on the board and none of them can be attacked by any other queen.

The task of the problem solver is to find a sequence of operators (a *pathway*) that transforms the initial state into a *goal state*, i.e. a state representation in which the goal test returns true. Starting from the initial state, the subsequent application of operators gives rise to a tree of problem states, defining the *search space*. Finding a pathway that leads to a solution state can be very difficult, as the search space is often very large. In the case of the 8-queens problem, the search space contains more than 10^{14} states, with only 92 states qualifying as a solution. In order to navigate through the search space in an informed way, problem solvers commonly employ search algorithms that rely on *heuristics* to help decide which pathway to pursue and which operators to consider in the current state.

As presented by Steels and Van Eecke (2018), the standard model of problem solving can be mapped to language comprehension and formulation in a straightforward way. In FCG, the basic components of the model are instantiated for language processing as follows:

- **State Representation** The state representation is called the *transient structure*. It is a feature structure containing a representation of all information that is known at that point in processing about the utterance that is being processed. The transient structure can contain any kind of information, including syntactic, semantic, pragmatic, morphological, prosodic, phonological, phonetic and multi-modal information.

- **Initial State** The *initial transient structure* is a feature structure containing all information that is available before processing has started. It is the result of a pre-processing step called *de-rendering*. In comprehension, the initial transient structure contains the form features from the input, which are typically strings and ordering constraints between these strings, possibly in combination with a representation of gestures and prosodic features. In formulation, the initial transient structure contains a representation of the meaning that needs to be formulated.

- **Operators** The operators that apply to transient structures are called *constructions*. Constructions can be seen as schemata that have *preconditons* and *postconditions*. This representation of operators is common in the field of planning (see e.g. STRIPS[2] , ADL[3], and more recently PDDL[4]). When the preconditions are satisfied by a transient structure, the construction can apply and the postconditions manipulate this transient structure to create a new transient structure. As language processing is bidirectional, the constructions have two sets of preconditions, one set for comprehension (called the *comprehension lock*) and the other for formulation (called the *formulation lock*). The preconditions for comprehension serve as postconditions in formulation and vice versa. Constructions can also contain a set of features that serve as postconditions in both comprehension and formulation (called the *contributing part*).

- **Goal Test** After each construction application, goal tests check whether the resulting transient structure qualifies as a solution. Usually, different goal tests are used for comprehension and formulation. In comprehension, goal tests typically check that no more constructions are applicable and that all meaning predicates are integrated into a single semantic network. In formulation, goal-tests typically check that no more constructions are applicable and that all meaning predicates that were present in the input have been used by constructions.

The FCG engine has the task of finding a pathway of constructions that leads from the initial transient structure to a transient structure that qualifies as a solution. Just like in the 8-queens problem, the problem solver relies on heuristics to navigate through the search space in an efficient manner. Here as well, the heuristics give either an indication of which construction should be applied next, or of which branch in the search tree should be pursued. A schematic representation of the problem solving process is presented in Figure 3.1.

[2]STRIPS: STanford Research Institute Problem Solver (Fikes and Nilsson, 1971)
[3]ADL: Action Description Language (Pednault, 1987)
[4]PDDL: Problem Domain Description Language (Malik Ghallab et al., 1998)

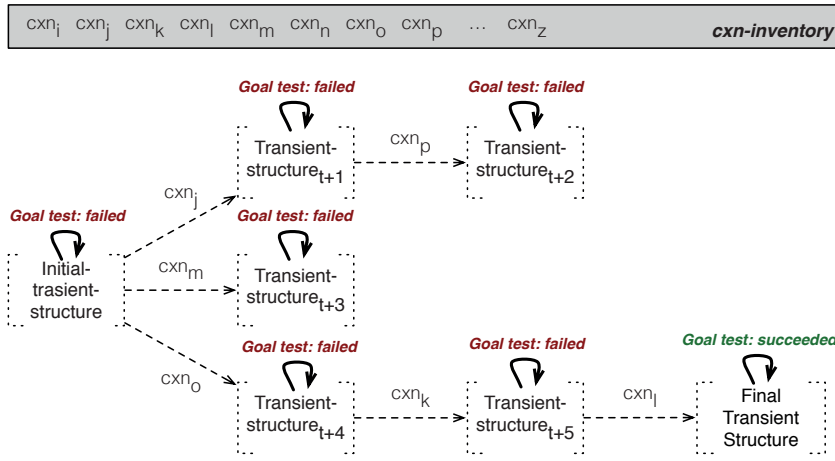


Figure 3.1: A schematic representation of the problem solving process in Fluid Construction Grammar. Constructions apply to create new transient structures. Goal tests check whether the new transient structure qualifies as a solution. If so, the search process finishes, otherwise it continues to apply further constructions, backtracking if necessary. In this figure, a pathway is found, namely $[cxn_o, cxn_k, cxn_l]$.

3.4 The Basic Building Blocks

In this section, I will present the basic components of FCG on a more technical level. As it is not easy to do this in sufficient detail in a written text, I have added interactive visualisations of these components to the web demonstration supporting this dissertation[5]. The best strategy for achieving a deep understanding however, consists in downloading the FCG system and playing around with a small grammar fragment. Technical documentation for the system is available at `https://www.fcg-net.org/tech-doc`.

3.4.1 Transient Structures

Transient structures, the state representations in our search problem, are represented as *feature structures*. Feature structures are widely used in grammar formalisms such as LFG [6], GPSG [7], HPSG [8], SBCG [9] and ECG [10]. In FCG, the feature structures are

[5] `https://www.fcg-net.org/demos/vaneecke-phd`
[6] LFG: Lexical Functional Grammar (Kaplan and Bresnan, 1982)
[7] GPSG: Generalized Phrase Structure Grammar (Gazdar et al., 1985)
[8] HPSG: Head-Drive Phrase Structure Grammar (Pollard and Sag, 1994)
[9] SBCG: Sign-Based Construction Grammar (Boas and Sag, 2012)
[10] ECG: Embodied Construction Grammar (Bergen and Chang, 2005)

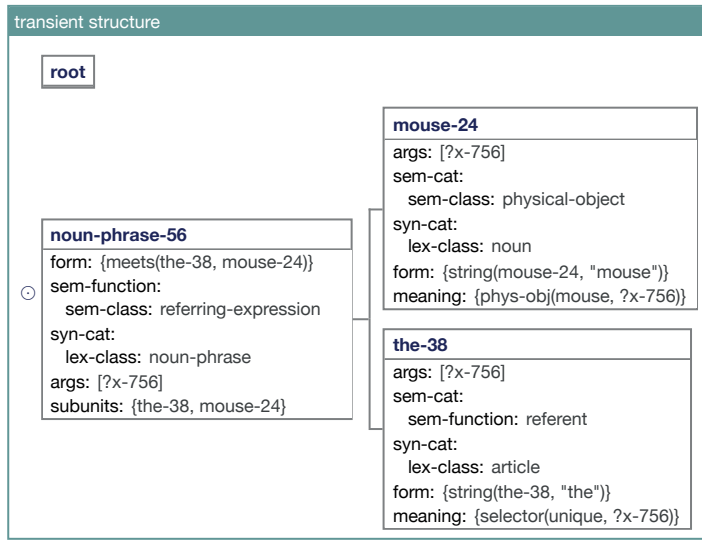


Figure 3.2: Example of a transient structure. It is a feature structure consisting of a set of units. The hierarchy is drawn based on the 'subunits' feature. In the value of that feature, 'the-38' and 'mouse-24' are symbols, not pointers to the units.

not typed, which facilitates a dynamical addition, modification or deletion of features and values in a grammar.

A transient structure consists of a collection of *units*. The units have a *unit name* and a *unit body*, which consists of a set of feature-value pairs. In the transient structure, unit and feature names are constants. The values of features can be either constants, logical variables (symbols that start with a question mark) or feature-value pairs. An example of a transient structure is shown in Figure 3.2.

This transient structure consists of 4 units: 'root', 'mouse-24', 'the-38' and 'noun-phrase-56'. I will skip over the root unit for the moment, and come back to it in the next section. The body of the three other units consists of a set of feature-value pairs, in which the internal ordering is not important. The feature-value pairs have different kinds of values. The 'args' feature for example takes a sequence of symbols as value (indicated by square brackets). The sequence consists here of a single element, '?x-756', which is a variable (as indicated by the question mark). The 'syn-cat' feature takes a feature-value pair as value. The 'lex-cat' feature takes an atomic symbol as value. Finally, the 'form' and 'meaning' features take a set of predicates as value, as indicated by the curly brackets (set) and the predicate notation of their elements (predicate). I will discuss the complete inventory of feature-types and their behaviour in section 3.4.3. For now, I just stress that the set of possible features is completely

open-ended and that their type can be freely chosen by the grammar designer.

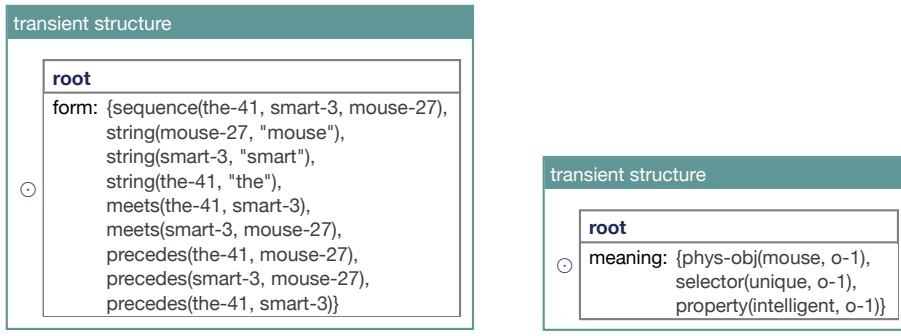
The figure visualises the transient structure in a hierarchical way, namely as a tree. The visualisation is in this case based on the 'subunits' feature, which has here a set of two symbols as its value. These symbols, 'the-38' and 'mouse-24' are equal to the unit names of two other units in the transient structure. This information is used to visualise the hierarchy. It is important to know that the transient structure is implemented as a set of units and that units are never nested. The value of the subunits features are just symbols that correspond to the unit name of other units and should not be thought of at pointers to these units. This means that multiple features can be used to represent different perspectives at the same time, for example constituent structure, dependency structure of information structure. The visualisation can simply be adapted by using a different feature for drawing the hierarchy or any other kind of network of units.

As a historical note, I would like to point out that the transient structure does not consist of a syntactic and a semantic pole any more, as it was the case in FCG-2011. All information in now contained in a single set of units and constructions are able to use and combine any part of this information in both comprehension and formulation.

3.4.2 The Initial Transient Structure

The initial transient structure is the transient structure that is at the root of the search space. It contains all information that is known before the problem solving process has started. The initial transient structure is the result of a *de-rendering* process. This process translates an input into a transient structure that can be used for problem solving. Although different de-render methods are implemented for different tasks, de-rendering for language processing is usually done in a standardised way.

In comprehension, the input consists of a string, such as "the smart mouse" . The input string is first tokenized. Then, a unique identifier symbol is attributed to each token, and a predicate *string(identifier,token)* is created. The unique identifier makes it possible to refer unambiguously to a specific token in the input, even if has the same form as other tokens in the input. This is for example the case when a word occurs more than once in the input utterance. Then, the internal ordering of the tokens is encoded in predicates as well. This is done using three kinds of predicates: *meets(identifier-1,identifier-2)* indicating binary left-right adjacency, *precedes(identifier-1,identifier-2)* indicating a binary precedence relation, and *sequence(identifier-1,identifier-2,..., identifier-n)* capturing the complete sequence. Finally, a feature *form*, which gets as value the set of string, meets, precedes and sequence predicates, is added to a unit called *root* in



(a) Comprehension (b) Formulation

Figure 3.3: Initial transient structure for the utterance "the smart mouse" in comprehension and formulation. All features in the initial transient structure are added to the root unit.

the transient structure. An example of an initial transient structure for the utterance "the smart mouse" is shown in Figure 3.3a.

In formulation, the input consists of a meaning representation. As the meaning representation is typically already specified in the form of a set of predicates, no additional pre-processing steps are needed. A feature *meaning*, with as value the meaning predicates from the input is added to the root unit. The initial transient structure for the meaning representation {*(selector unique o-1) (phys-obj mouse o-1) (property intelligent o-1)*} is shown in Figure 3.3b.

As shown in Figure 3.3, the initial transient structures contain thus a single unit, named *root*, with either a form feature (in comprehension) or a meaning feature (in formulation). The value of this feature is a set of predicates capturing the information in the input. These predicates will later be used by constructions the during the problem solving process.

3.4.3 Constructions

Constructions are the operators in the problem solving process. Based on information present in a transient structure, they can build a new transient structure with more information added. The same constructions and processing mechanisms are used in comprehension and formulation, making FCG a truly bidirectional language processing framework.

Design and Structure

Like transient structures, constructions are implemented as feature structures. But whereas the feature structures representing transient structures are simply sets of units, those representing constructions are more structured. Constructions are data structures consisting of two parts:

- **Conditional Part** The conditional part specifies the preconditions of the construction. It consists of one or more units. The unit names are variables and the unit bodies are split into two parts: the *comprehension lock* and the *formulation lock*. The comprehension locks specify the preconditions of the construction in comprehension and the formulation locks specify the preconditions in formulation. A construction can apply in comprehension when the comprehension locks of the units on its conditional part *match* the transient structure and the construction can apply in formulation when the formulation locks match the transient structure. Matching is a unification process that succeeds if for each unit on the conditional part of a construction, a unit in the transient structure can be found such that the active locks (comprehension locks in comprehension, formulation locks in formulation) unify. The non-active locks are *merged* into the transient structure. Merging is another unification process, in which on top of matching, features that are present in the construction but not in the transient structure are added. In the visualization of a constructions, the conditional part is written on the right-hand side of the arrow. For each conditional unit, the comprehension lock is written above the horizontal line and the formulation lock below. The structure of a construction is schematically sketched in Figure 3.4.

- **Contributing Part** The contributing part of a construction is written on the left-hand side of the arrow. It contains zero or more units, of which the names are variables. When the conditional part of a unit matches a transient structure, the units on the contributing part are merged into the structure. When the contributing part contains features that cause conflicts during merging, the process fails and the construction cannot apply[11]. If merging succeeds, the construction application succeeds.

There are no restrictions on what can be written on the left-hand side or right-hand side of a construction, which makes an FCG grammar unrestricted (type-0 grammar) in the Chomsky hierarchy (Chomsky, 1956).

As a historical note, I add that this layout of constructions allows the elimination of the

[11] In technical terms, this would be called a *second-merge-fail*, whereas a *first-merge-fail* indicates a failure in merging the non-active locks into the transient structure.

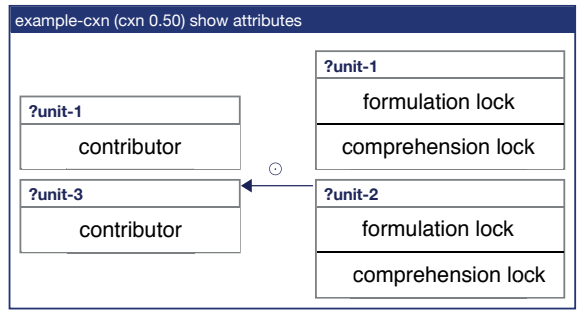


Figure 3.4: A schematic representation of a construction in FCG, with its conditional part on the right and its contributing part on the left. Two units with their comprehension and formulation locks are shown in the conditional part and two units with their contributor are shown in the contributing part. Note that the unit names are variables, and that the unit name of one of the units of the contributing part is bound to the unit name of one of the units of the conditional part.

J-unit notation known from FCG-2011 (De Beule, 2007: 47-76). This is a substantial improvement, as understanding the J-unit notation was a major difficulty for learners of FCG.

Feature Types

In the previous section, I have defined construction application as *matching* the active locks of the construction with the transient structure and *merging* the non-active locks and the contributors into the transient structure. I have not given a precise formalisation or algorithm for the matching and merging operations other than describing them as *unification processes*. In fact, FCG allows for a representation of features that is so expressive that features can trigger the use of a specific matching and merging algorithm. The algorithms are associated to features on grammar level or construction level by the declaration of *feature types* and are reflected in the visualisation of feature-value pairs. Note that feature types in FCG have nothing to do with 'typed feature structures', which are not used in FCG at all.

I will briefly discuss the feature types that are commonly used in FCG, without going into too much detail. An overview of these feature types, their notation and associated algorithms, is shown in Table 3.1. For a formal definition of the underlying unification processes, I refer the reader to Steels and De Beule (2006), De Beule (2012) and Sierra-Santibáñez (2012).

- **Default Feature Type** If no other type is declared for a certain feature, the FCG

engine will assume that it is of this default type. The feature has a value that is either atomic, i.e. a constant or variable, or complex, in which case it consists of one or more feature-value pairs. In matching, the feature name should be present in the transient structure. If the value is atomic, it should unify with the value in the transient structure. If it is complex, the feature-value pairs in the construction should be a subset of the feature-value pairs in the transient structure, with each corresponding pair matching. In merging, the feature should not necessarily be present in the transient structure. If it is already there, the bindings from matching are instantiated. If it is not, the feature and its value are added to the transient structure. The value of a feature of the default type cannot have multiple features with the same feature name.

- **Set** This feature type indicates that the value of a feature is a set. The value is written between curly brackets. In matching, the FCG engine checks whether the elements in the value of the feature in the construction unify with a subset of the elements in the value of the same feature in the transient structure. In merging, the bindings from matching are instantiated and any new elements in the value are added to the transient structure. In sets, the same element can occur in the value multiple times.

- **Sequence** The value of a sequence feature is written between square brackets. In both matching and merging, the order and number of elements in the value is meaningful and every element should match and merge with the corresponding element in the transient structure.

- **Set-of-predicates** This feature type is very similar to the set feature type, but the elements of the value here are predicates. This feature type is most often used with the form and meaning features in FCG. The predicates are written in predicate notation and placed in curly brackets.

- **Sequence-of-predicates** This feature type is very similar to the sequence feature type, but the elements of the value here are predicates. The predicates are written in predicate notation and placed in square brackets.

Another device that can be used in a construction to call a specialised matching and merging algorithm is the negation operator. This operator can be used to indicate that a certain feature-value pair or set element should NOT be present in the transient structure. Negation is indicated by adding the $\neg$ symbol in front of a feature or set element. Negations are only matched and not merged into the transient structure.

As a historical note, I add that the concept of feature types in FCG is quite different from how its functionality was achieved in FCG-2011. In FCG, the feature names are

Table 3.1: The feature types typically used in FCG, with their notation and associated match and merge algorithms. f: feature, v: value, p: predicate, a: argument.

Feat. Type	**Notation**	**Match**	**Merge**
default	f_1 : $f_2 : v_2$ $f_3 : v_3$	subset	instantiation / extension
set	$f_1 : \{v_1, v_2, ..., v_n\}$	subset	instantiation / extension
sequence	$f_1 : [v_1, v_2, ..., v_n]$	exact	instantiation
set-of-predicates	$f_1 : \{v_1(a_1), v_2(a_2, a_3),$ $..., v_n(a_4, a_5, a_6)\}$	subset	instantiation / extension
sequence-of-predicates	$f_1 : [p_1(a_1), p_2(a_2, a_3),$ $..., p_n(a_4, a_5, a_6)]$	exact	instantiation

associated with a specific unification algorithm on grammar or construction level while in FCG-2011, special operators (e.g. '==', '==0', '==1' and '++') were written inside the feature structure. The special operators triggered a different unification algorithm at that point in processing. The current notation makes the feature structures much clearer and easier to understand. Behind the scenes however, the feature type notation is still compiled into the special operator notation.

Operator

One of the most widely used operators is the # operator (*hash-operator*). By putting a # sign in front of a feature in one of the locks of a construction, that feature is not matched in the corresponding unit in the transient structure, but in the 'root' unit. In the merging phase, it is taken from the 'root' unit and merged into the unit it is specified in. If this unit does not exist yet, a new unit is built. The # operator can be thought of in terms of cutting and pasting a feature from the 'root' to another unit. The # operator is extensively used by morphological and lexical constructions to match on strings and meaning predicates from the input. Grammatical constructions often use this operator to match on word-order predicates.

Let us have a look a construction that employs the # operator, namely the 'mouse-cxn' visualised in Figure 3.5. This construction has one unit on the conditional part, with a form feature in the comprehension lock and a meaning feature in the formulation lock. Both features are preceded by a #. We will now apply the construction to the transient structure shown in Figures 3.3a (for comprehension) and 3.3b (for formulation) above. In comprehension, the construction will match the form feature in the 'root' unit of the transient structure. Matching will succeed and in merging, the construction will build a new unit, called mouse-word-xx. Then, it will move the *string(mouse-word-27,*

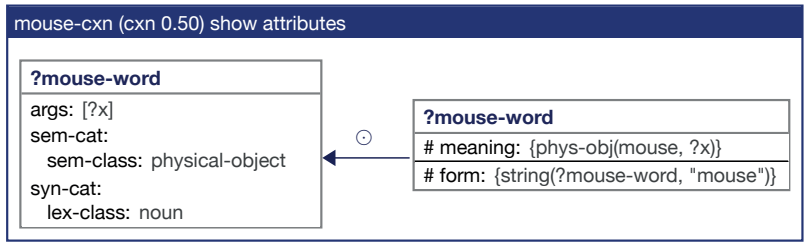


Figure 3.5: An example of a construction that employs the # operator. The feature preceded by a # will be matched in the root unit and merged into the unit they are written in. In this case a new unit 'mouse-word-xx' will be created.

"mouse") predicate to the form feature of this new unit. It will also add the meaning feature from the formulation lock and all features from the contributor. In formulation, the construction will match the meaning feature in the 'root' unit, move it to a new unit and add the form feature from the comprehension lock as well as all features from the contributor.

Another historical note: the # operator replaces FCG-2011's 'tag' mechanism, which would move features from one unit in the transient structure to another unit. The # operator is still mapped to the tag mechanism behind the scenes. Another use of FCG-2011's tag mechanism, which allowed to express both positive and negative constraints on the same feature, is now included in the negation operator.

Expansion (procedural attachment) and Overwriting operators

Two features that were present in FCG-2011 were initially not foreseen in the new FCG notation. But in order to make FCG as expressive as possible, support for these features was added and an appropriate syntax was designed. The two features are an *overwriting* and an *expansion* operator. The overwriting operator ($a => b$) allows matching on one value (a) and, if matching succeeds, to replace this value by another value (b) in merging.

The expansion operator allows for *procedural attachment*, a technique well-known from knowledge representation systems in the field of AI (Bundy and Wallen, 1984). This operator is used to call an arbitrary function with the data on which a feature matches and to merge the data that this function returns into the transient structure. The use of the expansion operator provides the complete power of the LISP programming language to an FCG construction. Moreover, it has access to the whole construction application process, including the construction inventory, previous transient structures

Table 3.2: The special operators used for the overwriting of features and for expansion (procedural attachment). f: feature, v: value.

Operator	**Notation**	**Match**	**Merge**
overwriting	$f_1 : \{v_1 => v_2\}$	match with v_1	replace v_1 by v_2
expansion	third element in feature declaration	procedure	procedure

and applied constructions. The expansion operator is very useful to perform tasks that would be inefficient to do declaratively, for example the numerical computation involved in calculating cosine distances when working with meaning representations based on distributional semantics. An overview of the operators and their notation is presented in Table 3.2.

Footprints

When the active locks of a construction match with a transient structure and the non-active locks and contributors are not in conflict with the transient structure, a construction can apply. This creates the need for control mechanisms preventing that a construction keeps applying an infinite number of times, as the transient structure usually still satisfies the active locks of the construction after its application. One way of preventing the reapplication of a construction is to use the # operator in one of the locks. As the 'hashed' feature is moved from the root into another unit, it cannot be matched in the root any more and the construction cannot apply a second time.

A second mechanism that has a long history in FCG consists in leaving a *footprint* in the transient structure and not applying a construction if its footprint is already there. Concretely, when a construction applies, it merges a feature 'footprints' with as value a set containing the construction name into one of the units in the transient structure that also appears on the conditional part. In the lock of this unit on the conditional part, a negated feature (using the negation operator ¬) 'footprints' with as value a set containing the construction name is added. The first time that the construction applies, the conditional part is satisfied and the footprint is merged in. The second time, the negated feature is in conflict with the footprint and the construction cannot apply. Footprints cannot only be used to prevent the same construction from applying, but can also reduce the search space by blocking large numbers of mutually exclusive constructions.

While in FCG-2011, the use of footprints was entirely the responsibility of the grammar designer, the new FCG notation handles footprints behind the scenes. A negated

footprints feature is added to each lock that contains no # operator of each unit on the conditional part of a construction. To the contributor of the corresponding unit, the footprints feature itself is added. This system ensures that by default, a construction can only apply once using the same units to satisfy its locks. If the grammar designer needs more fine-grained control over the footprints, he can simply turn off the automatic footprints for a single construction or an entire grammar. In that case, he becomes once again responsible for managing footprints himself.

3.4.4 The Construction Inventory

In FCG, a grammar is concretely implemented as a *construction inventory*. The construction inventory contains not only the constructions of a grammar, but also all information that is required to use the grammar in processing. This information consists of the declaration of feature types, a configuration object for processing and a configuration object for visualisation. The feature types declaration holds the feature types that are used throughout the grammar, although they can be overloaded on construction level. The configuration object for processing specifies many different processing options, including the required render and de-render method, search algorithm, maximum search depth and goal tests. The configuration object for visualisation specifies a.o. which features should be used to draw hierarchies and which level of detail should be shown in the visualisation, e.g. for debugging purposes.

Depending on whether a grammar supports meta-layer processing (see Section 3.5) or type hierarchies (see Chapter 4), the construction inventory also holds a list of diagnostics and repairs or the type-hierarchy that needs to be build up or used.

3.4.5 Construction Application and Search

Now that we have seen what transient structures and constructions look like, we will have a closer look at how the FCG engine deals with the construction application and search processes. As explained above, the aim of the FCG engine is to find a pathway, i.e. a sequence of construction applications that leads from the initial transient structure to a transient structure that qualifies as a goal state.

In essence, the construction application process is steered by two mechanisms: the *queue regulator* and the *construction supplier*. The task of the queue regulator is to select the transient structure to which a new construction will be applied. It determines which partial pathway will be explored further. The task of the construction

supplier is to select a construction that will be matched with a certain transient structure. Together, these two mechanisms regulate which part of the search space will be explored.

In its most basic setting, the queue regulator always selects the transient structure that was most recently added to the queue, leading to a depth-first exploration of the search space. The construction supplier in its most basic setting randomly picks a construction from the construction inventory. If the construction can apply, the resulting transient structure is added to the queue and the queue regulator selects this new transient structure. If no construction can apply, the queue regulator will backtrack to the previous node in which not all constructions from the transient structure were tried. Obviously, exploring the search tree in an uninformed way is not feasible for large search spaces. For this reason, FCG provides a number of mechanisms for steering the search process in a way that a solution can be found by exploring only a very small part of the search space.

Construction Sets and Construction Networks

A first way to make the task of the construction supplier easier, is to internally structure the constructions in the construction inventory. There are two common ways for doing this, using *construction sets* or *construction networks*.

When using construction sets, the constructions of a construction inventory are divided into two or more sets. The sets are ordered, with potentially a different ordering for comprehension and formulation. At first, the construction supplier will only supply constructions from the first set. When all these constructions have been supplied, it will continue with constructions from the second set and further sets, until all sets have been exhausted. Construction sets can drastically reduce the search space, as they ensure that certain constructions can only be applied after other constructions have been tried. It is for example common to specify that the set of morphological constructions should be applied first in comprehension and last in formulation. This ensures that at least all morphological information is known before starting to apply grammatical constructions in comprehension, and that all grammatical information is known before starting to realise words as specific morphologic forms in formulation.

Construction networks, also called *priming networks* (Wellens, 2011), organise the constructions in the construction inventory as a network. Each construction has ingoing and outgoing priming links connecting the construction with other constructions. The construction supplier will first supply the constructions that are primed by outgoing links. The links can be trained using a test corpus or in a multi-agent experiment. The

construction networks capture which constructions often apply after a given construction in a successful pathway. By doing this, they manage to extend a partial pathway immediately with a construction that is likely to lead to a solution and therefore reduce the search space.

Hashing Constructions

For any realistic grammar of an existing language, the construction inventory easily consists of over 500.000 constructions. Most of these however, are morphological or lexical constructions. The preconditions of these constructions are very straightforward. In comprehension, they only require a string present in the input and in formulation, they only need a meaning predicate present in the input. Instead of looping through all morphological and lexical constructions and matching them with the transient structure, we organise these constructions in two hash tables. The first hash table has strings as keys and a list of constructions that match on these strings as values. The second hash table has meaning predicates as keys and a list of constructions that match on these meaning predicates as values. The construction supplier can now find a small set of constructions to match for every string or meaning predicate in constant time. When all strings or meaning predicates are moved out of the 'root' unit, the high number of lexical and morphological constructions should not be considered any more by the construction supplier. Using hashed construction sets, a grammar can be extended with as many lexical and morphological constructions as necessary, without causing any performance loss.

Scoring Constructions and Transient Structures

Scores can be assigned to both constructions and transient structures. Scores of constructions can be used by the construction supplier while scores of transient structure can be used by the queue regulator.

In evolutionary experiments, the score of a construction reflects how confident the language user is that this construction will contribute to achieving successful communication. It is dynamically updated based on the outcome of each communicative interaction. The construction supplier will select first constructions with a higher score. These scores are thus mainly used to lead the search process to a *specific* solution state, rather than to lead it faster to *any* solution state.

The score of a transient structure estimates how far away this state is from a solution state. Heuristic search algorithms (such as A*) rely on this score for deciding which

partial pathway in the search tree should be pursued. There has been little research into finding good estimates for the scores of transient structures. It would however be interesting to study the influence of factors such as the number of variable links in the meaning representation.

Grammar Design

Last but not least, the design of the constructions in the grammar itself has also a major influence on the size of the search space. As a general rule, it should be avoided that two constructions with conflicting features can apply to the same transient structure. If certain information is not yet known at one point in processing, it is preferable to underspecify the corresponding features as compared to splitting the search tree into two hypotheses. For example, a morphological construction realising an article as a singular form should only apply after a noun phrase has been build and the number of the noun with which the article needs to agree with is known. Apart from the mechanisms described above, the feature matrices technique presented by van Trijp (2011) has been particularly successful in underspecifying linguistic agreement information for case, number and gender.

3.4.6 Goal Tests and Solutions

Every node in the search tree contains a transient structure. At the moment that the node is created, all goal tests specified in the construction inventory are run on the node. If all goal tests succeed, the node is returned as a solution. If one or more goal tests fail, the search process continues (see Figure 3.1). In comprehension, goal tests typically fail when other constructions can still apply, when the 'root' unit still contains strings of when the meaning predicates in the transient structure are not linked into a single semantic network. In formulation, goal tests typically fail when other constructions can still apply or when the 'root' unit still contains meaning predicates.

When all goal tests succeed and a node is returned as a solution, the final transient structure is *rendered*. In comprehension, rendering is done by extracting all predicates from the meaning feature in every unit of the transient structure. These meaning predicates are then drawn as a semantic network, as shown in Figure 3.6 for the transient structure from Figure 3.3b. In formulation, rendering is done by extracting all predicates from the form feature in every unit of the transient structure. Then, the strings in the *string(identifier,string)* predicates are concatenated taking into account

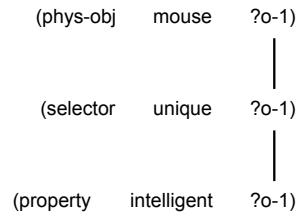


Figure 3.6: In comprehension, a solution is rendered by extracting all meaning predicates from a final transient structure and connecting linked variables with a line. The resulting representation is called a *semantic network*. The network in this figure is rendered from the transient structure shown in 3.2.

the ordering constraints in the other predicates. The output in comprehension is a semantic network and the output in formulation is an utterance.

3.4.7 Meaning Representations

FCG does not impose a particular kind of meaning representation and leaves its design up to the grammar designer. As long as the semantic networks are built up from predicates that share arguments by linking variables, FCG can handle the meaning representation without any additions necessary. Two types of meaning representation are commonly used with FCG. If the aim is to ground the meaning in the sensory-motor systems of physical robots, a procedural semantics called Incremental Recruitment Language (IRL) (Steels, 2000, 2007; Spranger et al., 2012a) is used. If this is not the aim, a minimal form of predicate calculus is used.

The examples in this dissertation use a minimal form of predicate calculus. All predicates are typed and their arguments can only be constants and variables. The predicates are written in the following form: $type(predicate, arg_1, arg_2, ..., arg_n)$. By linking the arguments of different predicates, complex compositional meanings can be expressed.

3.5 Meta-Layer Problem Solving and Learning

In the previous sections, I have presented how FCG implements language processing as a problem solving process. During this process, constructions are consecutively applied to the initial transient structure until a solution is found. This assumes that all linguistic knowledge that is necessary for processing the input utterances or meaning

representations is captured by the constructions of a grammar. There are two problems here. The first problem is that it remains implicit where and how the constructions of a grammar originated. The second problem is that language is constantly evolving and full of innovations, making it an absolute necessity for a grammar to be dynamic, flexible and open-ended. The two problems are closely related and boil down to the question of how a computational construction grammar system can deal with input that is not (yet) covered by a grammar, and how new lexical and grammatical constructions can be learned. In FCG, this is achieved by a meta-layer architecture (Maes and Nardi, 1988), as discussed in the next sections.

3.5.1 Meta-Layer Architecture

FCG's meta-layer architecture divides language processing into two layers: the *routine layer* and the *meta-layer*. The routine layer employs the described machinery for construction-based language processing and is optimized for efficient processing of input that is covered by the grammar. The meta-layer is designed to process input that is not covered by the grammar and to learn new constructions from previously unseen observations. The meta-layer architecture consists of three components: *diagnostics*, *repairs* and *consolidation strategies*.

- **Diagnostics.** Diagnostics are tests that are run after each construction application and inspect the resulting transient structure for any abnormalities or potential problems. If a diagnostic detects an abnormality, it creates a *problem* of a certain type, e.g. a problem of type 'unknown-string' or 'unconnected-semantic-network'. The problem object can also hold further information that is added by the diagnostic, such as the string that was not covered or the construction that has just been applied. Diagnostics have access to the complete construction application process, including the search tree, the previously applied constructions, the transient structures that these constructions have created and the construction inventory. Diagnostics are specified on grammar level and stored in the construction inventory. The problems detected by the diagnostics are local to one branch in the search tree.

- **Repairs.** Repairs are methods implementing problem solving strategies. They specialise on one or more classes of problems that are triggered by diagnostics. When faced with a problem, a repair will try to find a solution in the form of a *fix* object. The type of the fix and the way in which the fix repairs the problem are open-ended. The most common type of fix in FCG is the *fix-cxn*, in which the fix comes in the form of a construction. The application of the fix-cxn to

the transient structure then repairs the problem. The repair methods are run after each construction application, just after the diagnostics are run. They can repair problems that were triggered in the same node of the search tree, but also problems from earlier nodes that were not yet fixed. Just like in the case of diagnostics, repairs have access the to the complete construction application process, the search tree and the construction inventory. Repairs are specified on grammar-level and stored in the construction inventory.

- **Consolidation Strategies.** While diagnostics and repairs diagnose and solve problems at the meta-layer, consolidation strategies are designed to store solutions to these problems for later reuse during routine processing. In the case of the fix-cxns discussed above, the transfer from the meta-layer to the routine layer is rather straightforward. As the fix-cxns work in exactly the same way as regular constructions, they can, in principle, simply be added to the construction inventory. However, only fixes that successfully repaired a problem should be consolidated. This is done by only storing fixes of branches in the search tree that ultimately led to a solution node. It is not easy to strike the optimal generality-specificity balance for a fix. Fixes that are too specific will only apply to exactly the same observation in the future, whereas fixes that are too general might apply to transient structures they should not apply to. This dissertation makes a major contribution to this generality-specificity balance by incorporating *anti-unification* (see chapter 5) and *pro-unification* (see chapter 6) algorithms in FCG.

A schematic representation of the meta-layer architecture in FCG is shown in Figure 3.7. Diagnostics are run after every construction application. The transient structure in which a problem is diagnosed is shown in orange. The transient structure that is the result of applying the fix-cxn is shown in green. At the end of the branch of the search tree, the fix-cxn is added to the construction inventory for later reuse in routine processing.

3.5.2 The Meta-Layer in Evolution Experiments

The use of a meta-layer architecture based on diagnostics and repairs has a significant history in evolutionary linguistics experiments. Beuls et al. (2012) describe three levels on which a meta-layer architecture is relevant in such experiments. The first level is the *language processing level*, on which diagnostics and repairs are run after each construction application, just like we explained in the previous section. Examples of diagnostics and repairs on language processing level implemented in earlier versions of

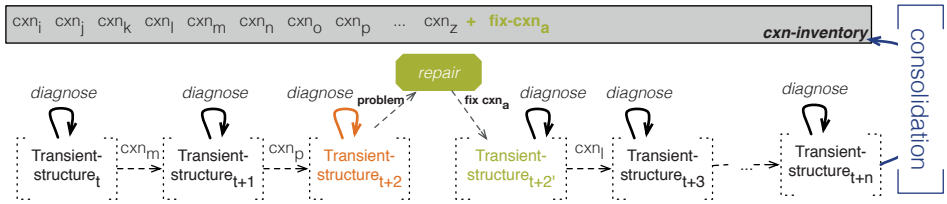


Figure 3.7: A schematic representation of FCG's meta-layer architecture. After each construction application, a set of diagnostics is run and a set of repairs tries to create fixes for the diagnosed problems. If a solution is found, the fixes that were created in that branch of the search tree are added to the construction inventory for later reuse in routine processing (consolidation). Figure adopted from Van Eecke and Beuls (2017).

FCG are presented by Steels and van Trijp (2011) and van Trijp (2012). The second level is the *process level*, in which diagnostics and repairs monitor each process in the semiotic cycle. Problems can be diagnosed and repaired after perception, conceptualisation, comprehension, formulation or interpretation. The third level is the *agent-level*, on which the meta-layer influences agent behaviours, such as turn-taking. This level is needed for problems that cannot be diagnosed or repaired within one process in the semiotic cycle, for example when re-entrance[12] is required.

In this dissertation, I focus on diagnostics and repairs on the language processing level only. My contribution is twofold. First, I have achieved a tighter integration of the meta-layer architecture into the FCG system. Whereas in previous versions of FCG, the grammar engineer needed to extend the system with his own classes and hooks for communication with the meta-layer, these classes and hooks are now included in the standard FCG distribution and are easily usable with any FCG grammar. Second, I have implemented a library of diagnostics and repairs for common language processing problems. With only minor modifications, these diagnostics and repairs can be used with almost any grammar. My work on the tighter integration of the meta-layer in standard FCG and on the library of diagnostics and repairs was done in collaboration with Katrien Beuls and a paper discussing the results was published as Van Eecke and Beuls (2017).

[12] Re-entrance refers to the capacity of an agent to use its language processing system in comprehension to monitor the output of its language processing system in formulation and vice versa. See Steels (2003) for a discussion of the concept and its role in language emergence and evolution, and Van Eecke (2015) for an application of the concept to robust language processing in FCG.

3.5.3 Library of Diagnostics and Repairs

The power of a meta-layer architecture lies in the diagnostics, repairs and consolidation strategies that are used. Although different tasks and grammars often require highly specific diagnostics and repairs, certain problems occur in almost any task and grammar. In order to solve these reoccurring problems in a user-friendly and standardised way, we have included a library of basic diagnostics and repairs in the FCG system. The following situations are covered:

- In comprehension, the input contains a form that is not covered by any construction. A new lexical construction needs to be created, mapping the perceived form to a hypothesized meaning predicate.
- In formulation, the input contains a meaning predicate that is not covered by any construction. A new lexical construction needs to be created, mapping the predicate to a new form.
- In comprehension or formulation, no solution can be found. This can be due to a case of coercion, an agreement mismatch, a word order error or a similar problem. Conflicting features of an existing construction need to be relaxed, in order for the construction to apply.
- In comprehension or formulation, variables in two meaning predicates need to be bound to each other. A new phrasal construction needs to be created, capturing the variable equalities and the word order or markers.

Table 3.3 presents the problems, diagnostics, repairs and consolidation strategies that cover these situations in the library. The table serves as a brief overview only, a comprehensive discussion of the algorithms that are used will follow in chapters 5 and 6 on anti-unification and pro-unification.

3.6 FCG Interactive: Web Service and API

In order to facilitate the dissemination of FCG, and in particular of the research project presented in this dissertation, I have created an interactive web service and API for the FCG framework. While the interactive web service provides interested people with the opportunity of getting to know FCG without needing to install the software environment on their computers, the API makes it easy for developers to integrate FCG as a language processing component into any application.

Table 3.3: An overview of the different problems, diagnostics, repairs and consolidation strategies that are used in FCG's meta-layer library.

Problem	Diagnostic	Repair	Consolidation
Unknown-word (comprehension)	No more applicable cxns + strings in root	Create new lexical cxn (hypothesized meaning).	Add to cxn-inventory.
Unknown-word (formulation)	No more applicable cxns + meaning predicates in root.	Create new lexical cxn (new form).	Add to cxn-inventory.
Matching-conflict (comprehension or formulation)	No more applicable cxns + unconnected meaning + anti-unification possible with low cost.	Anti-unification of existing construction with transient structure.	Pro-unification and add to cxn-inventory.
Missing-phrasal-cxn (comprehension or formulation)	No more applicable cxns + unconnected meaning + anti-unification not possible with low cost.	Create new phrasal cxn (variable equalities and word order / markers).	Pro-unification and add to cxn-inventory.

3.6.1 FCG Interactive Web Service

The *FCG Interactive* web service was launched on the 17th of October 2016 at the 'Intensive Science Scientific Festival', an event organised by Sony Computer Science Laboratory Paris for celebrating its 20th anniversary. Since then, it is publicly available at `https://www.fcg-net.org/fcg-interactive`. The web service has Fluid Construction Grammar running under the hood, with grammars for different languages loaded in memory. The user can select a grammar and enter the utterance or meaning representation that he wants to process. Depending on the input, he can then click 'comprehend' to comprehend the input utterance, 'comprehend-and-formulate' to comprehend the input utterance and reformulate the comprehended meaning, 'formulate' for formulating the input meaning representation, or 'formulate-and-comprehend' for formulating the input meaning representation and recomprehending the produced utterance. The user can also choose between two visualisation options, the first one including the complete construction application process, and the second one including the final result only and hence speeding up the whole process with a few seconds.

The visualisations shown by the interactive web service are generated using FCG's default, browser-based visualisation library, which makes ample use of expandable/collapsible elements. These elements allow the presentation of a very clear, high-level overview of the construction application process and its result, while full detail on intermediate results, including transient structures, applied constructions and bindings,

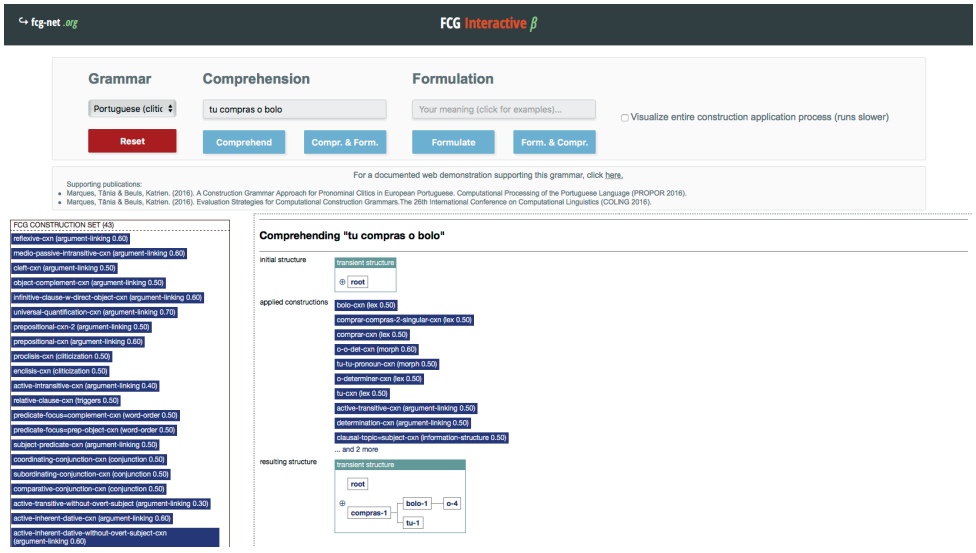


Figure 3.8: A screenshot of the FCG Interactive web service, comprehending the utterance 'tu compras o bolo' ('you buy the cake') with a grammar focussing on Portuguese clitics implemented by Marques and Beuls (2016).

are only a single click away.

A screenshot of the FCG Interactive web service in use is shown in Figure 3.8. The upper part shows the fields that take input from the user: grammar name, utterance to comprehend or meaning representation to formulate, and requested visualisation. The lower part displays the construction inventory of the chosen grammar on the left, and the analysis of the processed utterance/meaning representation on the right. The screenshot shows the comprehension process for the utterance 'tu compras o bolo' ('you buy the cake'), using a grammar that focuses on clitics in Portuguese (Marques and Beuls, 2016).

3.6.2 FCG Interactive Web API

In order to make it easier to integrate FCG as a language processing component into other applications, I have developed a web API (application programming interface) to the FCG environment. The API defines a standardised, yet extensible, request-response based protocol for the communication between external applications and FCG. The API was launched at the same time as the web service and provides the link between the web service and the underlying FCG system.

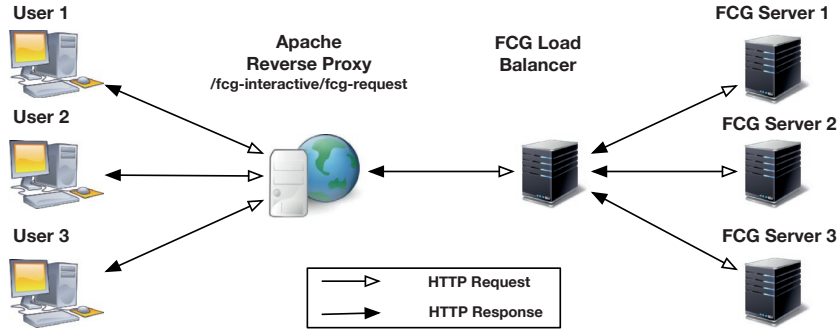


Figure 3.9: A schematic representation of the architecture behind the FCG Interactive web API.

API Architecture

The general architecture of the web API is based on HTTP requests and responses. From the client side, http requests are sent to a single endpoint, namely `https://www.fcg-net.org/fcg-interactive/fcg-request`. For security reasons, the request first arrives at an apache server that is configured as a reverse proxy. The reverse proxy sends the request to a custom-written load balancer that checks the load of the available FCG servers. It forwards the request to the least busy server, which handles the request and formulates a response. Then, the response travels back through the load balancer and the reverse proxy to the user. A schematic representation of this architecture is shown in Figure 3.9.

At an individual FCG server, the different HTTP requests all arrive at the same port. In order to assure a correct handling, each request should include a handler-method field, which specifies with which method the request needs to be handled. When a request arrives, it is first parsed into the *request-type* (e.g. GET or POST), the *handler-method*, and a *data* field that groups together all other information passed in the request. Then, a method named `handle-http-request` is invoked with as arguments the request-type, the handler-method and the data field.

The API contains different `handle-http-request` methods that each specialise on a request type and a handler-method. This makes the API easily extensible, as a new functionality can be added by simply adding an additional `handle-http-request` method. If a request then passes the corresponding handler-method name, it will automatically be directed towards this method. Figure 3.10 shows a didactic example of such a method that will be invoked when an HTTP POST request with the handler-method 'draw-cxn-inv' (for drawing a construction inventory) is sent to the server. Note

```
(defmethod handle-http-request (http-request
                                (request-type (eql :post))
                                (handler-method (eql :
                                    draw-cxn-inv)))
  "Handle method returning a visualisation of the cxn-inventory"
  (let* ((data (cdr (assoc "DATA" http-request :test 'equalp)))
         (cxn-inventory (upcase (cdr (assoc "CXN-INVENTORY" data
                                            :test 'equalp)))))
    (make-html (eval (find-symbol cxn-inventory)))))
```

Figure 3.10: Code fragment featuring an example of a `handle-http-request` method that will be invoked when an HTTP POST request with the handler-method 'draw-cxn-inv' is sent to the server. HTML code visualising the construction inventory that was passed as data will be returned.

the specialisers of the second and third argument of the method, :post and :draw-cxn-inv respectively. The method first retrieves the data field from the HTTP request, and retrieves the specified construction inventory from the data field. It then finds the construction inventory object in FCG and returns the HTML code of its visualisation as a string.

API Specification

Within the architecture described above, the API specifies a set of handler-methods that can be used by external applications, provided of course, that the corresponding request type is sent with the appropriate data. An overview of these methods is presented in Table 3.4. At this moment, the API only supports the *use* of FCG grammars, i.e. comprehending and formulating, and not their *design*. This does however not mean that the grammars are static, as the meta-layer can be active and scores and network links can be updated during use. If in the future, an application would need to design grammars through the API, additional handler-methods for creating construction inventories or adding new constructions could easily be implemented within this architecture.

Grammar Requirements

The API has been designed in such a way, that it does not put many restrictions on the grammars. In fact, it works flawlessly with any FCG grammar as long as the grammar runs in standard FCG and the following basic guidelines are respected.

Table 3.4: Specification of the web API to the FCG Interactive Server

Handler-method	Type	Data	Documentation
`fcg-comprehend` `fcg-comprehend-and-formulate`	get/post	`utterance=`*utterance* `cxn-inventory=`*name* `package=`*package* `visualisation=`*nil/t/link-only* `monitor=`*monitor*	Comprehends the *utterance* with the grammar stored under *cxn-inventory* in *package*, with *monitor* activated. Depending on *visualisation*, returns the resulting meaning representation (*nil*), html code with a visualisation of the process (*t*), or a url pointing towards an html page with this visualisation (*link-only*). In case of *fcg-comprehend-and-formulate* also formulates the comprehended meaning.
`fcg-formulate` `fcg-formulate-and-comprehend`	get/post	`meaning=`*meaning* `cxn-inventory=`*name* `package=`*package* `visualisation=`*nil/t/link-only* `monitor=`*monitor*	Formulates the *meaning* with the grammar stored under *cxn-inventory* in *package*, with *monitor* activated. Depending on *visualisation*, returns the resulting utterance (*nil*), html code with a visualisation of the process (*t*), or a url pointing towards an html page with this visualisation (*link-only*). In case of *fcg-formulate-and-comprehend* also comprehends the formulated meaning.
`fcg-get-example-sentences` `fcg-get-example-meanings`	get/post	`cxn-inventory=`*name*	Returns a string of comma-separated example sentences / meanings for the grammar stored under *cxn-inventory*.
`fcg-get-cxn-inventory`	get/post	`cxn-inventory=`*name*	Returns a url pointing to an html file with a visualization of the grammar stored under *cxn-inventory*.
`fcg-get-reference-text`	get/post	`cxn-inventory=`*name*	Returns a documentation string on the grammar stored under *cxn-inventory*, including focus, author names and relevant publications.
`static-html-busy`	get/post	*no data required*	Returns "NIL" if the server can accept a new request and "T" otherwise.

- The construction inventory should be accessible via a global variable.
- A *reference text* is provided, i.e. a snippet of html code describing the authors of the grammar, as well as relevant publications.
- A text file with *example utterances* is provided.
- A text file with *example meaning representations* is provided.
- A link to an html page visualising the *construction inventory* is provided.

3.7 Conclusion

The main aim of this chapter was to lay out the main architecture and design concepts of Fluid Construction Grammar, on which the rest of this dissertation will build. I have described how FCG implements constructional language processing as a problem solving process, with its state representations (transient structures), operators (constructions), goal tests, heuristics and optimization strategies. I have especially focused on my own contributions to the FCG system, which can be summarized in the following three points:

1. I have implemented a mapping between a new high-level FCG notation and the FCG-2011 system. The high-level notation is used for all interfacing with the system, including the representation of grammars and visualisation of the construction application process and processing result. The mapping has already shown its worth in multiple grammars and evolution experiments.
2. I have tightly integrated an existing meta-layer architecture into FCG, extending the construction inventory with slots for holding diagnostics and repairs, and providing hooks for running diagnostics, repairs and consolidation strategies at the appropriate places in the construction application pipeline. I have also developed a library of basic diagnostics and repairs for frequent problems, which can readily be used with any grammar.
3. I have implemented an interactive web service that can be used to query FCG grammars and visualise the construction application process and processing result. I have also implemented a web API, which allows an easy embedding of FCG into external applications.

One of the most important properties of Fluid Construction Grammar is that it is particularly flexible and open-ended. It is a general problem solving architecture using rich and powerful data structures, in part due to its support for logic variables in

transient structures and constructions. There are virtually no restrictions other than the design choices discussed in this chapter. Feature names and values are completely open-ended and can be dynamically added to or removed from the grammar. If needed, specialised render and de-render methods, node tests, goal tests and construction suppliers can be implemented and used without requiring any change to the system itself.

The open-ended nature of FCG, as well as its powerful meta-layer architecture, reflect that FCG is designed with the image of a dynamic and evolving language use in mind. For this reason, problem solving and learning are primary concerns. Errors and innovations are not seen as 'special cases', but as the necessary variation thriving language evolution. Powerful problem solving and learning strategies are the only hope that a language processing system will ever be able to communicate in a humanlike way and keep its grammar up-to-date with the ever-evolving conventions of the language community. While the present chapter has laid out the architecture of the system, the next chapters will present these powerful problem solving and learning strategies in detail.

Chapter 4

A Type Hierarchy System for FCG Symbols

4.1 Introduction

This chapter introduces a type hierarchy system for Fluid Construction Grammar. The system makes it possible to capture generalisations about the categories used in a

grammar, while preserving the fluid and open-ended nature of the formalism. The chapter is structured as follows. I will first discuss the nature of categories in FCG (4.2). Second, I will introduce the type hierarchy system and its implementation (4.3). Then, I will demonstrate different aspects of its use through examples (4.4). Finally, I will compare it to the use of typed feature structures in other grammar formalisms (4.5).

The examples that are used in this chapter can be explored in full detail at `https://www.fcg-net.org/demos/vaneecke-phd/type-hierarchies`. These examples are chosen for didactic reasons only and aim to demonstrate the main properties of of the type hierarchy system and its implementation. The actual use of the type hierarchy system in evolutionary linguistics experiments, namely to capture categorical networks, will be discussed in Sections 5.6 and 7.4.4 below.

4.2 The Nature of Categories in FCG

The previous chapter has discussed the use of feature structures for representing constructions and transient structures in FCG. The feature names that are used and the atomic or complex values that these features can take, are not predetermined, but are entirely open-ended. The feature names are always constants[1] and the atomic elements in their values are either constants or logical variables. Neither constants nor logical variables have any *meaning* apart from how they are used within the constructions of a grammar. When, for example, a construction contains a feature 'lex-class' with value 'noun' and a feature 'sem-class' with value 'animal', then each of these symbols in itself has no meaning for the FCG interpreter. FCG does not know about lex-classes, sem-classes, nouns or animals at all. Feature names and atomic elements in their values are all treated purely as *symbols*. Symbols can only be compared to each other in terms of equality, which is for example the case inside the matching and merging algorithms. In fact, substituting each occurrence of 'lex-class' in a grammar by 'y-222' and each occurrence of 'noun' by 'x-256' or any other symbol would not make any difference in terms of processing and results, as long as it is done consistently. Although symbol names can thus be chosen in a completely random fashion, those that are used in the examples that are shown in this dissertation do reflect their function in the grammar, in order to enhance the readability of the grammar for humans.

Because FCG treats all feature names and atomic elements in their values as either

[1] Feature names are always constants, but unit names in constructions are usually logical variables. They can be constants in exceptional cases, when they refer to a specific unit in the transient structure, e.g. 'root'.

constants or logical variables, there is no need to tell the FCG engine which features or values will be used by a particular grammar. There is no centralised *type definition system* that defines which features can occur and which values a particular feature can take.[2] The feature structures used in FCG are *untyped*, which is an important advantage for a formalism that is used in evolutionary linguistics experiments. In these experiments, new features and values can be invented at any time and are incorporated into existing or new constructions. In these conditions, it would be a very difficult task to keep a type definition system up-to-date and consistent with the ever-evolving constructions of a grammar.

However, the use of untyped feature structures also has a disadvantage. It makes it difficult to capture certain generalisations that are easily captured using *typed* feature structures, especially in the case of ontological categorisations. Consider the following example, in which a grammar contains lexical constructions for the words '*sparrow*', '*pigeon*', '*crow*' and '*magpie*'. Each of these lexical entries contains a feature 'semantic-class' with value 'bird'. Any construction can use this specific feature to select for birds. Now, imagine that a construction wants to select for animals. Obviously, the construction cannot match on the 'semantic-class: animal' feature, as '*sparrow*', '*pigeon*', '*crow*' and '*magpie*' have 'bird' as the value for this feature. The problem can be solved by changing the value of 'semantic-class' in each of the lexical entries into the set {*bird, animal*}, such that the construction can match on this feature. Although this is a working solution in this case, it is not a scalable solution. Birds belong to many more categories, including vertebrates, chordates and tetrapods. Moreover, many characteristics are associated to each of these categories, for example oviparous, endothermic, toothless and beaked. Storing this amount of ontological information locally inside each construction would not only be inelegant, but processing these massive feature structures would also be computationally expensive. Moreover, the systematic relationship between 'bird' and 'oviparous' would still not be explicitly captured, but implicitly at best, if in every construction that has 'bird' as an element of its 'semantic-class', 'oviparous' is also part of this set.

The challenge that I take up in this chapter is to incorporate one of the major advantages of typed feature structures, namely the possibility to generalise over categories, into FCG, without loosing the important advantages of untyped feature structures,

[2] This might sound odd, as the previous chapter has introduced the use of 'feature types' in FCG (see Section (3.4.3) of the previous chapter). The feature type declaration is used to influence the unification algorithms (match and merge) that are used for specific features during processing. Practically speaking, a unification algorithm that is specialised on a feature type might not cover all possible structures for the value of that feature. For example, a specialised matching algorithm for sets might expect sets and not support atomic values. Although the feature type declaration can in this way put restrictions on the structure of the values of particular features, it does not put limits on which symbols can occur as value, or, if the value is complex, on which features it can hold.

namely the possibility to extend the grammar on-the-fly with novel features and categories.

4.3 A Type Hierarchy System for FCG symbols

I tackle the challenge formulated in the previous section by integrating an efficient and flexible type hierarchy system into FCG. The type hierarchy system makes it possible to express generalisations over categories, while respecting the fluidity of FCG. It is dynamic in the sense that it can be built up, extended and altered during processing. Moreover, the type hierarchy system only alters FCG's processing behaviour for symbols that occur in the type hierarchy, while it is left unchanged for all other symbols.

4.3.1 Type Hierarchy Concept

In FCG, constructions can apply to transient structures through matching and merging, which are unification processes. The algorithms implementing these processes make construction application either succeed or fail. One type of failed case is when two different constants (not logical variables) need to be unified. In Figure 4.1 for example, the *lay-an-egg-cxn* on the right cannot apply to the transient structure on the left. The construction requires an NP unit with a feature 'semantic-class: oviparous' as its subject. The transient structure on the left, which was built with 'the crow' as input does not contain such a unit. It only contains an NP unit with the feature-value pair 'semantic-class: crow'. Because 'crow' and 'oviparous' are two different symbols, their unification fails and the construction cannot apply.

The fact that the 'lay-an-egg-cxn' cannot apply to the transient structure because it requires its subject to be an NP of 'semantic-class' 'oviparous' and observes an NP of 'semantic-class' 'crow' might be counterintuitive for humans, as crows are a proper subset of oviparous animals. If we want the construction to apply, we need to inform the FCG system that the symbol 'crow' is a subtype of the symbol 'oviparous'. This is done by creating a type hierarchy in the form of a network. The nodes in the network are constants, and the directed edges indicate subtype relations. As it can be seen on Figure 4.1 in the middle, the type hierarchy contains the information that the symbol 'crow' is a subtype of the symbol 'bird' and that the symbol 'bird' is a subtype of the symbol 'oviparous'. Taking into account this information while matching and merging the construction with the transient structure leads to a successful construction application, as shown at the bottom of Figure 4.1. For symbols that are not connected

through the type hierarchy, or that do not occur in it, FCG's default matching and merging behaviour is preserved.

4.3.2 Type Hierarchy Implementation

A type hierarchy is defined on grammar level, and is stored in the *:type-hierarchy* field of the *data* blackboard, which is a slot in the construction-inventory. The type hierarchy is a directed graph, in which the vertices are symbols and the arcs represent subtype relations. The graph is implemented in a scalable way as an object consisting of a collection of hash tables holding the nodes, the incoming arcs, the outgoing arcs, the labels, etc. For the representation and processing of the graphs, an open-source package called 'graph-utils'[3] is used. This package is hidden from the FCG user, who interacts with the type hierarchy of a grammar using the following interface functions in the FCG package:

- *get-type-hierarchy (cxn-inventory)*: returns the type hierarchy of a construction inventory, and sets it when used with *setf*.
- *make-instance ('type-hierarchy)*: CLOS function used to create a new object of the type-hierarchy class.
- *add-category (type type-hierarchy)* and *add-categories (types type-hierarchy)*: adds one or more categories as vertices to the type-hierarchy.
- *node-p (type type-hierarchy)*: returns true if type is a node in the type hierarchy.
- *add-link (subtype supertype type-hierarchy &key (weight 0.5))*: Adds an arc between two symbols in the hierarchy.
- *link-weight (subtype supertype type-hierarchy)*: Returns the weight of the type-hierarchy link.
- *set-link-weight (subtype supertype type-hierarchy weight)*: Sets the weight of the type-hierarchy link
- *incf-link-weight (subtype supertype type-hierarchy delta)*: Increments the weight of the link with delta.
- *decf-link-weight (subtype supertype type-hierarchy delta)*: Decrements the weight of the link with delta.

[3] https://github.com/kraison/graph-utils

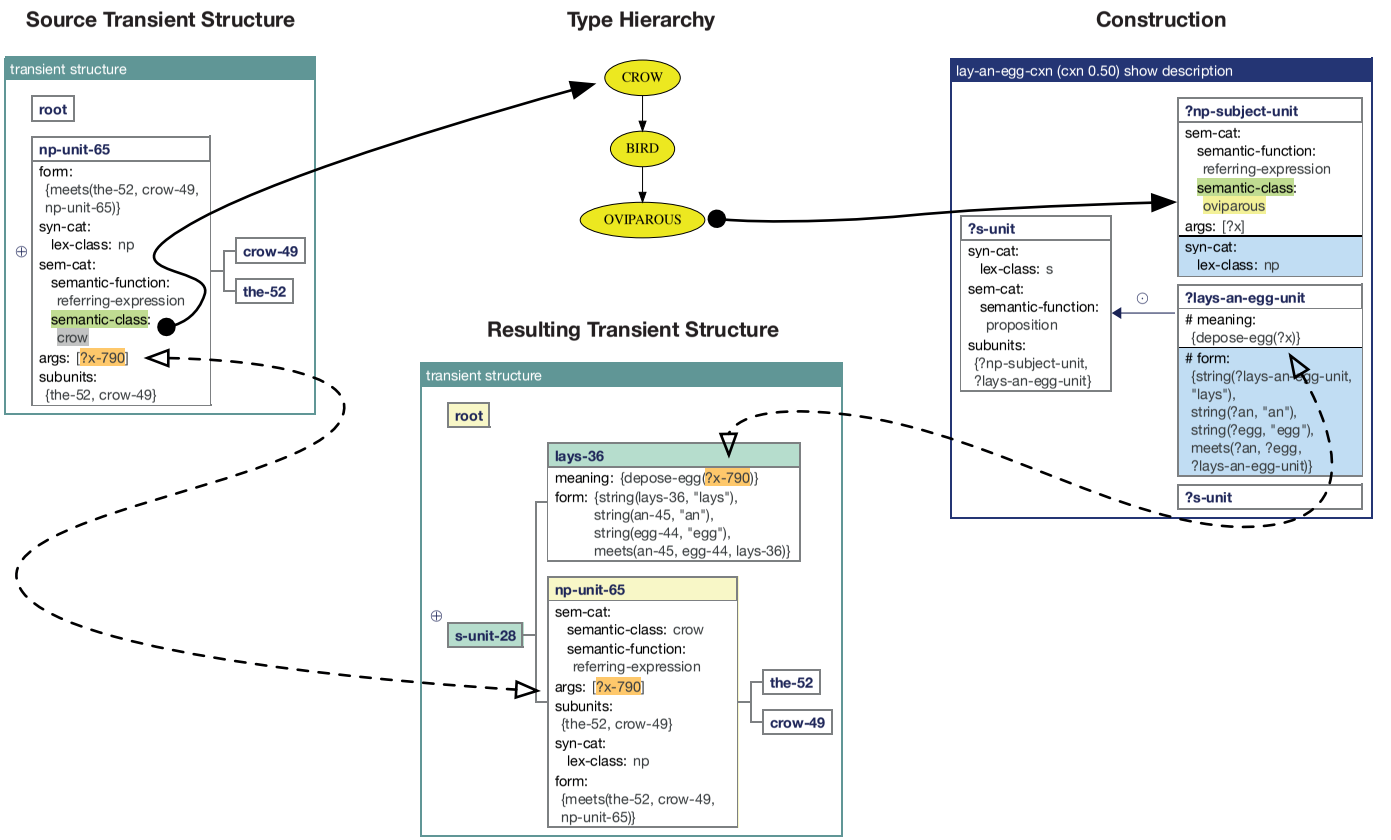


Figure 4.1: A schematic representation of the integration of a type hierarchy system in FCG. The construction can apply because a path from 'crow' to 'oviparous' is found in the type hierarchy.

- *make-html (type-hierarchy)*: Generates HTML code visualising the type-hierarchy as a graph.
- *directed-path-p (subtype supertype type-hierarchy)*: Returns true if there is a path from subtype to supertype in the type-hierarchy, nil otherwise.
- *directed-distance (subtype supertype type-hierarchy)*: Returns the cost of the shortest path from subtype to supertype in the type-hierarchy, nil if no path exists.
- *connected-p (type1 type2 type-hierarchy)*: Returns true if there is a path between type1 and type2, regardless of the direction of the edges, nil otherwise.
- *undirected-distance (type1 type2 type-hierarchy)*: Returns the cost of the shortest path from type1 to type2 regardless of the direction of the edges, nil if no path exists.

The arcs are weighted and by default, each arc has a weight of 0.5. The distances returned by the interface functions are the sums of the weights on the arcs along the path. The shortest path is also defined in terms of the weights on the arcs and is calculated using Dijkstra's algorithm (Dijkstra, 1959).

4.3.3 Type Hierarchy: Match and Merge

At this point, a representation for type hierarchies is in place, as well as interface functions for building, extending and querying the hierarchies. Now, FCG's matching and merging algorithms need to be adapted, in order to take into account the type hierarchy that is specified inside the construction inventory.

Type hierarchies only need to be taken into account when two constants are being processed, because only constants can occur as nodes in the hierarchy. For processing two constants, FCG's matching and merging algorithms use the same function, namely *unify-atom*. When unify-atom succeeds, it returns the list of bindings for which it succeeds and when it fails, it returns the constant +fail+, which evaluates to nil. The default unify-atom algorithm succeeds in the following three cases only:

1. The two atoms are equal to each other.
2. The first atom is a logical variable and the *unify-variable* function succeeds with the two atoms and the list of bindings as arguments.
3. The second atom is a logical variable and the unify-variable function succeeds with the two atoms and the list of bindings as arguments.

In order to take the type hierarchy of the construction inventory into account, a fourth case is added.

4. There exists a path in the type hierarchy of the construction-inventory from the symbol in the transient structure to the symbol in the construction.

The algorithm is presented in pseudo-code in Algorithm 1. The code printed in black belongs to the original unify-atom algorithm and the code printed in blue shows the case that was added. Note that unify-atom only succeeds if the path goes from the symbol in the transient structure to the symbol in the construction, and not the other way around. The symbol from the construction should always be the more general symbol. For example, when a construction matches on 'noun', then a 'common-noun' is also acceptable. But when a construction matches on 'common-noun', then 'noun' is not specific enough to be matched.

Algorithm 1: unify-atom adapted for use with type hierarchies

```
input: cxn-atom                          // an atom from the construction
       ts-atom                  // an atom from the transient structure
       bindings                                    // a bindings list
       cxn-inv                          // the construction inventory

// If unification succeeds, the algorithm returns the list of bindings
   for which it succeeds, otherwise it returns +fail+.

if equal(cxn-atom, ts-atom) then                    // symbols are equal
 | return bindings

else if variable-p(cxn-atom) then                 // cxn-atom is a variable
 | return unify-variable(cxn-atom, ts-atom, bindings, cxn-inv)

else if variable-p(ts-atom) then                   // ts-atom is a variable
 | return unify-variable(ts-atom, cxn-atom, bindings, cxn-inv)

else if directed-path-p(ts-atom, cxn-atom, get-type-hierarchy(cxn-inv))
 then                                              // path in type-hierarchy
 | return bindings

else
 | return +fail+
end
```

4.4 Examples

I will now demonstrate different aspects of the use of type hierarchies in FCG through four more elaborate examples. The first example shows that a type hierarchy can

group many different kinds of information, ranging from very formal to very semantic categories (4.4.1). The second example shows how the knowledge contained in a type hierarchy can be exploited when new words are learned (4.4.2). The third example presents two ways of dealing with prototypical generalisations and the cancellation of these generalisations in the case of exceptions (4.4.3). Finally, the fourth example demonstrates how the weights on the links can indicate the entrenchment of the relations captured in the type hierarchy (4.4.4).

4.4.1 Diversity among Categories

While presenting the type hierarchy system in the previous sections, I have only used examples of semantic categories that were connected through the type hierarchy: crows are birds, birds are vertebrates, and birds are oviparous. These semantic features were then used in constructions to match on particular units in a transient structure, e.g. the lay-an-egg-cxn selected for an oviparous subject-unit. Although this is an intuitive use of a type hierarchy, it is certainly not the only one possible. A type hierarchy can combine many different kinds of information, whether it is used for semantic, morphological, syntactic or other purposes. This first example will walk you through three cases in which progressively more knowledge is transferred from the constructions to the type hierarchy. The first case, visualised in Figure 4.2, does not use any type hierarchy. The second and third case demonstrate a mild (Figure 4.3) and extensive (Figure 4.4) use of the type hierarchy system respectively.

Case 1: No Use of Type Hierarchies

I will use the same example as before, namely a small grammar that consists of four constructions. Together, these constructions can process the utterance 'the crow lays an egg'. The four constructions are shown in Figure 4.2. When we look at the categorisations that are required and added by the constructions, we see that the 'the-cxn' specifies that the word 'the' is an 'article', a 'determiner' and an 'identifier'. The 'crow-cxn' specifies that the word 'crow' is a 'count-noun', a 'common-noun' and a 'noun', that it is a 'bird' and a 'physical-object' and that it is 'oviparous' and 'flying'. The 'np-cxn' combines a 'determiner' that is also an 'identifier' and a 'common-noun' that is also a 'physical-object' into an 'np' that is also a 'referring-expression'. Finally, the 'lay-an-egg-cxn' takes an 'np' that is also a 'referring-expression' and 'oviparous' as its subject, and combines it with the words 'lay an egg' and their associated meaning *depose-egg(?x)* into a proposition.

These four constructions can process the utterance 'the crow lays an egg' in both comprehension and formulation without the use of a type hierarchy. All categories that are required by the conditional part (i.e. the comprehension and formulation locks) of the constructions were added to the transient structure by merging the contributing part of other constructions. For example, the 'np-cxn' matches in comprehension on a unit that has a feature-value pair 'lex-class: determiner', which is added by the contributing part of the 'the-cxn'. No generalisations about the categories, such as the fact that articles are a subset of determiners, are explicitly captured in the grammar. Only information that is locally specified in the constructions is used. For clarity, the links between the symbols in the contributing part of the lexical constructions and those in the conditional part of the grammatical constructions are highlighted in colour and connected through arrows in Figure 4.2.

Case 2: Mild Use of Type Hierarchies

In this second case, I will use the same grammar as in the first case, but certain generalisations about categories will now be expressed using the type hierarchy of the grammar. This case is visualised in Figure 4.3 and referred to as a 'mild' use of the type hierarchy.

In case 1 above, the features 'lex-class', 'semantic-class' and 'semantic-properties' of the lexical constructions, namely the 'the-cxn' and the 'crow-cxn', had as value a set containing multiple symbols. For example, the 'lex-class' of the word 'the' contained both 'article' and 'determiner'. 'article' would then be used by constructions that select for an article only and 'determiner' by constructions that select for any determiner, such as the 'np-cxn' in our grammar. This is however redundant information, as every article is also a determiner. In this second case, we will only specify the most specific category inside the constructions. The 'lex-class' of 'the' is now only 'article', the 'lex-class' of 'crow' is now only 'count-noun' and the 'semantic-class' and 'semantic-properties' of 'crow' are now only 'bird'. Obviously, the 'np-cxn' and 'lay-an-egg-cxn' cannot apply any more, because they match on the more general classes, such as 'determiner', 'physical-object', 'common-noun' and 'oviparous'. The solution consists in declaring the relations between the more specific and the more general classes in the type hierarchy of the grammar. We declare that an 'article' is a 'determiner', a 'count-noun' is a 'common-noun' and a 'bird' is a 'physical-object' and 'oviparous'. Now, the constraints in the conditional part of the 'np-cxn' and the 'lay-an-egg-cxn' are satisfied through the type hierarchy, and the four constructions of the grammar can analyse the utterance 'the crow lays an egg' in both directions. Figure 4.3 shows the constructions and the type hierarchy. The categories that are matched and merged

through the type-hierarchy are connected with arrows.

The use of the type hierarchy allowed to explicitly capture certain generalisations about categories outside of the constructions. The use of these generalisations allowed in turn to reduce the amount of information that needed to be present in each construction.

Case 3: Extensive Use of Type Hierarchies

Case 1 presented a grammar that did not use any type hierarchy. For any construction, all categories that could be used by other constructions for matching, needed to be merged into the transient structure by the construction itself. Case 2 only required the merging of the more specific categories and relied on the type hierarchy for matching on these categories using more general categories. In case 3, we go even further in reducing the number of categories that are explicitly declared inside the constructions, and extensively use the type hierarchy for matching and merging. The grammar for case 3 is visualised in Figure 4.4.

Instead of specifying the most specific category for each feature in the lexical construction, we now assign the same symbol as the value of each feature. The symbol is unique to the construction in which it occurs. For example, the 'crow-cxn' in Figure 4.4 has the symbol 'crow' as value for the features 'lex-class', 'semantic-class' and 'semantic-properties'. The symbol 'crow' does not appear in any other construction. In the type hierarchy, we declare that 'crow' is a subset of 'bird' and that 'bird' is a subset of 'oviparous', 'physical-object' and 'count-noun'. Finally, 'count-noun' is a subset of 'common-noun'. The value of each feature in the lexical construction is now matched via the type hierarchy, which combines in this case semantic and morpho-syntactic information. Without any changes to the 'np-cxn' en the 'lay-an-egg-cxn', the grammar will still give the same processing results. In Figure 4.4, the symbols that are matched through the type-hierarchy are highlighted with arrows.

4.4.2 Exploiting Generalisations for Learning

One of the main advantages of the use of type hierarchies is that the generalisations that they capture can be exploited for learning new constructions based on novel observations. The main idea is that when observing a new word or grammatical structure for the first time, only a very limited part of its behaviour is exposed. The large part that is not exposed is filled in with default knowledge derived from the type hierarchy, as shown in the following examples.

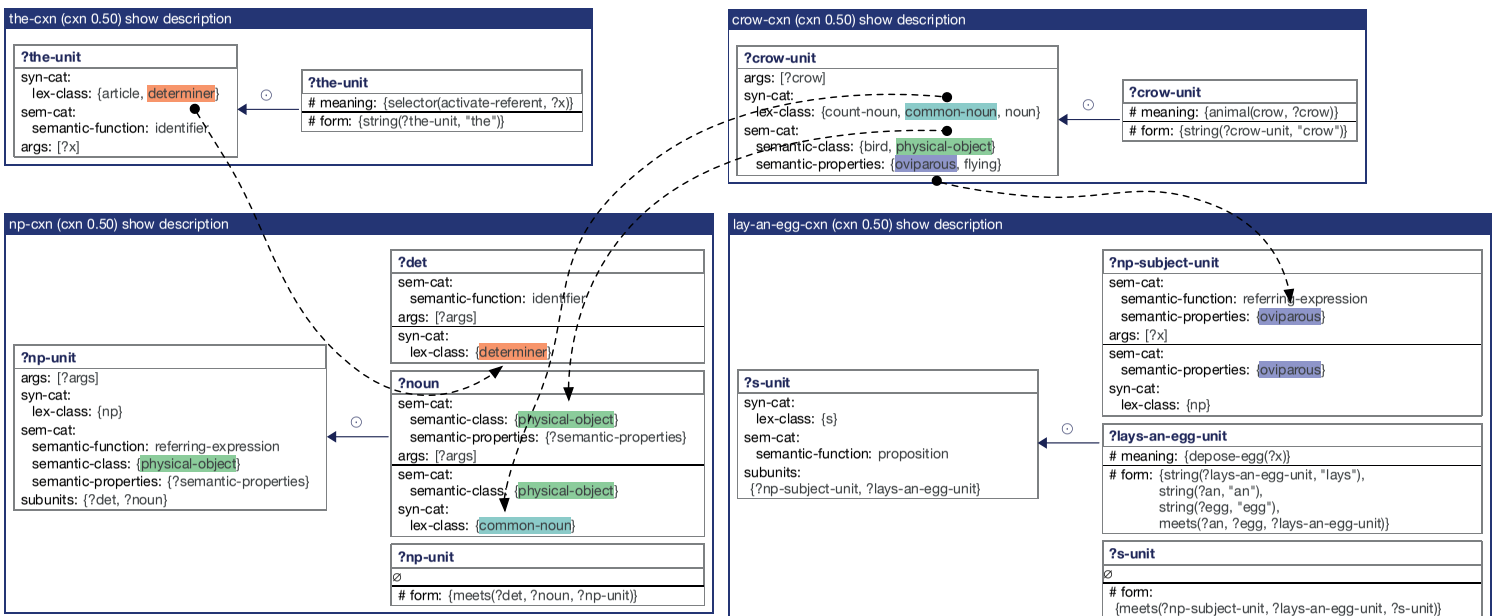


Figure 4.2: Constructions for processing 'the crow lays an egg' without the use of type hierarchies. The arrows and colour highlighting indicate which categories added by the lexical constructions above are used by the grammatical constructions below.

Coercion Imagine a 'transitive-clause-cxn' that matches on a unit that has as 'lex-class' 'verbal' and as 'semantic-class' 'event', like in "*he sees him*" or "*he would have employed him*". The grammar is now faced with the utterance "*he googled him*", in which the slot that normally should have had a verbal category now has a nominal category. Although 'google' has only been observed in a verbal role in a single construction, the grammar will now assume that it can serve a verbal role in general and opens the way for productively using it in other constructions that take a verbal element, such as "*he would have googled him*" or "*he said he would google the hell out of this*". It is the use of generalisations and abstractions that allows the use of this lexical item in other constructions. Note that this example can be implemented using a type hierarchy (cf. Case 2 and 3 of 4.4.1), but also by adding the verbal category inside the lexical construction (cf. Case 1 of 4.4.1).

New bird This second example revisits the example with 'crows' that was worked out in Case 3 of the previous section. Case 3 presented the 'extensive' use of the type hierarchy, in which a single, unique symbol ('crow') was used as the value of each feature and the matching and merging of these symbols relied extensively on the type hierarchy. This grammar was shown in Figure 4.4 and will serve as the starting point for the current example on learning, which is visualised in Figure 4.5. In this example, the grammar is faced with a new concept, namely 'magpie'. FCG's meta-layer architecture diagnoses that the word "magpie" is unknown and a repair will create a new construction for this word. On the conditional part, the construction will map between the form "magpie" and a predicate representing its meaning: (magpie ?x). On the contributing part, the features 'lex-class', 'semantic-class' and 'semantic-properties' all get the value {magpie}. The symbol 'magpie' is at this moment unique in the grammar, it does not appear in any other construction. From the interaction, it can be deduced that a magpie is a kind of bird, so an arc from 'magpie' to 'bird' is added to the type hierarchy. This ensures that the constructions that match on categories linked to 'bird' will be able to match on the unit created by the 'magpie-cxn'. The grammar will be able to use "magpie" in the same constructions as it can use "crow". A visualisation of the use of the 'magpie-cxn' is shown in Figure 4.5.

4.4.3 Cancellation of Generalisations

In the previous section, I have shown that the generalisations that are captured in a type hierarchy are useful when learning new constructions from observations. In all the examples up to this point, I have used relatively safe categories in the sense that more specific categories were strict subsets of more general categories. Birds

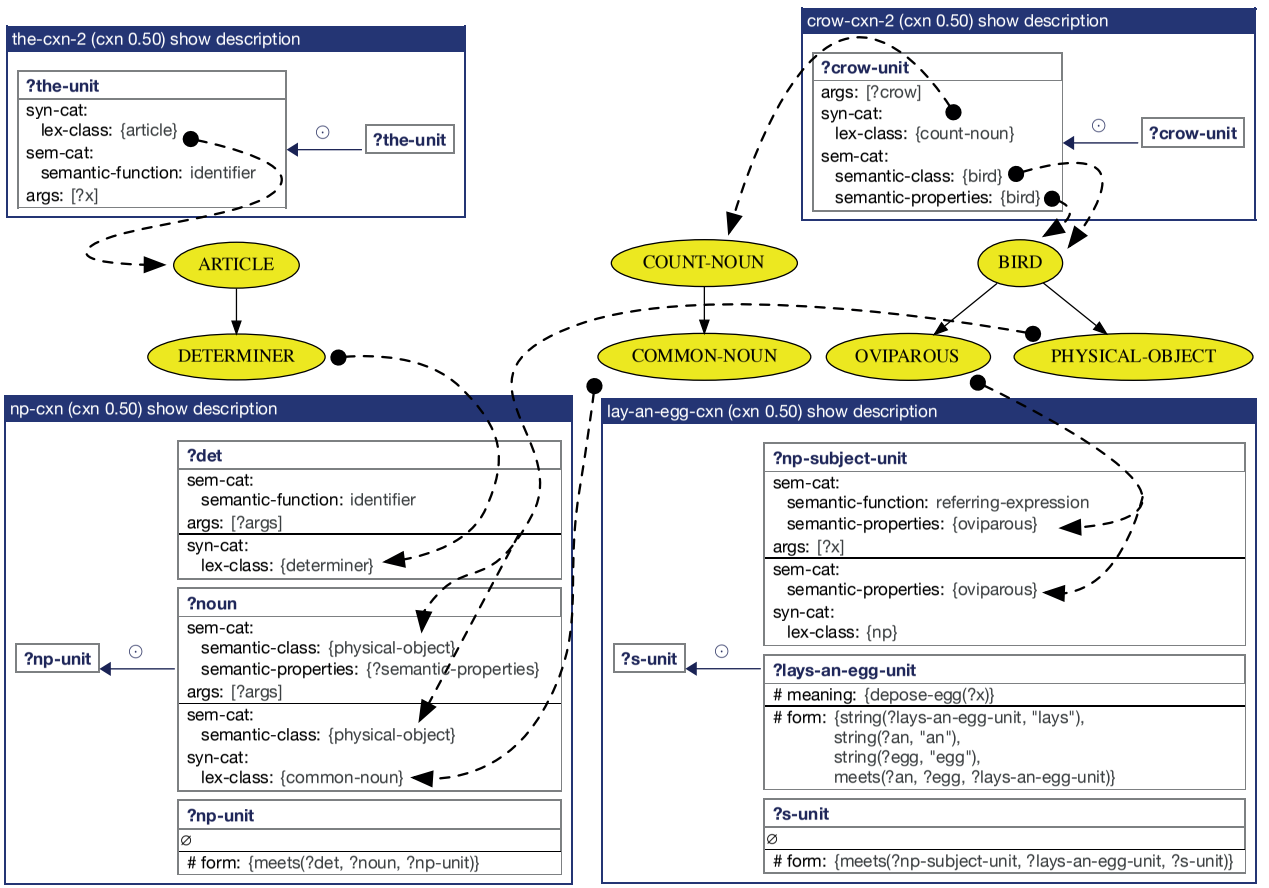


Figure 4.3: Constructions for processing 'the crow lays an egg' with a mild use of the type hierarchy. The arrows show the categories that are matched and merged via the type hierarchy.

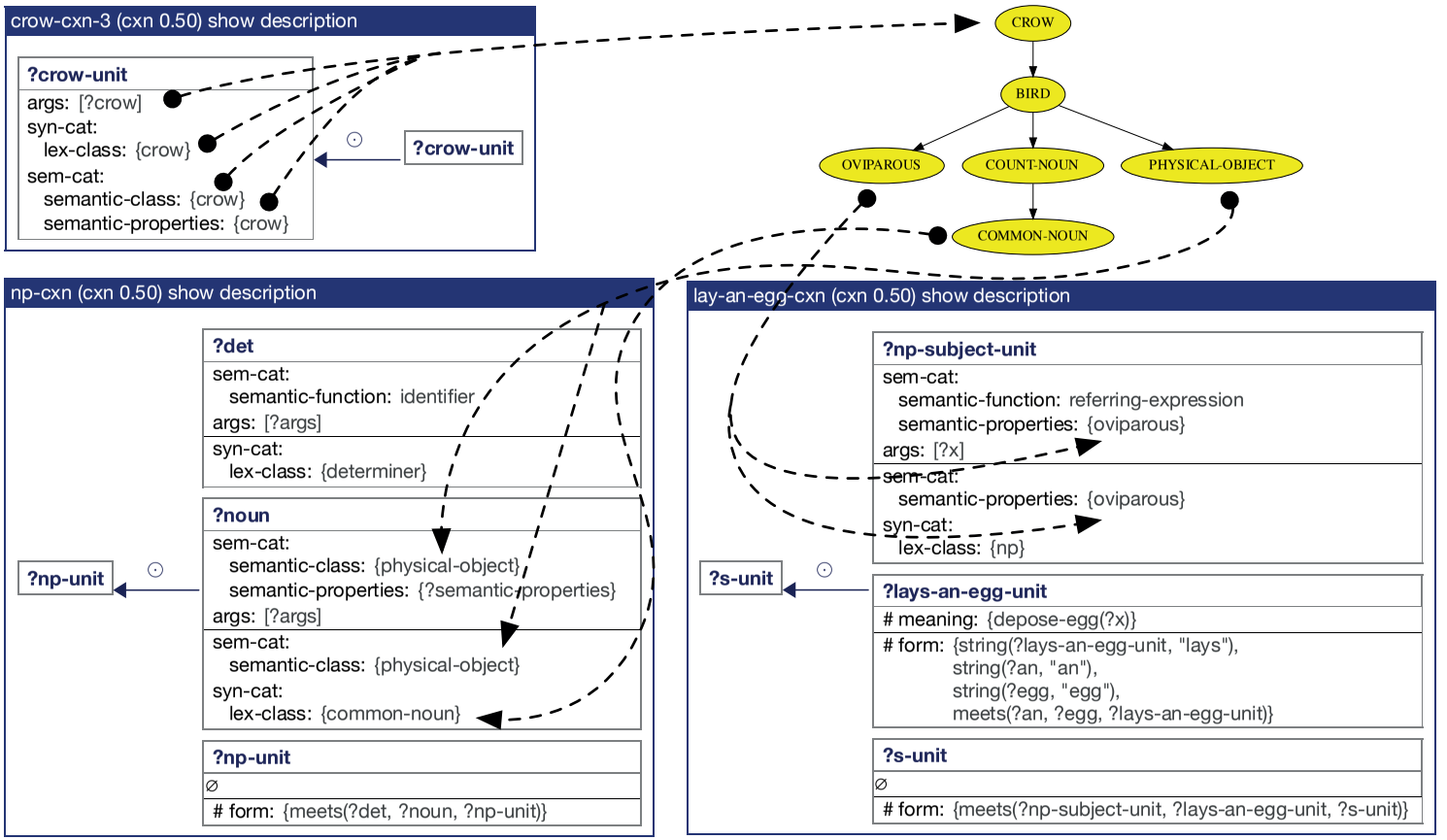


Figure 4.4: Constructions for processing 'the crow lays an egg' with an extensive use of the type hierarchy. The arrows show the categories that are matched and merged via the type hierarchy.

are oviparous and physical objects, and there are probably no exceptions. In general however, relations between categories are often of a much more prototypical nature. Birds typically fly, but ostriches and penguins don't. When learning about a new bird, it is a good idea to assume that it flies, up until the point that the opposite becomes known. In order to capture these prototypical relations as default knowledge, a mechanism for cancelling inferences is needed when exceptions are processed. Two different mechanisms for cancelling inferences are presented, the first one through information in the construction itself and the second through information in the type hierarchy.

The starting point for the two mechanisms is the same. The grammar is very similar to the grammar used in the previous examples, but now the 'fly-cxn' is added. This construction combines the word "flies" and a subject np of which the value of the 'semantic-properties' feature includes 'flying'. We know that birds typically fly, so we add a link from 'bird' to 'flying' in the type hierarchy[4]. This is fine for sentences like 'the crow flies' or 'the magpie flies'. The challenge is to incorporate the knowledge that ostriches don't fly into the grammar.

Cancellation through the Construction

A first mechanism is to add a feature-value pair 'non-prototypical: {}' to each construction. The value of the feature is a set containing all non-prototypical categories of the construction. In the case of 'ostrich', 'non-prototypical: {non-flying}' is specified and in the case of the np, the value of the same feature in the noun copied. In the '?np-subject-unit' on the conditional part of the 'fly-cxn', a feature-value pair 'non-prototypical: {¬ non-flying}' is added. This feature requires that the value of the 'non-prototypical' feature of the '?np-subject-unit' does not contain the symbol 'non-flying'. The 'fly-cxn' can still apply in the utterance "the crow flies", but cannot apply in the utterance "the ostrich flies". A visualisation of this way of cancelling a generalisation from the type hierarchy, namely through information that is local to the construction, is shown in Figure 4.6. The feature that blocks construction application in the matching phase is marked with a red arrow.

Cancellation through the Type Hierarchy

A second mechanism for the cancellation of generalisations is to include information about non-prototypical properties into the type-hierarchy, instead of into the construc-

[4]Note that the aim of this example is to demonstrate that the type hierarchy can capture systematic information, the system does not contain built-in mechanisms for detecting that birds typically fly.

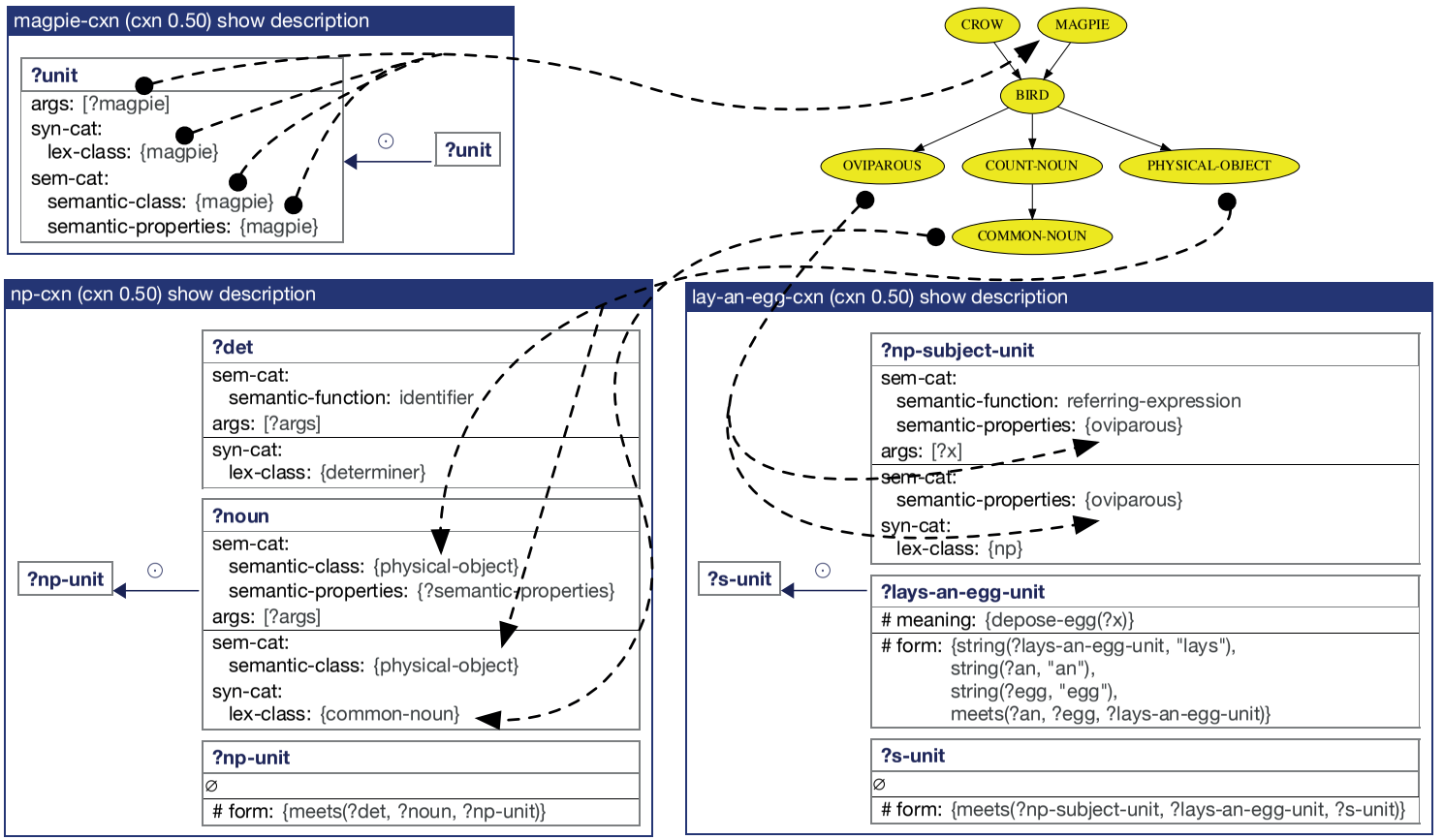


Figure 4.5: Learning of new constructions using the type hierarchy. By adding a link from 'magpie' to 'bird' in the type hierarchy, the use of the symbol 'magpie' will take over the same behaviour of the symbol 'bird'.

tions. Just like for the first mechanism, a feature-value pair 'non-prototypical: {}' is added to the constructions. However, the value is now a single symbol, which is uniquely used inside this construction. For the 'crow-cxn', it could be 'non-prototypical: {crow}' and for the 'ostrich-cxn', it could be 'non-prototypical: {ostrich}'. In the type hierarchy, an arc from 'ostrich' to 'non-flying' is added. The 'fly-cxn' is left unchanged, with the feature-value pair 'non-prototypical: {¬ non-flying}' in the '?np-subject-unit' on the conditional part. Through the type hierarchy, this feature blocks construction application for 'the ostrich flies', but not for 'the crow flies' and 'the magpie flies'. A visualisation of this mechanism is shown in Figure 4.7. Matching links through the type hierarchy are highlighted by black arrows, and the blocking link is highlighted in red.

Interestingly, this grammar also allows 'the bird flies', as bird is a subtype of 'flying', but not a subtype of 'non-flying'. This shows how the prototypical relationship is expressed here. Birds fly in general, but it is possible that specific subtypes of birds, e.g. individuals or subgroups, don't fly. Note that in section 4.2, it was stressed that symbols carry no meaning other than the way they are used in the grammar. This is still entirely true, if the type hierarchy is considered to be part of the grammar. For this reason, it is also not inconsistent to have links from ostrich to both 'flying' and 'non-flying'. There is no explicit notion of complementarity or mutual exclusivity for symbols in the type-hierarchy. The fact that it might seem that 'non-flying' 'overrides' 'flying' in this example is solely an effect of how the 'flying' and 'non-flying' categories are used in the 'fly-cxn'.

4.4.4 Entrenchment of Type Hierarchy Links

The links in a type hierarchy are weighted, and the weights indicate the degree of entrenchment of the links in the grammar. In evolutionary experiments, the weight of a link reflects the confidence that an agent has that the use of this link will contribute to achieving successful communication. It can in this respect be compared to the score of a construction (see 3.4.5). The weights range from 0 to 1, with 0 indicating that the link is fully entrenched and 1 indicating that the use of this link will most likely not lead to communicative success. In evolutionary experiments, new links that are added to the type hierarchy are initialised with a weight of 0.5, and the weights are updated after each communicative interaction, based on its outcome. The updating of the weights of the type hierarchy links plays a crucial role in learning and aligning the type hierarchies of the agents in a population, as will be shown in chapter 7.

Figure 4.8 shows an example of a type hierarchy in which the links are weighted. The

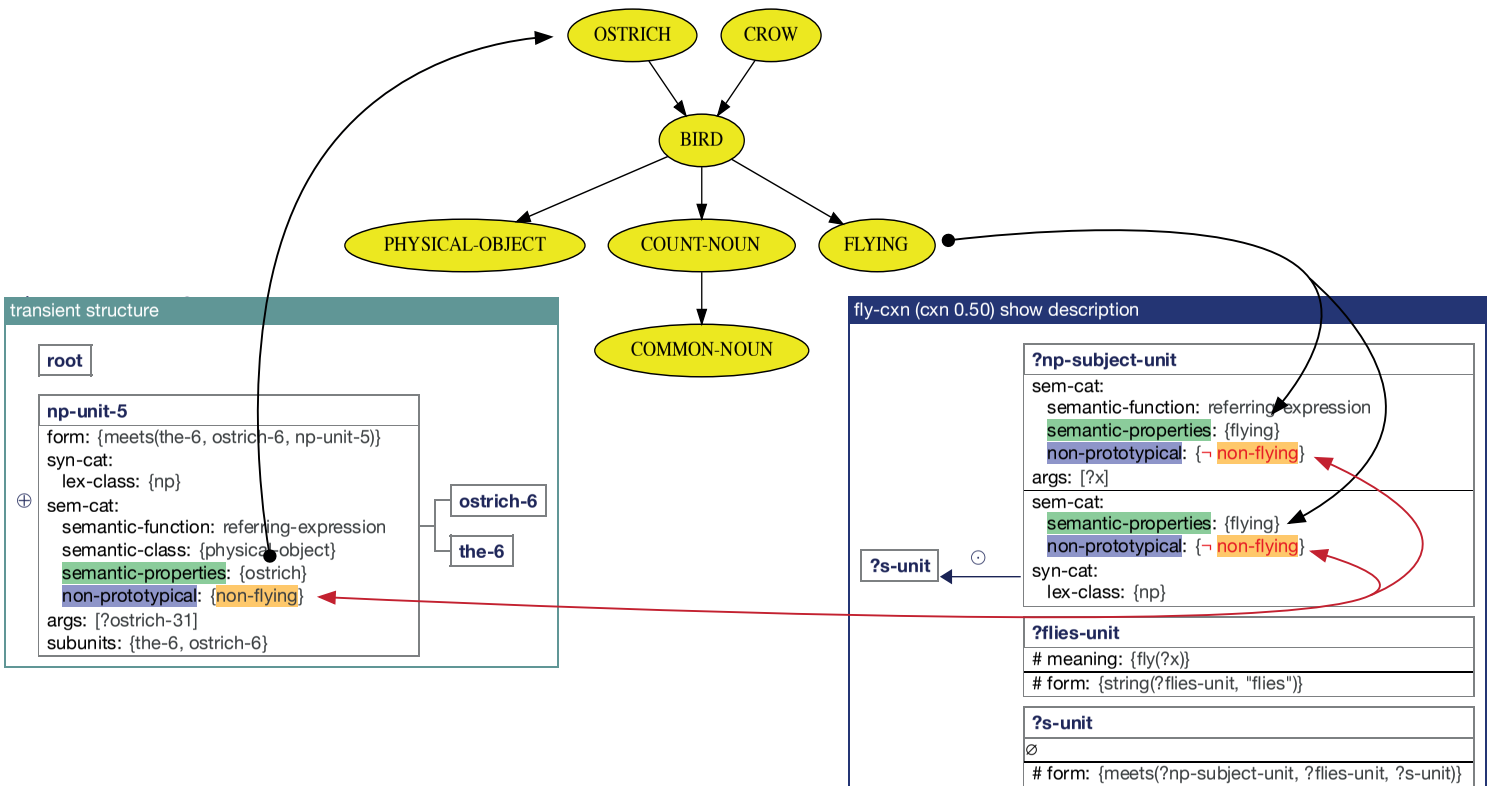


Figure 4.6: Cancellation of generalisations through information local to the construction. The construction cannot apply. Matching fails because of the non-prototypical feature, as highlighted in red.

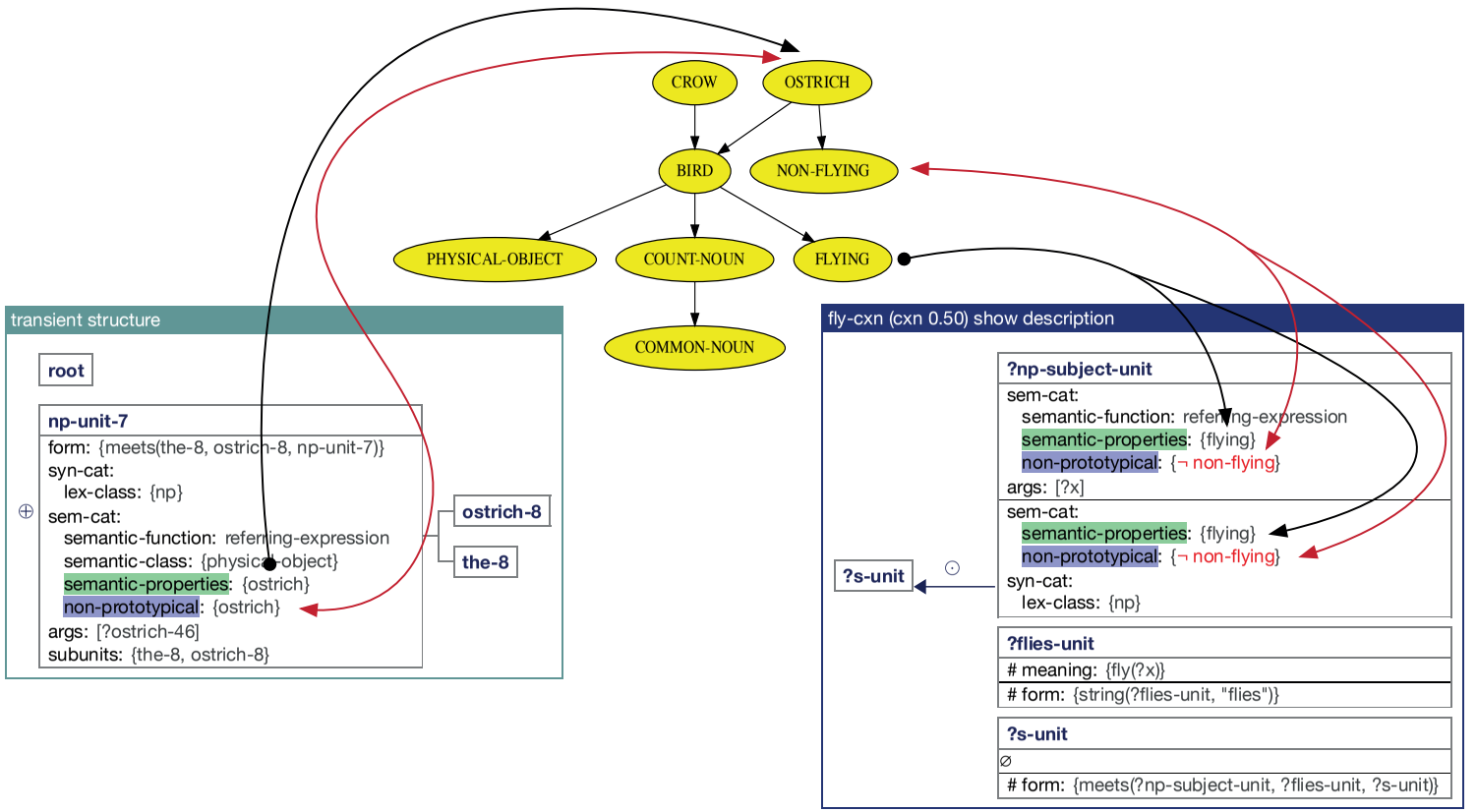


Figure 4.7: Cancellation of generalisations through information in the type hierarchy. The construction cannot apply. Matching fails because of the non-prototypical feature, of which the link through the hierarchy is highlighted in red.

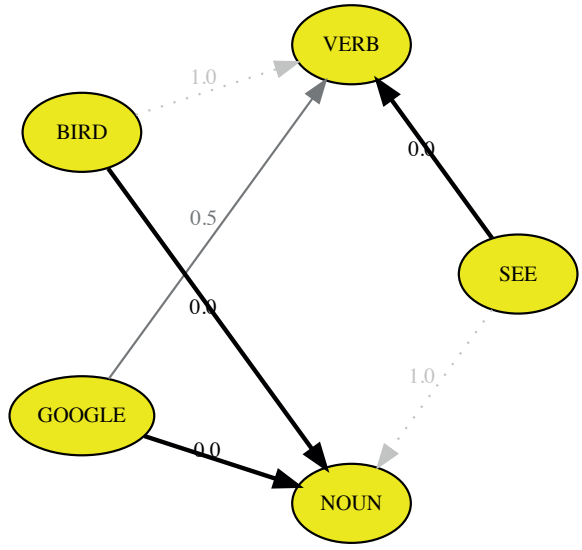


Figure 4.8: A type hierarchy showing the weights on the links that reflect their entrenchment in the grammar. More entrenched links are shown in a darker color and minimally entrenched links are visualised using dotted arrows.

links from 'bird' and 'google' to 'noun' are fully entrenched as indicated by their weight of 0.0, and the same holds for the link from 'see' to 'verb'. The links from 'bird' to 'verb' and from 'see' to 'noun' have a weight of 1.0, indicating that the use of these links is unlikely to lead to communicative success. Finally, the link from 'google' to 'verb' has just been created (like in 'he googled him'), and has a score of 0.5. The more entrenched a link is, the darker it is visualised, and minimally entrenched links (with score 1.0) are visualised using dotted arrows.

4.5 Type Hierarchies versus Typed Feature Structures

Throughout this chapter, I have used the term 'type hierarchy' to refer to my implementation of a system for handling super- and subcategories in FCG. This use of the term 'type hierarchy' deviates from its typical denotation when used in the context of unification grammars. There, the term is usually used in relation to *typed feature structure grammars* (Smolka and Ait-Kaci, 1989; Moens et al., 1989; Copestake, 2002; Carpenter, 2005), where it denotes the data structure indicating the specialisation and consistency of types. In that context, it is also sometimes referred to as *sort hierarchy*, *type lattice* or *sort lattice*. The next paragraphs will focus on the key differences

between FCG's implementation of type hierarchies and typed feature structures.

Typing of feature structures As suggested by its name, each feature structure in a typed feature structure grammar is typed with a particular label. A type definition system specifies for each type which features it allows, and for each of these features, of which type its value should be. The type hierarchy defines the compatibilities (in terms of unification) between the different types that are used. FCG's type hierarchy system does not type feature structures and, on purpose, does not specify which features can appear in the value of another feature. The collection of features that is used, and of possible values is open-ended and can be dynamically changed.

Closed World Assumption Typed feature structure grammars and their implementations usually assume a closed world, in the sense that all types that exist in a certain grammar have a specified place in the type hierarchy (Pollard, 1997; Copestake, 2002). A closed world assumption allows making additional inferences about types in the grammar. For example, if the type hierarchy specifies that 'plural' and 'singular' are subtypes of 'number' and you know that a certain value is of type 'number' and is not 'singular', it can be inferred that it is plural. In FCG's type hierarchy system, there is no closed world assumption, and no appropriateness restrictions for symbolic values. As a consequence, no inferences based on a closed world should be made.

4.6 Conclusion

In this chapter, I have introduced a type hierarchy system for Fluid Construction Grammar. The system aims to combine the main advantages of typed feature structures, in particular the possibility to capture hierarchical relations among categories in the grammar, with the advantages of untyped feature structures, in particular the dynamic, constructivist and open-ended nature of the grammar. This was achieved by adding a type hierarchy graph to the grammar, in which hierarchical relations between symbols can be stored. FCG's standard matching and merging algorithms were adapted, such that they also succeed when two symbols are connected through the type hierarchy, with the symbol in the construction being a supertype of the symbol in the transient structure. Default matching and merging behaviour is kept for any symbols that do not occur in, or are not connected through, the type hierarchy.

The type hierarchy system allows explicitly capturing generalisations over categories in the grammar, and exploiting these generalisations when learning new constructions.

For the case where a hierarchical relation between two categories is prototypical rather than absolute, and exceptions need to be handled, two mechanisms for cancelling generalisations were presented. The first mechanism included non-prototypical categories in the construction, whereas the second mechanism included the non-prototypical categories into the type hierarchy.

Finally, I have pointed out some key differences with typed feature structure grammars, especially in the context of open-endedness and dynamicity. How these type hierarchies can be learned from individual observations will be addressed in the next chapters.

Chapter 5

Generalising Constructions using Anti-Unification

5.1 Introduction

FCG implements language processing as a search process, in which the operators, in this case constructions, are applied to state representations, in this case transient structures, until a solution is found (see Figure 3.1 of Chapter 3). For the system to return a solution, it is necessary that somewhere in the search space that is created by all possible sequences of construction applications, there exists a solution state. In other terms, a solution can only be found if the exact constructions that are necessary for reaching a solution state are available in the construction inventory.

This will often not be the case when processing natural language, which is inherently creative, dynamic and open-ended. Human speakers often use language elements in a way that deviates from the way in which they are normally used. Consider for example, the French utterance in (1), found in a discussion forum on the internet[1].

(1) Tu es love d' elle, elle le sait pas, [...]
`You.sg are.sg love of her, she it knows not,` [...]
You are in love with her, she doesn't know it, [...]

In this utterance, the word 'love' is used in a very unconventional way. 'Love' is an English noun or verb, which is not commonly used in French. Semantically, the utterance uses the word 'love' in the same way as it is used in the English phrase 'to be in love with someone'. Grammatically however, 'love' fills the adjective slot in the 'être ADJ de NOUN'-cxn (*'to be ADJ of NOUN'-cxn*) and not the noun slot in e.g. 'tu es en love avec elle' (*you are in love with her*) or the verb slot in e.g. 'tu la loves' (*you love her*). The 'être ADJ de NOUN'-cxn was probably preferred by the French speaker because of the influence of the alternative constructions 'être amoureux de' (*to be in love with*) and 'être fou de' (*to be crazy about*), which are very common in French. The use of 'love' in the adjective slot is not limited to this construction, as it also occurs in for example the 'tomber love de'-cxn (*'fall in love with'-cxn*).

Although the utterance in (1) might sound a bit unusual when encountered for the first time, the French listener has no problem at all comprehending it. His language processing system is robust enough to deal with the unexpected violation of usual constraints, here in particular the use of the word 'love' as an adjective. In FCG terms, the speaker and listener flexibly apply the relevant constructions of their construction inventory, relaxing a constraint on grammatical category. The flexible application of constructions is crucial for computational construction grammar, both when used for

[1] `http://www.jeuxvideo.com/forums/1-51-64360220-1-0-1-0-j-ai-like-sans-le-faire-expres-la-photo.htm`. Consulted on 27/07/2017.

processing real-world data and when used in evolutionary experiments. When processing real-world data, the input utterances will always feature new ways of using words, or grammatical structures that were not foreseen in the grammar. When used in evolutionary experiments, speakers often need to innovate their language, and listeners need to be able to comprehend and adopt these innovations. The processing of unexpected input requires the possibility to flexibly apply the constructions of the grammar, relaxing features that would otherwise block construction application.

In this chapter, I introduce concrete mechanisms for achieving flexible construction application in FCG. The chapter is structured as follows. I will first discuss previous work on resolving grammatical impasses in FCG and specify my own contributions (5.2). Then, I will introduce the concept of anti-unification (5.3), discuss its relation to unification (5.3.1) and present a basic anti-unification algorithm (5.3.2). Then, I will present the innovative machinery for anti-unification-based flexible construction application that I have developed for FCG (5.4). After that, I will demonstrate with a variety of examples the kinds of generalisations that the algorithm makes (5.5). Finally, I will show how the algorithms can be combined with the type hierarchy system introduced in the previous chapter (5.6) and how the anti-unification machinery can used as a debugging tool in grammar engineering (5.7).

5.2 Resolving Grammatical Impasses

There has been a large body of previous work on resolving grammatical impasses in FCG, as they are at the core of the evolutionary linguistics experiments that the formalism was originally designed for (Steels, 2012b). A grammatical impasse occurs when the grammar of an agent does not contain the constructions that are necessary to provide a satisfactory processing result, i.e. an adequate meaning representation in comprehension or an adequate utterance in formulation. In FCG, grammatical impasses are typically dealt with using a meta-layer architecture (see Section 3.5, as well as Steels and van Trijp (2011); Steels (2012d); Beuls et al. (2012); Van Eecke and Beuls (2017)), in a process that involves the following steps:

- **Diagnosis** The system needs to be able to detect and identify grammatical impasses during processing.
- **Repair** The system needs to be able to come up with a solution to the problems that were diagnosed.
- **Consolidation** The system needs to store the solution to the problem in a way that allows it to be reused for solving similar problems in the future.

Grammatical impasses can be diagnosed either system-internally or with reference to a shared semantic context. In comprehension, system-internal diagnostics typically check whether the resulting meaning network forms one integrated chunk, and whether there are any strings left in the 'root' unit, which would mean that not all words in the input utterance have been 'consumed' by morphological or lexical constructions. System-external diagnostics typically check whether the comprehended meaning can be interpreted in the world without leading to any inconsistencies. In formulation, system-internal diagnostics check whether comprehending the formulated utterance (re-entrance (Steels, 2003)) yields the original input meaning network and system-external diagnostics can check whether the interpretation in the world of the meaning network that is obtained through re-entrance yields the same result as the interpretation of the original meaning network.

Just like diagnostics, repairs also function either system-internally or with reference to a shared semantic context. The previous literature has focussed on resolving grammatical impasses with reference to a shared context, and in particular on the sophisticated semantic processing that is needed to achieve this (Beuls and Steels, 2013; Spranger, 2016; Garcia Casademont and Steels, 2016).

In this chapter, I introduce a complementary repair strategy that does not require access to a shared semantic context. The strategy is particularly useful in cases of displaced communication (e.g. telephone conversations), in cases where the participants lack a substantial amount of prior common ground, or when the application of the existing construction inventory does not lead to a plausible partial meaning hypothesis, and therefore does not provide a semantic interpretation that a repair can work with (Steels and Van Eecke, 2018).

The proposed strategy is based on flexible construction application. In standard FCG, construction application (match and merge) either succeeds or fails. Even if a single feature or value from the construction cannot be matched or merged with the transient structure, the construction cannot apply. Using the flexible construction application strategy introduced in this chapter, the units, features or values that block construction application are relaxed, such that a construction can always apply. For controlling construction application, a specific cost is assigned to the relaxation of different kinds of constraints. The construction that can apply with the lowest cost, corresponding to the smallest number of constraint relaxations that was needed, will be preferred.

5.3 Anti-Unification

The machinery for flexible construction application that I have developed and integrated into FCG is based on *anti-unification*, more in particular *first order syntactical anti-unification* (Plotkin, 1970, 1971; Reynolds, 1970). Anti-unification is an operation that computes the opposite of unification. Whereas unification computes the *most general specialisation (MGS)* of two or more terms, anti-unification computes the *least general generalisation (LGG)* of two or more terms. The power of anti-unification to generalise over structures has been applied in many domains, including machine learning (Flach, 2012), inductive reasoning (Feng and Muggleton, 2014), information extraction (Thomas, 1999), case-based reasoning (Armengol and Plaza, 2012), metaphor and analogy detection (Gust et al., 2006), duplicate code detection (Bulychev and Minea, 2008) and generalisation over linguistic parse trees (Galitsky et al., 2011) and thickets (Galitsky et al., 2014). In this dissertation, I apply the generalising power of anti-unification to constructional language processing.

I will first introduce the concept of anti-unification and its relation to unification (5.3.1) and then present a basic algorithm for the anti-unfication of two feature structures (5.3.2).

5.3.1 Anti-Unification vs. Unification

Anti-unification is a rather intuitive concept for those familiar with unification. In this section, we will have a look at two examples that show the differences between unification and anti-unification while introducing the necessary vocabulary for the more detailed discussion in the rest of this chapter. The terms that are used in the examples are taken from Flach (1994)'s discussion on inductive reasoning and anti-unification (p. 178).

Figure 5.1 demonstrates the *unification* operation, which computes the MGS of two terms, called the *pattern* and the *source*. The pattern and the source consist of variables, which start with a question mark, and/or constants, which don't start with a question mark. The unification operation computes a minimal set of *bindings* that, after substitution, makes the two terms equal. In the example in Figure 5.1, the two terms '$2 \cdot ?x = ?x + ?x$' and '$?y \cdot 3 = ?x + ?x$' can be made equal to each other by binding ?x to 3 and ?y to 2. The set of bindings is notated as ((?x . 3) (?y . 2)). The *resulting pattern*, which is the most general specialisation of the two terms, is computed by substituting the set of bindings obtained through unification in one of the terms. In this case, the resulting pattern is '$2 \cdot 3 = 3 + 3$'. If there exists no

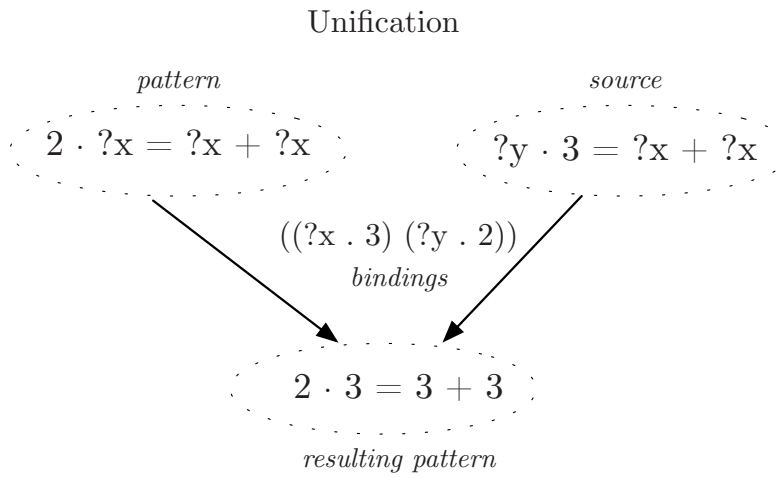


Figure 5.1: A schematic representation of the unification of a pattern and a source, yielding a bindings list and a resulting pattern.

bindings list that makes the two terms equal, unification fails.

Figure 5.2 demonstrates the *anti-unification* operation, which computes the LGG of two terms, called the *pattern* and the *source*. The pattern and the source consist of constants and/or variables. The anti-unification operation computes two sets of bindings, called the *pattern-bindings* and the *source-bindings*, as well as a *resulting pattern* that is the least general generalisation of the two terms. In the example in Figure 5.2, the pattern is '2 · 2 = 2 + 2' and the source is '2 · 3 = 3 + 3'. The resulting pattern, and thus the least general generalisation of the two terms, is '2 · ?x = ?x + ?x'. The pattern-bindings are ((2 . ?x)) and the source-bindings are ((3 . ?x)). Note that unlike in the unification case, the resulting pattern cannot be reliably computed through substitution of the bindings. This is because not every instance of a symbol or variable occurring in the bindings necessarily needs to be substituted. In the pattern in Figure 5.2 for example, only 3 of the 4 occurrences of 2 need to be substituted by ?x to yield the least general generalisation. As a consequence, an anti-unification algorithm always needs to compute the two sets of bindings and the resulting pattern, whereas a unification algorithm only needs to compute a single set of bindings. As opposed to unification, anti-unification always succeeds. In the extreme case, the resulting pattern is then a single variable that is bound to the pattern in the pattern bindings and to the source in the source bindings. The resulting pattern is also guaranteed to unify with both terms.

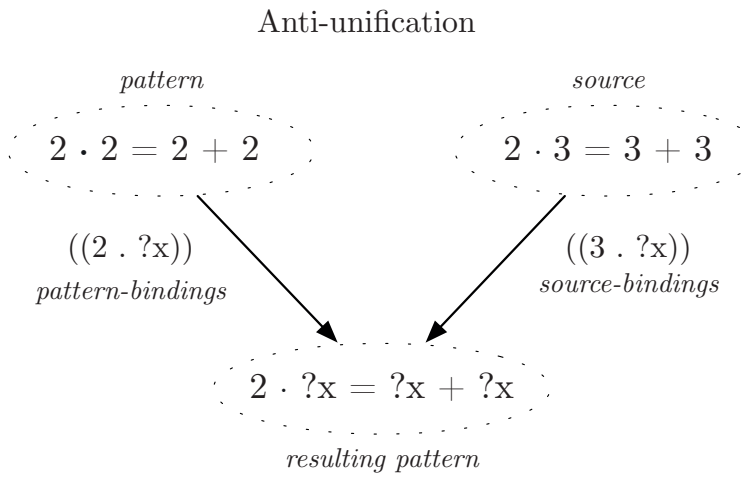


Figure 5.2: A schematic representation of the anti-unification of a pattern and a source, yielding a resulting pattern and two bindings lists.

5.3.2 A Basic Anti-Unification Algorithm

Implementing a basic algorithm that anti-unifies a pattern with a source is a rather straightforward task, and many can be found in the literature, for example in Flach (1994, p. 177). The algorithm synchronously goes through the pattern and source and compares the terms that occur in the same position. If the terms are equal, the algorithm proceeds to the next term. If the terms are different, they are substituted by a new variable and the algorithm proceeds to the next term. The substitutions are collected into two lists, one for the pattern-bindings and one for the source-bindings. If the same two terms occur a second time in the pattern and the source, they are substituted by the same variable as the first time (hence the necessity of storing the bindings). An implementation of this algorithm is shown in Figure 5.3. In order to be readily runnable and as concrete as possible, the algorithm is given in Common Lisp code (the language is which FCG is implemented), rather than pseudo-code. The first function does the actual anti-unification, the second function takes care of the recursive nature of feature structures. As an illustration, a few inputs and outputs computed by this algorithm are shown in Table 5.1. The first line shows an anti-unification result where no generalisations were needed. The second, third and fourth line show substitutions by new variables. The fifth line shows the reuse of substitutions. The sixth line gives an example of variable decoupling and the last line shows an example of the reuse of substitutions in nested structures.

```
(defun anti-unify (pattern source &optional
                           (pattern-bindings +no-bindings+)
                           (source-bindings +no-bindings+))
  ;; Anti-unifies pattern with source. Returns resulting-pattern,
  ;; pattern-bindings and source-bindings.
 (cond ((equalp pattern source) ;; Case 1 : Pattern equals source.
        (values pattern pattern-bindings source-bindings))
       ;; Case 2: Reuse bindings from previous substitution.
       ((subs-lookup pattern-bindings source-bindings pattern source
          )
        (values (subs-lookup pattern-bindings source-bindings
            pattern source)
                pattern-bindings source-bindings))
       ;; Case 3: Anti-unify subterms as sequence.
       ((and (not (variable-p pattern)) (not (variable-p source))
             (listp pattern) (listp source)
             (equalp (feature-name source) (feature-name pattern))
             (anti-unify-sequence pattern source '()
                                  pattern-bindings source-bindings))
        (multiple-value-bind (resulting-pattern
            resulting-pattern-bindings resulting-source-bindings)
            (anti-unify-sequence pattern source '()
                                 pattern-bindings source-bindings)
          (values resulting-pattern resulting-pattern-bindings
                  resulting-source-bindings)))
       ;; Case 4: Replace terms by new variable, and extend bindings
          .
       (t (let ((var (make-var)))
            (values var
                    (extend-bindings pattern var pattern-bindings)
                    (extend-bindings source var source-bindings))))))

(defun anti-unify-sequence (pattern source accumulator
                                   pattern-bindings source-bindings)
  ;; Anti-unifies a sequence of subterms
 (cond ;; Case 1: Base case, return accumulator.
    ((and (null pattern) (null source))
     (values accumulator pattern-bindings source-bindings))
    ;; Case 2: anti-unify the first term, and anti-unify the other
    ;; terms as a sequence.
    (t (multiple-value-bind (resulting-pattern
                resulting-pattern-bindings
                    resulting-source-bindings)
          (anti-unify (first pattern) (first source)
                      pattern-bindings source-bindings)
        (anti-unify-sequence (rest pattern) (rest source)
                             (pushend resulting-pattern accumulator
                                 )
                             resulting-pattern-bindings
                             resulting-source-bindings)))))
```

Figure 5.3: An implementation of a basic anti-unification algorithm.

Table 5.1: Inputs fed to and outputs computed by the anti-unification algorithm in Figure 5.3.

Pattern	Source	Result. Patt.	Comment
(a b c)	(a b c)	(a b c)	Pattern equals source.
(a b c)	(a y c)	(a ?v-1 c)	Substitution by new variable.
(a b c d)	(a y c z)	(a ?v-1 c ?v-2)	2 substitutions by new variables.
(a b ?v-1)	(a b ?v-2)	(a b ?v-3)	Substitution by new variable.
(a b c b)	(a y c y)	(a ?v-1 c ?v-1)	Reuse of substitution.
(a ?v-1 c ?v-1)	(a b c d)	(a ?v-2 c ?v-3)	Variable Decoupling
(a b (c (a b)))	(a y (c (a y)))	(a ?v-1 (c (a ?v-1)))	Nesting + Reuse

5.4 Anti-Unification for FCG Structures

In the previous section, I have introduced the concept of anti-unification and presented a basic anti-unification algorithm. In this section, I take up a more challenging task, namely the development of an anti-unification algorithm for FCG structures. Concretely, this algorithm gets as input a transient structure and a construction that does not match the transient structure. The algorithm should return a construction that is the least general generalisation of the construction with respect to the transient structure and thus matches this transient structure. I will respectively discuss the concepts of pattern and source (5.4.1), the calculation of an anti-unification cost (5.4.2), the problem of unit pairing (5.4.3) and the anti-unification of particular feature types (5.4.4).

5.4.1 About Pattern and Source

When introducing anti-unification in the previous section, I have used the words *pattern* and *source* for designating the terms that were to be anti-unified. It might have seemed a bit odd to use different names for the two terms, as the anti-unification operation that I have described is a commutative operation. By using pattern and source, I was already anticipating the application of the algorithm to FCG. In FCG, the operations involved in construction application are not commutative. The match operation includes a subset constraint. All units, features and values of the construction should unify with those in the transient structure, but the transient structure can contain additional units, features and values. Constructions can also include special operators (see section 3.4.3), whereas this is not possible in the case of transient structures.

As a consequence, the dual of the match operation, *anti-unify-match*, is not commutative either. The first argument to anti-unify-match, called the *pattern*, is always (a part

of) a construction, the second argument, the *source*, is always a transient structure. The return value, called the *resulting-pattern*, which is the least general generalisation of the pattern, is also (a part of a) construction.

The part of the construction that needs to be anti-unified depends on the direction of construction application. In comprehension, it is the collection of comprehension locks and in formulation the collection of formulation locks. As the mechanisms are exactly the same in comprehension and formulation, I will abstract away from the specific direction and call the collection of relevant locks the *matching-pattern*.

The concrete task of *anti-unify-match* is to operate on the *matching-pattern* of a construction in a given direction and a *transient structure*, and to return a *resulting-pattern*, which is the least general generalisation of the matching pattern that matches the transient structure. After substitution of the matching-pattern in the construction[2], the construction is guaranteed to match the transient structure.

5.4.2 Integrating Cost Calculation

As mentioned above, anti-unification never fails, but it might have to make generalisations that are so general that they become virtually meaningless. As a consequence, it is a highly non-trivial problem to decide which construction to anti-unify with and apply to a transient structure. In order to tackle this problem, a cost calculation system was integrated into the anti-unification algorithm. The idea is that, with each anti-unification result, the algorithm returns a cost, which reflects inhowfar the construction needed to be generalised before it could apply to the transient structure. The construction that required the least amount of generalisation will then be preferred by a repair. The cost calculation system can be tuned to the needs of the user by assigning different costs or cost functions to different types of generalisations.

The cost calculation parameters are passed to the anti-unify-match function with the keyword `:cost-params`. The parameters are formatted as a list of lists. The first element of each sublist is the label designating the type of generalisation. The rest of the list consists of either one element, namely an integer representing the cost of the generalisation, or two elements, namely the name of a cost function that will be called and a parameter that will be passed to this function. The types of generalisations that are supported and their default values are the following:

[2] In order to preserve all variable bindings in the construction that do not cause conflicts (i.e. between locks or between the matching pattern and the contributing part), the variable substitutions that were used when computing the resulting-pattern should also be applied to the other parts of the construction.

- `equality 0`: The pattern and the source are equal to each other. There is no need for generalisation, so there is no associated cost.

- `non-matching-unit 10`: A unit from the pattern does not match with any unit from the source. It will need to be paired (see 5.4.3) and generalised (5.4.4).

- `subst-from-bindingslist 0`: The pattern and source are different, but they already have a common binding in the bindings list. They will be substituted by this binding, but there is no additional cost.

- `source-variable-pattern-in-bindingslist 1`: The pattern has already has a binding in the bindings list, the source is a variable. The pattern will be substituted by its binding, with a small cost.

- `replace-by-new-var depth-of-pattern 1`: An atomic or complex value in the pattern needs to be generalised. The cost is equal to 1 + the depth of the pattern. E.g. for 'singular', the cost will be 1, for '(TAM: (tense: (present: +))', the cost will be 3.

- `discarded-feature 5`: A feature from the pattern is not found in the source. The feature is discarded in the resulting pattern.

- `discarded-negated-feature 4`: A negated feature from the pattern is found in the source. The negated feature is discarded in the resulting pattern.

- `removed-pattern-unit 20`: A unit from the pattern is removed. This can for example occur e.g. when an 'Art+Adj+Noun'-cxn is generalised to an 'Art+Noun'-cxn.

The costs are cumulative. For example, if a unit of the pattern is paired with a unit in the source with which it does not match, this will add a cost of 10 to the total cost of the process. But then, this unit of the pattern will still need to be anti-unified with the features of the unit of the source it is paired with. If for example one feature needs to be discarded (+5) and one value needs to be replaced by a variable (+1) the total cost for this unit will be 16 (10+5+1). Another possibility would be to remove the unit from the pattern. This would however be more costly, at a cost of 20, thus the pairing option will be preferred.

The default values of the cost parameters are set for demonstration purposes. Optimal values are best experimentally determined and will depend on the specific grammar that is used and on the type of input that is processed.

5.4.3 Pairing Units

The first step in the anti-unification process is to decide which units in the source will be anti-unified with which units in the transient structure. Concretely, each unit in the pattern needs to be paired with exactly one unit in the transient structure. The pairing of units is a non-trivial problem, as the unit names in the pattern are variables [3] (see Section 3.4) and can therefore not be used for guiding the pairing. Moreover, the transient structure most often consists of many more units than the construction. Overall, there exist three possible pairing strategies for each unit in the pattern:

- If the unit matches a unit in the transient structure, these two units will be paired with no cost. Additionally, the unit in the transient structure cannot be paired any more with any other unit in the pattern.

- If the unit does not match any unit in the transient structure, it will be paired with a unit in the transient structure that does not match one of the other units in the pattern. This will most often lead to multiple alignment options, which can be explored in parallel. For each unit that needs to be paired in this way, the `non-matching-unit` cost is added to the total anti-unification cost.

- Alternatively, if the unit does not match any unit in the transient structure, the unit can also be deleted from the pattern (*unit-deletion*). The pattern will then have one unit less and the `discarded-unit-cost` will be added to the total anti-unification cost.

The unit pairing process takes a matching pattern and a transient structure as input, and its output is a list of unit alignment options. For each alignment option, each unit in the pattern is either aligned with exactly one unit from the transient structure, or discarded from the pattern. A schematic representation of how the unit pairing process works is presented in Figure 5.4. The matching pattern of the construction consists of three units (?unit-X, ?unit-Y and ?unit-Z) and is shown in the top-left corner. The transient structure consists of 4 units (unit-1, unit-2, unit-3 and unit-4) and is shown in the top-right corner. Blue lines connect the units that match each other ((?unit-X . unit-4) and (?unit-Y . unit-2). Red lines indicate that ?unit-Y finds no matching units in the transient structure. The bottom part of the Figure shows the three alignment options that are possible. The first two options pair ?unit-Y from the construction with a unit of from the transient structure that is at that moment still unpaired. Option 1 aligns ?unit-Y with unit-1 and option 2 aligns ?unit-Y with unit-3. Alignment 3 on the other hand, discards ?unit-Y from the construction (unit-deletion).

[3] Except for the root unit, for which, by consequence, the unit pairing problem is trivial.

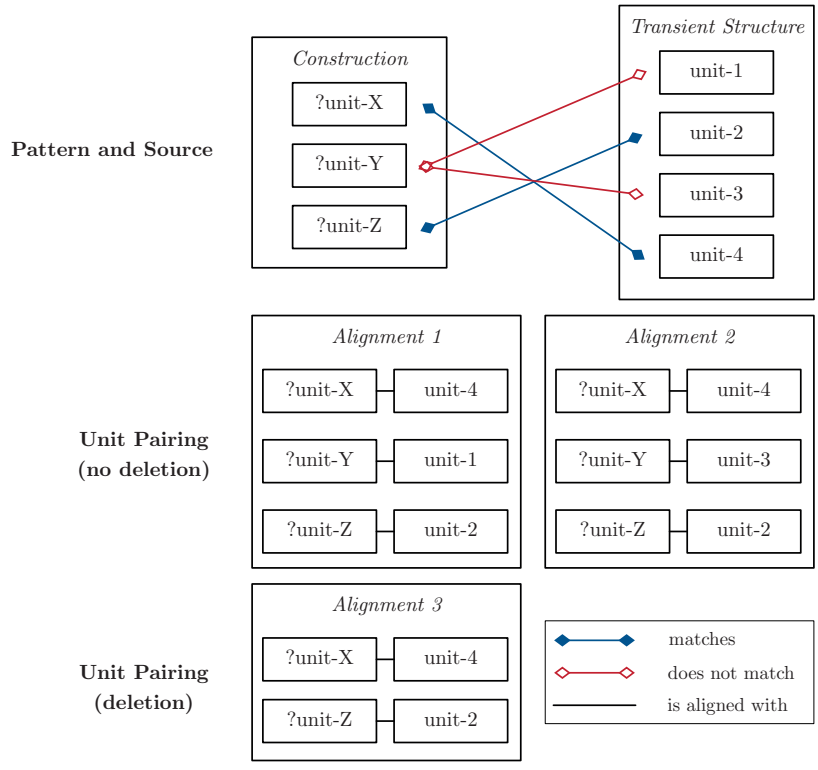


Figure 5.4: A schematic representation of the unit pairing process. Each of the three units in the construction needs to be paired with one unit in the transient structure. Two units can be paired with matching units. The third unit has two pairing possibilities with the two left-over units in the transient structure or can be discarded from the construction. The three alignment options are shown at the bottom.

The unit pairing process might seem a costly matter when applied to larger constructions and transient structures. When multiple units from the construction find no matching unit in a transient structure that contains many units, a combinatorial explosion is not far away. While this is certainly true, it is not necessarily problematic. There are very few grammatical impasses for which more than 2 units need to be paired. For impasses caused by agreement problems (e.g. gender, case or number), all individual units can most often be matched, as the problem resides in variable bindings between units. For impasses caused by category mismatches (e.g. coercions), only a single unit needs to be paired with a non-matching unit. For impasses caused by missing words or markers, most often only a single unit needs to be discarded. If more than 2 units need to be discarded or paired with non-matching units, which is reflected by a high cost, it is probably a better idea to stop the anti-unification process for that construction and try to anti-unify a different construction from the construction inventory. In general, constructions that have been generalised too far carry no meaning any more and might even do more harm than good.

5.4.4 Anti-Unifying Features and Values

The unit pairing process returns a pattern and a source consisting of the same number of units, in the aligned order. Next, the features in these units need to be anti-unified. Although the algorithm that performs the anti-unification is conceptually close to the basic anti-unification algorithm presented in Section 5.3.2 and Figure 5.3, there are quite a few additional difficulties. These are mainly due to the feature type system and special operators that are used in FCG constructions.

As explained in Section 3.4.3, FCG constructions use feature types and special operators for influencing the behaviour of the unification algorithm, for example for subset unification, negation and overwriting. In order to keep the anti-unified constructions consistent with their original counterparts and in order to appropriately compare and generalise features and values, the anti-unification algorithm also needs to take into account the meaning of these feature types and special operators. This is done in the following way:

- **Unit Names** Unit names in the pattern are treated just like any other variables. As the unit names in the pattern will never be equal to those in the source, they are always substituted. If the unit name already occurs in the substitutions list, it will be replaced by its binding there, otherwise, it will be replaced by a free variable and added to the substitutions list.
- **Top-level Features** Top-level features are the features that depend directly from

the unit, i.e. they are not part of the value of another feature. The top-level features of a unit are treated as a set. This means that for each top-level feature in a unit of the pattern, the anti-unification algorithm will search for a top-level feature in the paired unit of the source, with the same feature name. If such a feature is found, their values are anti-unified. If no such feature is found, the feature is discarded from the pattern.

- **Default Feature Type** For the default feature type, we distinguish between atomic values and complex values. If the value of a feature of the default type is atomic, it is compared to the value of this feature in the source. If the two are equal, no action is taken. If they are not equal, the atom from the pattern is replaced by its binding in the substitutions list (if applicable) or by a free variable. In the case of a complex value, its features are treated as a set. Each feature from the pattern is searched in the source. If it is found, their values are anti-unified. If not, the feature is discarded from the pattern.

- **Set** The value of a feature of the type set is always a set of atoms, otherwise the default feature type should be used. Each atom is searched in the value of this feature in the source. If it is found, no action is taken. If it is not found, it is replaced by its binding in the substitutions list (if applicable), or by a free variable.

- **Sequence** The value of a feature of the type sequence can either be a sequence of atoms are a sequence of feature-value pairs. In both cases, the elements (atoms or feature-value pairs) from the pattern and the source are anti-unified in the order in which they occur. If an element does not occur in the source, it is replaced by its binding in the substitutions list (if applicable), or by a free variable.

- **Set-of-Predicates** On anti-unification level, a set-of predicates is treated exactly like a complex value of the default feature type. If a predicate with the same predicate name is found in the source, they are anti-unified. If not, the predicate is discarded from the pattern.

- **Sequence-of-Predicates** A sequence of predicates is treated like any other sequence. The predicates are anti-unified in the other in which they occur and replaced by their binding in the substitutions list (if applicable) or a free variable.

- **#-operator** So-called 'hashed' features are searched in the 'root' unit of the transient structure, instead of in the unit in which they occur in the construction. In general, only top-level features can safely be hashed. Generalisation of these features happens according to the description under top-level features.

- **Negation Operator** Features or values that are preceded by the negation operator, which is notated as 'NOT', are treated as an element of a set in the transient structure. If the feature or value is not found, no action is taken. If the feature or value is found, it is added to the list of discarded features.
- **Overwriting Operator** The part written in front of the operator is anti-unified with whatever occurs at the same place in the transient structure.

When the anti-unification machinery is used in combination with the type hierarchy system introduced in chapter 4, the algorithm compares symbols not only in terms of equality, but also in terms of super- and subtypes in the type hierarchy. Note that the anti-unification algorithm can also be used for building type hierarchies. This topic will be discussed in Section 5.6 below.

5.5 Demonstration

In this section, I present a selection of examples that illustrate the use of the anti-unification algorithm in FCG and demonstrate the kinds of generalisation that it makes. All examples feature simple French noun phrases. The grammar contains lexical constructions that can apply to each word in the input and a noun phrase construction that tries to combine the lexical items into a noun phrase. In each example, the noun phrase construction cannot apply for a different reason and a different kind of generalisation is made by the anti-unification algorithm: variable decoupling, value relaxation, feature/predicate deletion or unit deletion. A more concise discussion of some of these examples has already been published by Steels and Van Eecke (2018). In order to explore the examples in full detail, I highly recommend the reader to visit the interactive web demonstration at `https://www.fcg-net.org/demos/vaneecke-phd/anti-unification`.

The examples are chosen for their didactic nature and, for space reasons, only the relevant features are shown in the printed figures. More elaborate examples in which the anti-unification algorithm is embedded in the meta-layer architecture, and the system needs to decide which construction of the construction inventory to apply, will be discussed in Chapter 6.

5.5.1 Variable Decoupling

The first kind of generalisation that the anti-unification algorithm makes is called *variable decoupling*. Variable decoupling happens when the same variable occurs at multiple

places in the construction and different, non-unifying values occur at these places in the transient structure, making unification fail. Anti-unification solves the problem by 'decoupling the variables', i.e. replacing the occurrences of the variable that cause conflicts with new variables. Variable decoupling is often used to solve agreement problems, e.g. gender agreement among the elements of the noun phrase or number agreement between subject and verb. I will illustrate variable decoupling with two examples. In the first one, it is used to solve a gender mismatch, and in the second one to handle a deviant word order.

Gender Mismatch

Consider a noun phrase construction (np-cxn) that combines an article, an adjective and a noun into a noun phrase. One of the constraints in the np-cxn is that the article, adjective and noun need to have the same number and gender. Consider now an input utterance "une petit fille" (*a small girl*), in which "une" (*a*) is a feminine singular article, "petit" (*small*) is a masculine singular adjective and "fille" (*girl*) is a feminine singular noun. While the lexical class (article - adjective - noun) and number (sg, sg, sg) of the three words are compatible with the construction, the gender (f, m, f) is not and the construction cannot apply.

Figure 5.5 shows the original np-cxn in the top-left corner, with only the '?noun' and '?adj' units expanded. Gender and number of these two units need to be the same, as required by the shared variables '?gender' and '?number' in the construction. The transient structure on the left of the figure shows that the gender of the noun is 'f' and the gender of the adjective 'm', blocking construction application. This is highlighted by red, dotted arrows. In the anti-unified construction in the bottom right corner, the values of the gender feature are now two different variables ('?gender-5488' and '?gender-5424'). After decoupling the variables, the construction can be applied without further action, as highlighted by green arrows. Note that the number agreement is unaffected in the anti-unified construction (variable '?number-3812'), as there were no conflicts there.

Deviating Word Order

This example demonstrates how the anti-unification algorithm uses variable decoupling for handling a word order problem. Consider an np-cxn that combines an article, an adjective and a noun into a noun phrase. The construction specifies that the article should be immediately left-adjacent to the adjective and that the adjective should be

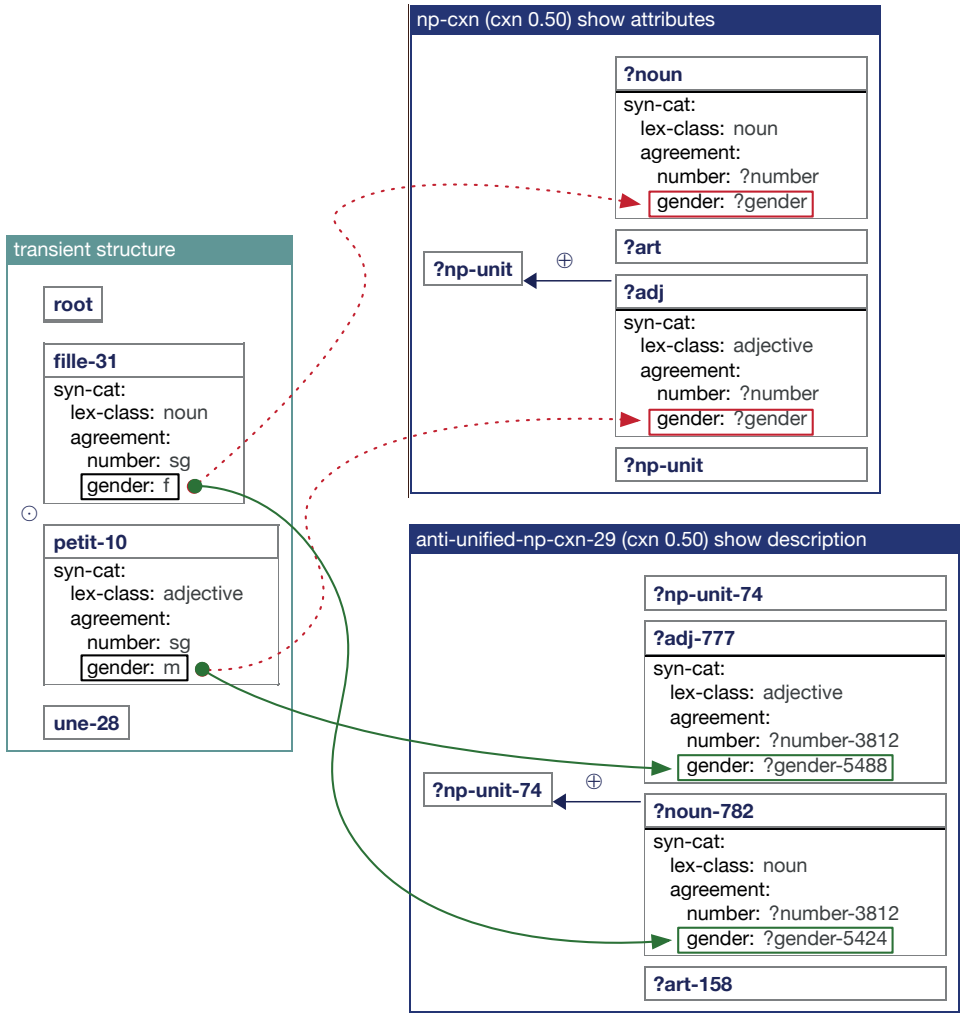


Figure 5.5: An illustration of variable decoupling in the case of a gender mismatch. The construction at top left corner requires the value of the gender feature of the noun and the adjective to be the same. This is however not the case in the transient structure (red dotted arrows). The anti-unification algorithm solves the problem by decoupling the '?gender' variables in the noun and the adjective (green arrows). The anti-unified construction is shown at the bottom right corner.

immediately left-adjacent to the noun. This construction is shown at the top of Figure 5.6. The word order constraints are expressed in the '?np-unit' on the right-hand side of the construction using 'meets' predicates. 'meets(unit-a,unit-b)' means that 'unit-a' should be immediately adjacent to 'unit-b'. Within the 'meets' constraints of the np-cxn, the orange, purple and grey colors of the '?art', '?adj' and '?noun' variables indicate their coupling with the names of the article, adjective and noun units respectively.

Now, consider the input utterance "une fille petite" (*a girl small*), in which the article "the" 'meets' the noun "fille" and the noun "fille" 'meets' the adjective "petite". This transient structure is shown in the middle of Figure 5.6 with the variable couplings highlighted in color. The word order constraints in the construction are incompatible with the word order constraints in the transient structure. Therefore, the construction cannot apply, as indicated with a red arrow in the figure. In order to overcome these incompatibilities, the anti-unification algorithm decouples the conflicting variables in the 'meets' constraints from the variables that serve as unit names. This is shown at the bottom of Figure 5.6. The unit names of '?adj-1452' (light green) and '?noun-1452' (yellow) do not occur in the 'meets' constraints any more, and are replaced by the new variables '?adj-1468' (dark green) and '?noun-1467' (purple). With these generalisations, the construction can apply, as indicated by the green arrow.

Note that only the occurrences of the '?adj' and '?noun' variables in the 'meets' constraints are decoupled. The occurrences of these variables in the subunits feature on the left-hand side of the construction are still coupled to the unit names. This is because these variable couplings did not cause any conflicts and should therefore not be broken during the generalisation.

When studying the anti-unified construction at the bottom of Figure 5.6, one can see that the word order constraints are not really meaningful any more. When the construction would be used in production, there are no features left that would actually constrain the word order. We will come back to this problem in Chapter 6, in which we will present an algorithm that constrains generalised constructions back to the observation for which they were generalised. In the case of deviating word order, the new word order will then be incorporated into the construction.

Besides demonstrating generalisation through variable decoupling, this word order example also clearly illustrates why it can be useful to handle word order just like any other feature, as is usually done in FCG. This makes it possible to process, generalise and learn word order with all the same techniques that are available for other features. This view is rather uncommon in formal grammars, probably due to the central role that word-order based tree structures have traditionally played in these formalisms.

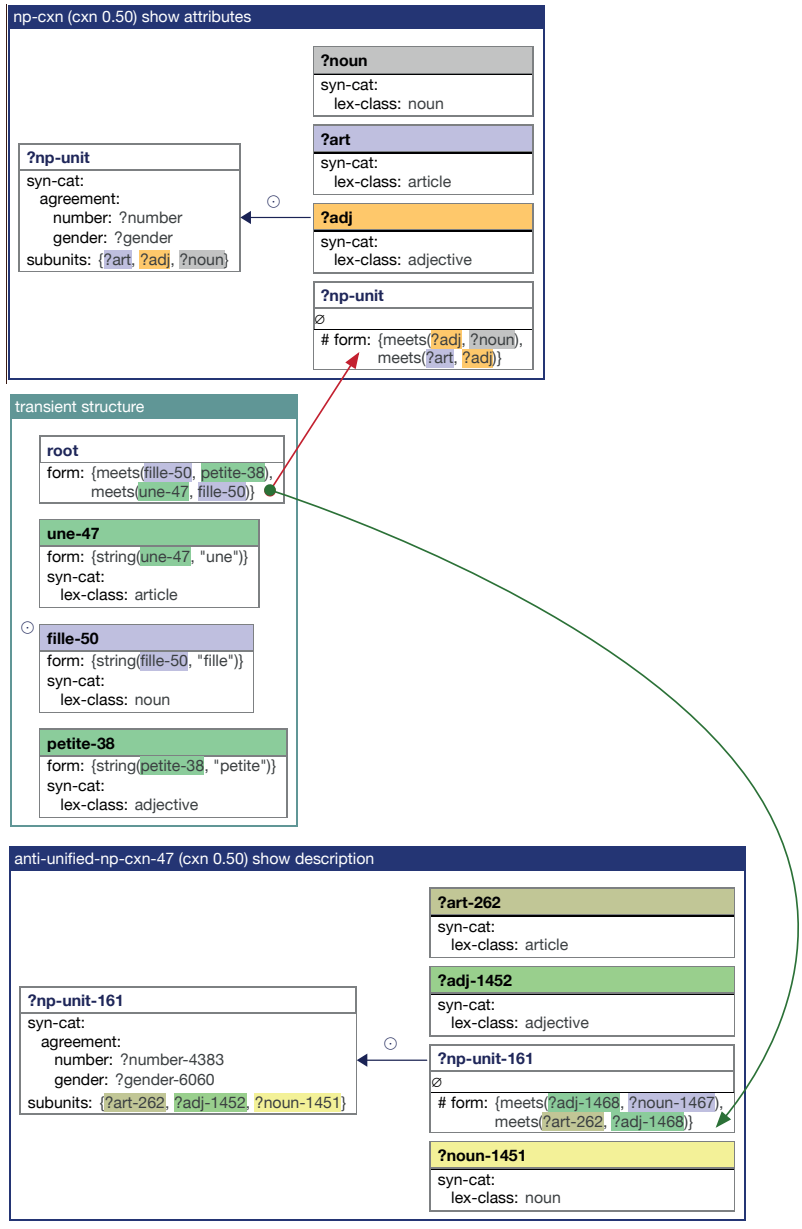


Figure 5.6: An illustration of variable decoupling in the case of a deviant word order. The construction on top cannot apply to the transient structure in the middle because the word order constraints don't match (red arrow). This problem is solved in the anti-unified construction at the bottom by decoupling the problematic variables from the unit names (green arrow). For example, the unit name '?noun-1451' (yellow) in the anti-unified construction is decoupled from the free variable 'noun-1467' in the meets constraint (purple).

5.5.2 Value Relaxation

The second kind of generalisation made by the anti-unification algorithm is called *value relaxation*. Value relaxation is applied when a value of a particular feature in the construction is different from the value of that feature in the transient structure. The value of the feature in the construction is then generalised by replacing it with a free variable. The following example demonstrates how value relaxation is used in the case of a non-matching grammatical category.

The np-cxn that we have used in the previous two examples combines three units into a noun phrase. The values of the 'lex-class' feature of these units need to be 'article', 'adjective' and 'noun' respectively. This construction is shown in the top-right corner of Figure 5.7. Now, suppose the grammar is used to comprehend the utterance "ma petite fille" (*my little daughter*), in which "ma" (*my*) is not of 'lex-class: article', but of 'lex-class: possessive-pronoun'. The transient structure before application of the np-cxn is shown at the left of Figure 5.7. A red dotted arrow indicates the two values that don't match ('article' and 'possessive-pronoun'). The matching conflict is solved by generalising the symbol 'article' from the construction into a free variable '?article-292'. The construction can now apply, as '?article-292' can be bound to 'possessive-pronoun' during the matching process.

5.5.3 Feature/Predicate Deletion

The third kind of generalisation made by the anti-unification algorithm is called *feature deletion* or *predicate deletion*. Feature or predicate deletion happens when a feature or predicate in the construction is not found at the corresponding place in the transient structure. The generalisation consists of deleting the feature or predicate from the construction. This generalisation has been preferred over the substitution of the feature or predicate with a variable, as this would obscure the FCG construction while being virtually meaningless.

Figure 5.8 shows an example of feature deletion. The np-cxn in the top-right corner combines three units with 'lex-class' 'article', 'adjective' and 'noun' into a noun phrase. Moreover, the construction matches on a 'definite' feature in the article unit. The transient structure, shown on the left side of the figure, consists of three units with 'lex-class' 'article', 'adjective' and 'noun', but the article unit does not contain a 'definite' feature. Hence, the construction cannot apply, as indicated by a red dotted arrow. Anti-unification solves the problem by deleting the 'definite' feature from the construction. The resulting generalised construction is shown in the bottom right cor-

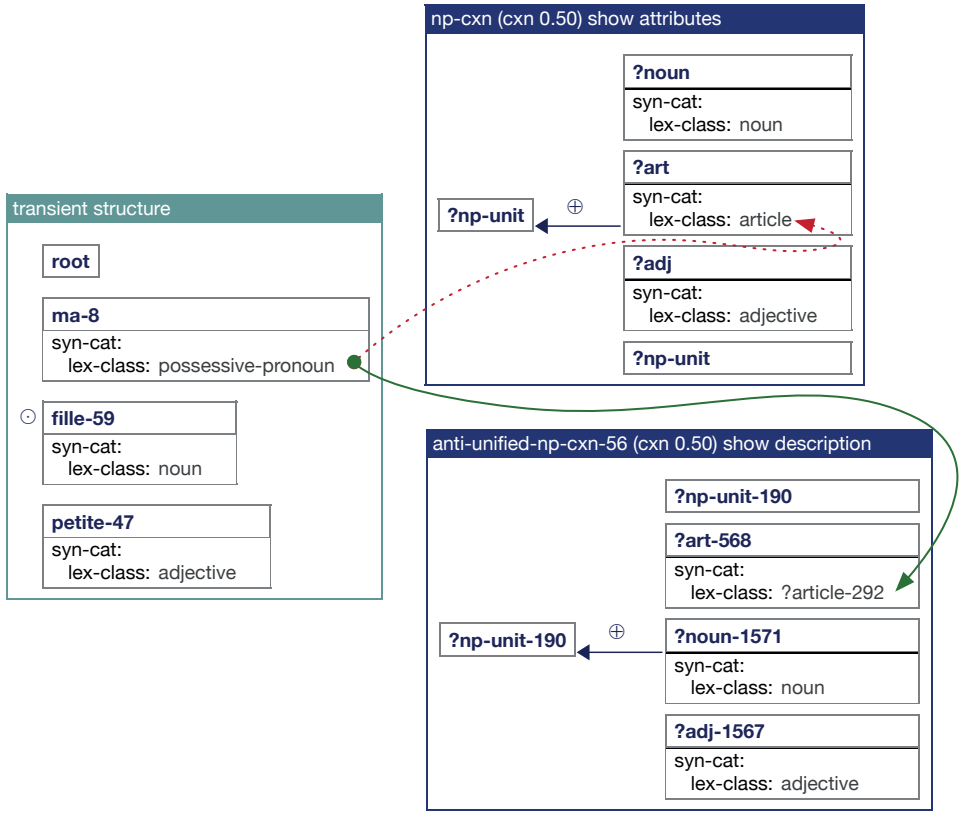


Figure 5.7: An illustration of value relaxation for solving a mismatch in grammatical category. The red arrow highlights that the value of the lex-class feature in the '?art' unit of the construction ('article') does not match the value of the corresponding feature in the transient structure ('possessive-pronoun'). The anti-unification algorithm relaxes the value in the construction to a free variable ('?article-292'). The green arrow indicates that the construction can now apply by binding '?article-292' to 'possessive-pronoun'.

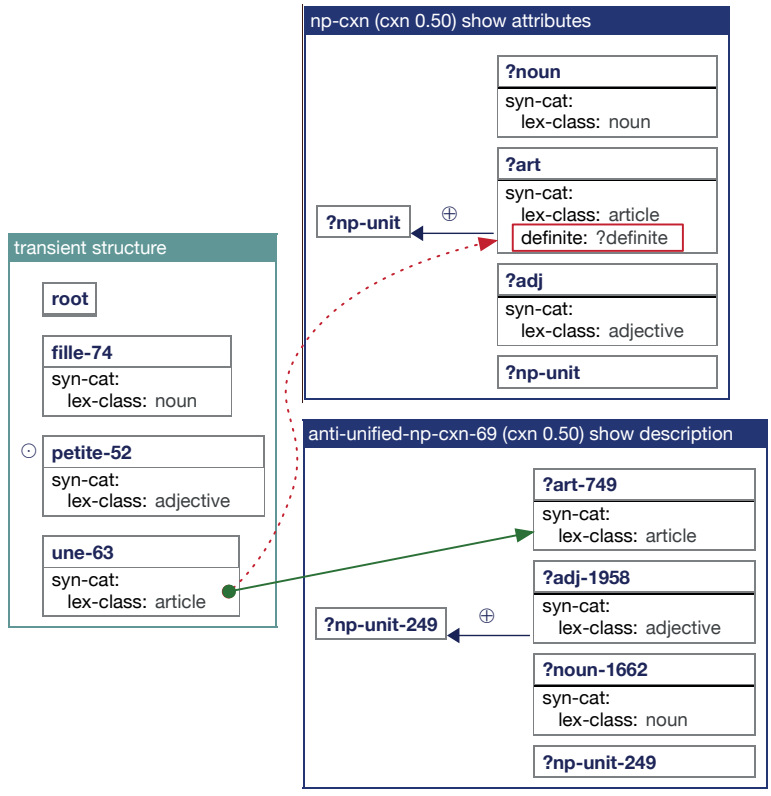


Figure 5.8: An illustration of feature deletion. The constructions matches on a 'definite' feature in the article unit. This feature is however not present in the article unit in the transient structure, blocking construction application. Anti-unifications solves the problem by deleting the the 'definite' feature from the construction.

ner of the figure. The green arrow indicates that the 'une-63' unit now matches the '?art-749' unit.

5.5.4 Unit Deletion

The fourth kind of generalisation that is made by the anti-unification algorithm is called *unit deletion*. Unit deletion consists of deleting a unit from the construction. It is applied when a unit from the construction does not find a suitable unit in the transient structure to match with. It is always considered as an option, but comes at quite a high cost. It will only be returned as the best solution when the cost of deleting a unit exceeds the cost of anti-unifying the features of that unit with any available unit of the transient structure.

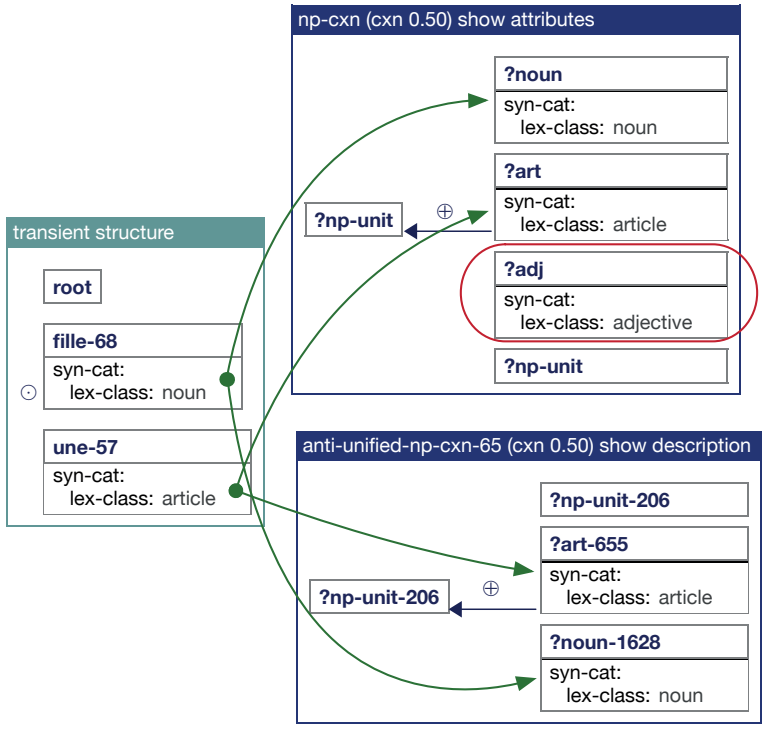


Figure 5.9: An illustration of unit deletion. The np-cxn matches on three units, with 'lex-class' 'article', 'adjective' and 'noun'. The transient structure only contains two units, with 'lex-class' 'article' and 'noun'. Anti-unification solves the problem by deleting the unit with 'lex-class' 'adjective' from the construction.

Figure 5.9 demonstrates a case of unit deletion. The np-cxn, shown in the top right corner of the figure, combines an article, an adjective and a noun into a noun phrase. The input utterance is now "une fille" (*a girl*), consisting of an article and a noun. The transient structure after application of the lexical constructions is shown at the left of the figure. The '?art' unit and '?noun' unit in the construction can be matched with the 'une-57' unit and 'fille-68' unit in the transient structure respectively, as indicated by green arrows. The '?adj' unit cannot be matched with any unit, as there are no units left. This is indicated with a red ellipse. Anti-unification generalises the construction into a construction with one unit less, now only requiring an article and a noun. The '?adj' unit is deleted from the construction. The anti-unified construction is shown at the bottom right of the figure.

5.6 Anti-Unification and Type Hierarchies

The anti-unification-based generalisation operator that was introduced and demonstrated in the previous sections returns a new construction that constitutes the least general generalisation of a construction that matches a given transient structure. All generalisations are captured locally within this new construction. The current section introduces a version of the operator that allows capturing certain generalisations in the type hierarchy of the grammar, instead of in the new construction itself. This has the advantage (i) that more fine-grained generalisations can be captured, (ii) that in cases in which all necessary generalisations can be captured in the type hierarchy, no new construction needs to be added to the construction inventory, and (iii) that the generalisations that are captured can immediately be used by all constructions in the grammar.

The generalisations that can be captured in the type hierarchy of a grammar are value relaxations. As explained above, a value relaxation is performed when a constant in a construction conflicts with a constant in the transient structure. In that case, the standard generalisation operator will replace the constant in the construction with a free variable in the anti-unified construction. This version of the operator will add both constants to the type hierarchy of the grammar, as well as a link that indicates that the constant in the transient structure is a subtype of the constant in the construction. For example, imagine that an intransitive construction that matches on an NP with a feature-value pair 'lex-class: noun-phrase' is anti-unified with with a transient structure that contains a unit with a feature-value pair 'lex-class: proper-name'. In this case the constants 'noun-phrase' and 'proper-name' will be added to the type hierarchy of the grammar, as well as a link from 'proper-name' to 'noun-phrase'. From this moment on, all constructions in the grammar that match on units containing 'lex-class: noun-phrase' will be able to also match on units containing 'lex-class: proper-noun'.

In order to achieve this behaviour, a number of changes need to be made to the generalisation operator and the anti-unification algorithm it makes use of:

- The unit pairing algorithm now makes use of the matching algorithm that was extended for being used with type hierarchies and that was introduced in Section 4.3.3.
- In addition to the resulting-pattern, pattern-bindings, source-bindings and cost, the anti-unification algorithm will now also collect and return a set of required type hierarchy links.
- The anti-unification algorithm treats constants that are connected through the

type hierarchy as if they were equal. No generalisation are performed for these constants.

- When two constants are not equal or connected through the type hierarchy, the anti-unification algorithm adds to the set of required type hierarchy links a link from the constant in the source to the constant in the pattern. The resulting-pattern, pattern-bindings and source-bindings are not altered.
- The operator now also includes a cost parameter for adding a new type hierarchy link.

The use of this version of the generalisation operator is shown in Figure 5.10. In this example, an NP construction matches on three adjacent units with the lex-classes 'determiner', 'adjective' and 'noun'. This construction cannot apply to the transient structure created by the application of the lexical constructions to the utterance "ma petite fille", as the value of the 'lex-class' feature in the 'ma-3'-unit is 'possessive-pronoun'. This is indicated with a red, dotted arrow in the figure. The generalisation operator adds the constants 'possessive-pronoun' and 'determiner', as well as a link from 'possessive-pronoun' to 'determiner', to the type hierarchy of the grammar. Through the type hierarchy, the construction can apply to the transient structure, as indicated by the green arrows. There are no changes needed inside the NP construction, and the generalisation that was incorporated into the type hierarchy of the grammar can now be used by all constructions of the grammar.

The generalisations made by this version of the generalisation operator are much more fine-grained then those made by the standard version, and it therefore greatly reduces the risk of over-generalisation. It also allows the incremental build-up of the type hierarchy of a grammar, gradually expanding the coverage of existing constructions. This is particularly useful in evolutionary linguistics experiments, as will be discussed and demonstrated in chapter 7.

5.7 Anti-Unification as a Debugging Tool in Grammar Engineering

Apart from its applications in language processing and evolutionary experiments, anti-unification also has a direct application in grammar engineering. When adding new constructions to a grammar, or modifying existing ones, a grammar engineer often needs to find out why a particular construction does not apply to a particular transient structure. This is not an easy task, as it requires going through all units, features and

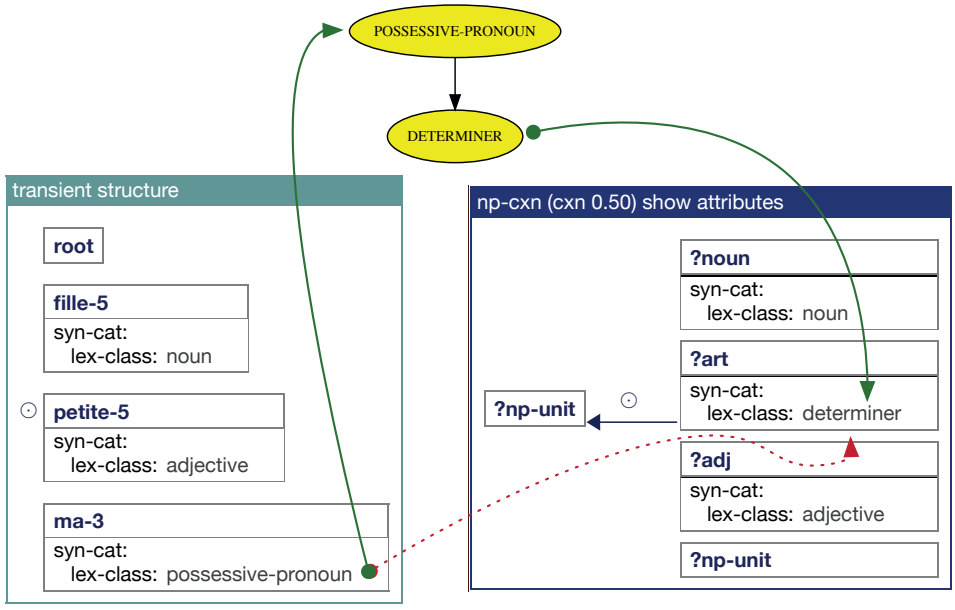


Figure 5.10: Illustration of the anti-unification-based generalisation operator performing value relaxation by capturing the hierarchical relation between two constants in the type hierarchy of a grammar. The red arrow highlights that the value of the lex-class feature in the '?art' unit of the construction ('determiner') does not match the value of the corresponding feature in the transient structure ('possessive-pronoun'). The anti-unification operator adds the constants 'possessive-pronoun' and 'determiner' to the type hierarchy of the grammar, as well as a link between the two constants. The green arrow indicates that the construction can now apply through the type hierarchy. There are no changes needed in the construction itself.

values in the construction and the transient structure, keeping all bindings in mind, until the conflict is found. Although experienced grammar engineers appear to have developed a remarkably well-trained eye for spotting these conflicts, an automated tool would still drastically cut development time.

The anti-unification machinery introduced in this chapter was specifically designed for generalising over conflicts that block construction application. By extending the algorithm with a few extra features, the machinery can be used to locate the conflicts and provide detailed feedback about their nature. I have made these extensions and integrated an anti-unification-based debugging tool into the FCG development environment.

5.7.1 Extending the Anti-Unification Algorithm

For debugging purposes, the anti-unification algorithm needs to present the grammar engineer with feedback about which specific features and values in the construction and transient structure cause matching conflicts. Only very few extensions to the algorithm are needed in order to achieve this. They can be summarized in the following three points:

- First, the algorithm needs to deal with conflicting values in the pattern and the source that are not variables, for example in (agreement (case nominative)) in the pattern and (agreement (case accusative)) in the source. After anti-unification, the algorithm loops through the pattern-bindings and the source-bindings and looks for non-variable values that share a substitution. In our example, it finds (nominative . ?var-6) and (accusative . ?var6), which means that the anti-unification algorithm has substituted both 'nominative' in the pattern and 'accusative' in the source with '?var-6'. The algorithm can then give feedback to the grammar engineer, specifying that 'nominative' is expected in the transient structure where 'accusative' is found. If the anti-unified construction is also given as feedback, the algorithm can also specify that the conflict is exactly located in the original construction where '?var-6' is found in the anti-unified construction. Naturally, this works not only for atomic values, but also for complex values.
- Second, the algorithm needs to deal with variables that occur multiple times in the pattern, where different values occur in the transient structure. This is for example the case if (gender ?gender) occurs twice in the construction, and (gender masculine) and (gender feminine) occur at those places in the transient structure. The variables are found by looping through the pattern-bindings and collecting the variables that occur more than once together with their substitutions. In our

example this would be (?gender . ?var-2) and (?gender . ?var-3). Then, the algorithm searches for the substitutions of these variables in the source-bindings, namely (masculine . ?var-2) and (feminine . ?var-3). The feedback given to the grammar engineer specifies that the two occurrences of '?gender' in the construction correspond to two different values in the transient structure, namely 'masculine' and 'feminine'. Here as well, the feedback can add that the conflict is exactly located in the original construction where '?var-2' and '?var-3' are located in the anti-unified construction.

- Third, the algorithm needs to deal with discarded features. The discarded features cannot be computed from the resulting-pattern, the pattern-bindings or the source-bindings, as they do not appear in the anti-unification result at all. Therefore, the anti-unification algorithm was modified in such a way that it collects all features that are discarded into a list, which is also returned by the algorithm. This list of discarded features can then be presented as feedback to the grammar engineer.

When multiple unit pairing options are possible, multiple anti-unification analyses are computed and ranked according to their cost. As the errors made by grammar engineers are mostly small (in anti-unification terms), the first analysis is virtually always the desired one. For cases where there are larger errors, the cost for unit-deletion is set to a high value. This ensures that unit-deletion options are not considered as feedback, because the grammar engineer is usually interested in conflicts in features and values and not in deleting units.

5.7.2 Example

Figure 5.11 shows an example of the use of the anti-unification algorithm that was extended for debugging purposes. The left side of the figure shows an 'np-cxn' that combines a determiner and an adjacent noun into a noun phrase. The right side of the figure shows a transient structure that was created by the application of the lexical constructions for 'une' and 'fille' to the input utterance "une fille" (*a girl*). The construction cannot apply to the transient structure, and anti-unification is used to reveal the conflicts.

First, the units need to be paired, which is a straightforward task in this case. The '?noun'-unit in the construction matches the 'fille-39'-unit in the transient structure. After pairing these two units, only two other units, the '?det'-unit and the 'une-13'-unit are left, so these two units can be paired as well. Then, the paired units (?noun . fille-39) and (?det . une-13) are anti-unified. This yields the following analysis:

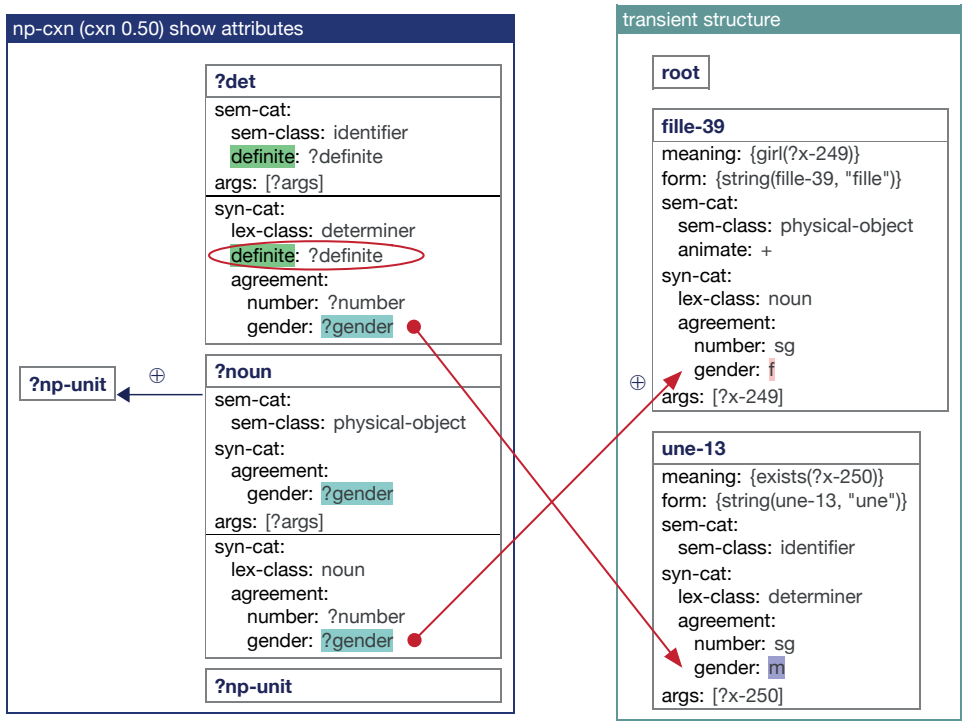


Figure 5.11: Example of the use of anti-unification for debugging purposes in FCG. The 'np-cxn' construction on the left cannot apply to the transient structure on the right. The anti-unification feedback report shows that the '?gender' variable occurs in the construction where both 'f' and 'm' occur in the transient structure and that the 'definite' feature required by the construction is not found in the transient structure.

- The discarded-features list returned by the anti-unification algorithm contains the feature (definite ?definite-93). This feature is required by the construction, but cannot be found in the transient structure and therefore blocks construction application. The feature is highlighted in green in Figure 5.11. The grammar engineer will either need to delete the feature from the 'np-cxn' or ensure that a feature (definitie -) is added by the 'une-cxn'.

- The list of pattern-bindings includes two substitutions for the '?gender' variable in the construction. These substitutions occur in the source-bindings as substitutions for the values 'm' and 'f' in the transient structure. In Figure 5.11, the '?gender' variable is highlighted in dark green and red arrows point to the conflicting values 'm' en 'f', highlighted in red and blue respectively. The grammar engineer will see that the 'une-cxn' wrongly assigns the gender 'm' to the lexical item and will easily be able to correct this error.

5.7.3 Integration into the FCG Environment

The debugging version of the anti-unification algorithm has been tightly integrated into the FCG development environment. Grammar engineers heavily rely on an interactive web interface that visualises construction inventories, constructions, transient structures, construction application processes and other FCG objects. When comprehending or formulating an utterance, the standard visualisation shows the utterance or meaning representation to be processed, the construction inventory, the construction application process, and the resulting meaning representation or utterance. In order to manually apply a construction, the grammar engineer can search for it in the construction inventory and drag and drop the construction onto a node in the construction application process. When dragging and dropping a construction on a node to which it cannot apply, a feedback field appears at the bottom of the web interface. This feedback field displays the different conflicts that block the application of the construction to the transient structure.

The dragging and dropping of a construction from the construction inventory onto a node in the construction application process for debugging purposes is demonstrated in Figure 5.12. Just like in the example in the previous section, the utterance "une fille" (*a girl*) is being comprehended. The 'fille-cxn' and 'une-cxn' apply, indicated by the green boxes in the application process, but the 'np-cxn' doesn't. The grammar engineer would like to find out why, and he drags the 'np-cxn' (blue box) out of the construction inventory and drops it on the last node in the construction application process. The construction cannot apply, but a feedback field appears at the bottom of

the screen. In the feedback field, the debugging information that was explained in the example in the previous section is shown. The first part specifies that the '?gender' variable from the construction is bound to both 'f' and 'm' in the transient structure. The second part specifies that the feature-value pair (definite ?definite) is expected in the transient structure, but not present.

For any construction that is dragged and dropped onto any node in the construction application process (not only the last one), the construction will either apply and extend the application process tree, or a new debugging feedback field will appear at the bottom of the screen.

5.8 Conclusion

Human language use is creative, open-ended and full of innovations. Moreover, a language needs to be learned and needs to adapt to novel situations. In computational construction grammar terms, this means that the grammar and processing mechanisms that are used, need to be flexible enough to handle utterances and meaning representations that deviate from the norm that is encoded in the grammar. Previous work in this area has mainly focussed on the sophisticated semantic processing that is needed to overcome grammatical impasses with reference to a shared context. In this chapter, I have introduced a complimentary strategy that can overcome these impasses without needing access to a shared context. This is particularly useful in the case of displaced communication, or when the necessary common ground between two agents has not yet been established. The strategy is based on making construction application flexible.

The algorithms for flexible construction application that I have developed and integrated into FCG are based on anti-unification. When a construction cannot apply to a transient structure, anti-unification finds the least general generalisation of the construction that can apply to the transient structure. Four different kinds of generalisations can be made: variable decoupling, value relaxation, feature/predicate deletion and unit deletion. Depending on the flavour of the algorithm, value relaxations are captured within the anti-unified construction, or within the type hierarchy of the grammar. Each kind of generalisation comes at a certain cost, so that the anti-unification results of different constructions can be compared.

Apart from its applications in language processing and evolution experiments, the anti-unification-based machinery for flexible construction application can also be used as a debugging tool in grammar engineering. With a few extensions to the algorithm, it helps the developer understand why a certain construction could not apply to a given

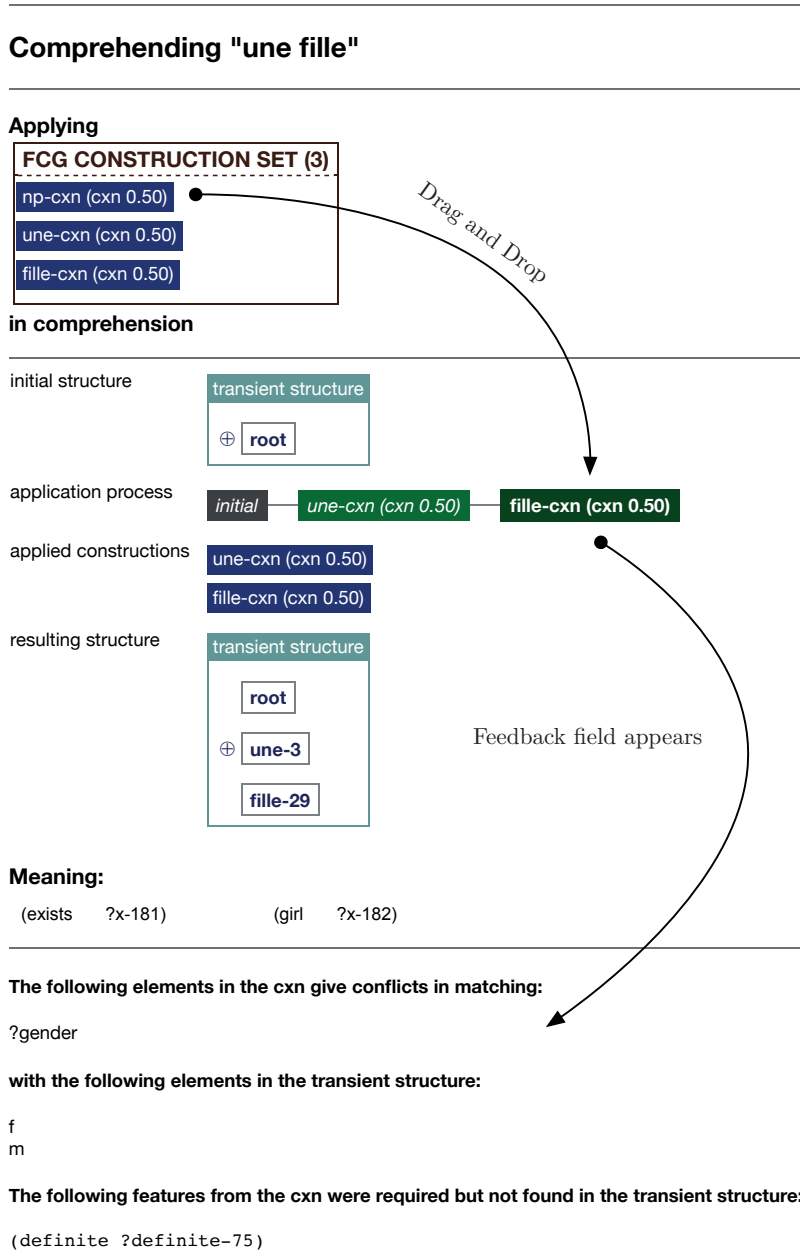


Figure 5.12: A screen shot showing how the anti-unification-based debugging tool has been integrated in the FCG development environment. In the web-interface, constructions (blue boxes) can be dragged and dropped onto nodes in the construction application process (green boxes) to apply. If the construction cannot apply, a feedback field with debugging information appears at the bottom.

transient structure.

Generalisation of constructions is a first important step in the learning of constructions. The second step, namely specialisation of constructions, will be discussed in the next chapter. Together, generalisation and specialisation will then be incorporated in powerful diagnostics and repairs that facilitate the learning of constructions from novel observations. The case study in chapter 7 will demonstrate the use of anti-unification to build up the type hierarchy of a grammar in an evolution experiment on the emergence of syntactic patterns.

Chapter 6

Specialising Constructions using Pro-Unification

6.1 Introduction

In the previous chapter, I have shown how the generalisation of existing constructions can resolve grammatical impasses. Generalisation was operationalised through the anti-unification of constructions with respect to novel observations. It provided a means to relax those features in a construction that blocked its application, making construction application more flexible. Suppose, for example, that a noun phrase construction requires a specific word order and that an observation features a different word order. In order to be able to apply the construction, its word order constraints are relaxed through generalisation. Then, the construction can apply and processing can continue.

While the generalisation of a construction can often overcome an impasse and lead to a valid solution for the problem at hand, it is often too unconstrained to be stored for later reuse. In the word order example, the generalised construction does not constrain the word order any more. This means that the grammar might produce noun phrases with any word order, while it should only produce those with the word order that was observed or the one that was already known.

Apart from the *generalisation* step discussed in the previous chapter, learning new constructions also requires a *specialisation* step. While the generalisation step relaxes the conflicting elements of a construction, and makes the construction more general, the specialisation step integrates specific elements from the observation into the new construction. This ensures that the construction is constrained enough to be stored for later reuse. In the word order example above, specialisation would typically incorporate features encoding the observed word order into the new construction.

This chapter introduces a strategy for specialising constructions with respect to novel observations, as well as a framework that integrates the generalisation, specialisation and consolidation of constructions. It is structured as follows. First, I will discuss the concepts of generalisation and specialisation from a learning point of view (6.2). Then, I will introduce a mechanism for specialising constructions, called pro-unification (6.3). Finally, I will present the integration of generalisation, specialisation and consolidation into FCG's meta-layer framework (6.3.2) and demonstrate this with a number of examples (6.4).

6.2 Generalisation and Specialisation

The hypothesis that learning can be achieved by generalising and specialising hypotheses over observations has been exploited in many domains of machine learning. In inductive learning and inductive logic programming, the hypothesis space is typically structured in terms of generalisation and specialisation (Michalski, 1983; De Raedt and Bruynooghe, 1992; Muggleton and De Raedt, 1994). This is also the case for version space learning, in which positive and negative examples respectively generalise and specialise the lower and upper bounds of the hypothesis space (Mitchell, 1978, 1982; Rendell, 1986; Dubois and Quafafou, 2002). Learning through generalisation and specialisation has also been applied in (contextual) reinforcement learning, where actions that were successful in certain contexts are generalised to new contexts and unsuccessful actions are inhibited in specific contexts (Berthouze et al., 2007). When it comes to language learning, grammar induction through generalisation and specialisation has been extensively studied in the field of formal language theory (Kapur and

Bilardi, 1992; Jain and Sharma, 1998; Oates et al., 2006), while linguistic theory and the associated formalisms have traditionally less focussed on this topic. In the context of learning typed feature structure grammars, Lüngen and Sporleder (1999) present a method to automatically induce inheritance hierarchies for morphological and lexical types, and Ciortuz (2002a, 2003) presents an extension to the LIGHT system (Ciortuz, 2002b) that implements generalisation and specialisation operators for inductively learning attribute-path values inside type definitions, for example for HPSG grammars. In the field of computational construction grammar, Chang (2008); Beuls et al. (2010); Gerasymova and Spranger (2010) introduce mechanisms that create new constructions by generalising over two existing constructions or by recombining structural elements from two existing constructions.

In our case , the specific challenge is to generalise and specialise constructions with respect to novel observations. Observations are represented as transient structures, which are gradually expanded during comprehension and formulation by the subsequent application of constructions (see chapter 3). When a novel observation is encountered, by which is meant that an utterance (in comprehension) or meaning representation (in production) cannot be completely processed in a satisfactory way by the existing constructions of the grammar, one of the existing constructions will first be generalised over the problematic transient structure, and then specialised towards this transient structure. This ensures that the new construction is both general enough to cover the novel observation and constrained enough to be stored in the construction inventory.

A schematic overview of this process and the algorithms involved is shown in Figure 6.1 (adapted from Steels and Van Eecke (2018)). The transient structure representing the novel observation is shown in green at the bottom of the figure. An existing construction is shown in blue at the left side of the figure. This construction cannot apply to the transient structure because the matching phase fails. Using the anti-unification algorithm discussed in the previous chapter, the least general generalisation of the construction given the transient structure is computed. This generalised construction, shown at the top of the figure, can apply to the construction, as the matching phase succeeds. However, its generality may cause important side-effects if it would be stored as such in the construction inventory. Therefore, it is constrained towards the transient structure by a pro-unification process that will be explained in more detail in the next section. Pro-unification will incorporate specific properties of the transient structure into the construction. This will decrease the probability that the new construction will inappropriately apply to observations in the future.

As for the properties of the generalised and specialised constructions, the generalised construction can apply to all transient structures to which the existing construction

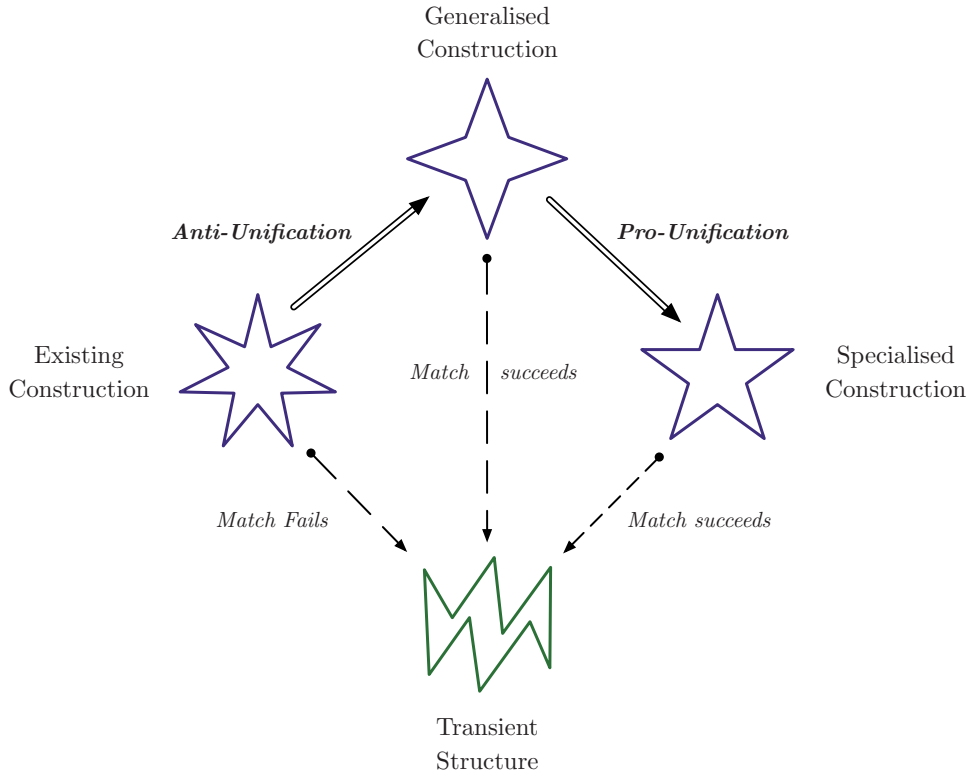


Figure 6.1: A schematic representation of how a new construction can be learned by generalising and specialising an existing construction with respect to a novel observation. The transient structure representing the novel observation is shown in green at the bottom of the Figure. The existing construction shown at the left cannot apply to this transient structure. It is first generalised over the transient structure using the anti-unification algorithm discussed in the previous chapter, yielding the generalised construction shown at the top. This construction can apply to the transient structure, but may be to general to store in the construction inventory. Therefore, it is specialised towards the transient structure using the pro-unification algorithm introduced later in this chapter, yielding the specialised construction shown at the right.

could apply, plus the transient structures that require the same generalisation as the one that was required for covering the novel observation. The specialised construction is guaranteed to apply to the transient structure representing the novel observation, but cannot necessarily apply to all transient structures to which the original construction could apply. In fact, it is well possible that it cannot apply to any of them. This might be desirable as a new construction is often not meant to become a competitor of an existing construction, but only to cover cases that the existing construction did not cover.

6.3 Anti-Unification and Pro-Unification

I have called the process of generalising a construction over a transient structure *anti-unification*, after the algorithm that performs the generalisation. Although the task of designing an anti-unification algorithm that works on FCG constructions and transient structures was certainly challenging, especially when it came to special operators and the unit structure in which the unit names are variables, I could benefit from both an established literature on well-understood anti-unification algorithms, and a clear idea of what the least general generalisation of a construction should look like. Therefore, it was possible to design and implement an algorithm that is very general in the sense that it purely works on symbolic structures and does not need any grammar-specific or problem-specific information.

For the specialisation of a construction towards a transient structure, the picture is not so clear. The main problem is that it is difficult to define an adequate level of specialisation. While the specialisation process should incorporate additional constraints from the transient structure into the construction, it cannot include all constraints, as this would make the construction so specific that it would only be able to apply to exactly the same observation, or possibly even to the exact same transient structure (e.g. if the unit names would be incorporated into the construction). On the other hand, when not enough constraints are incorporated into the construction, the side-effects of applying the construction where it is not appropriate still remain. The choice of which elements from the transient structure to incorporate into the construction (e.g syntactic categories, word order constraints or extra units, ...) will often be experiment-specific and grammar-specific, although more general mechanisms can also be isolated. In analogy to the anti-unification process that generalises constructions over transient structures, I will call the process of specialising constructions towards transient structures *pro-unification*. While anti-unification was operationalised as a single, very general algorithm, pro-unification should more be seen as a collection of

strategies, some of which are more generally applicable, others of which are more grammar-specific or experiment-specific.

6.3.1 A General Pro-Unification Algorithm

I will now introduce a general and powerful pro-unification algorithm. Like the anti-unification algorithm, it purely works on symbolic structures and does not require any grammar-specific or experiment-specific information. The algorithm proceeds as follows. It first matches the (generalised) construction against the transient structure and collects the matching bindings. Then, it goes through these bindings and looks whether there are variables (from the construction) that are bound to the same constants (in the transient structure). If so, these variables in the construction are made equal by replacing them with a single, new variable.

A minimal example is shown in Figure 6.3. Suppose that we are comprehending the utterance "the book". The two lexical constructions on the left side of the figure, 'a-cxn' and 'book-cxn' apply, yielding the transient structure in the middle of the figure. This transient structure contains two units, 'a-6' and 'book-2'. 'a-6' contains a feature 'lex-class: article' and 'book-2' contains a feature 'lex-class: noun'. Both units also contain an agreement feature 'number: singular'. The np-cxn in the upper right corner matches on two units: '?art', which requires a feature 'lex-class: article' and '?noun', which requires a feature 'lex-class: noun'. Additionally, '?art' also matches on an agreement feature 'number: ?number-article' and '?noun' matches on an agreement feature 'number: ?number-noun'. When applying the pro-unification algorithm, it will first match the 'np-cxn' and the transient structure. The matching process yields the bindings '((?art . a-6) (?noun . book-2) (?number-article . singular) (?number-noun . singular))'. The algorithm then loops through these bindings and and detects that both '?number-article' and '?number-noun' in the construction are bound to the same constant 'singular' in the transient structure. It converts this information into the following 'renamings' ((?number-article . ?number-noun-1) (?number-noun . ?number-noun-1)). Then, the new, specialised construction is created, as shown in the bottom right corner of the figure. In this construction, the 'renamings' have been applied, meaning that throughout the construction, '?number-article' and '?number-noun' have been substituted by a single, new variable '?number-noun-1'. Note that the substitution has not only renamed the variables that were involved in matching, but all '?number-noun' and '?number-article' variables in the construction, including the one in the '?np-unit' in the contributing part of the construction. By substituting the '?number-article' and '?number-noun' variables by the same, new variable, the pro-unification algorithm has ensured that the specialised construction will only be able to apply when both

units in the transient structure have the same value for the number feature, whereas the original construction could apply to any two units. It has effectively induced this equality constraint from the transient structure and incorporated it into the specialised construction.

The Common Lisp implementation of the algorithm that I have included in Fluid Construction Grammar is shown in Figure 6.2. The function *apply-pro-unification* deals with extracting the matching-pattern from the construction, creating the new specialised construction, and substituting the renamings. The function *pro-unify* computes the actual renamings based on the matching-pattern from the construction and the transient structure.

6.3.2 Integration in FCG's Meta-Layer Architecture

I have integrated the learning of new constructions using anti-unification and pro-unification in FCG's meta-layer architecture (Steels and Van Eecke, 2018; Van Eecke and Beuls, 2017). As explained in Section 3.5 above, the meta-layer is FCG's preferred place to handle processing problems and learn solutions to these problems. The meta-layer architecture is based on three concepts: *diagnostics*, *repairs* and *consolidation strategies*. Diagnostics are tests that are run after each construction application and which can signal problems of different types. Repairs are active at the meta-layer and implement strategies that solve different types of problems. Consolidation strategies ensure that successful solutions to these problems are stored in such a way that a next occurrence of the same problem will not require meta-layer processing any more. The integration of anti-unification and pro-unification in the meta-layer architecture consisted in implementing the following diagnostic, repair and consolidation strategy:

- **Diagnostic: 'no-match-or-solution'.** This diagnostic checks whether a node is fully expanded, which means that no more constructions can apply[1]. If this is the case, it checks whether all elements of the meaning representation (in comprehension) or unit structure (in formulation) are connected into a single network. If this is not the case, the diagnostic signals a problem of the type 'no-match'.

- **Repair: 'anti-unify-pro-unify'.** This repair applies to problems of the type 'no-match'. It first extracts the transient structure from the node. Then, it loops

[1] The diagnostic actually checks the 'fully-expanded' slot of the node object. This slot is set to nil by default and set to true when, for this node, no more constructions are scheduled for application. As this is a standard feature of FCG, the computational overhead created by the diagnostic is very limited.

```
(defun apply-pro-unification (cxn ts direction)
;; Returns a specialised construction which is the result of the
;; pro-unification of cxn with ts in comprehension or formulation
 (let* (;; Get processing construction, matching-pattern and source
        (processing-cxn (get-processing-cxn cxn))
        (matching-pattern (matching-pattern processing-cxn direction
            ))
        (source (left-pole-structure ts))
        ;; Pro-unify matching pattern and source, collect renamings
        (renamings (pro-unify matching-pattern source))
        ;; Make a new construction by copying the existing one
        (new-cxn (copy-object processing-cxn)))
   ;; Set the name of the new construction
   (setf (name new-cxn) (make-id (string-append "pro-unified-" (name
       processing-cxn))))
   ;; Substitute the variables in both poles of the construction
       according to the obtained renamings
   (setf (pole-structure (left-pole new-cxn))
         (substitute-bindings renamings (left-pole-structure
             processing-cxn)))
   (setf (pole-structure (right-pole new-cxn))
         (substitute-bindings renamings (right-pole-structure
             processing-cxn)))
   ;; Convert the processing-cxn into an fcg-cxn
   (processing-cxn->fcg-cxn new-cxn))

(defun pro-unify (pattern source)
;; Returns a list of pro-unification renamings.
 (let (;; match pattern and source, collect bindings
       (matching-bindings (first (match-structures pattern source)))
       (renamings nil))
  (loop for (variable . binding) in matching-bindings
    do  ;; For each pair, if the binding occurs more than once:
    (when (binding-occurs-more-than-once-p variable binding
        matching-bindings)
      (let ((binding-in-renamings (find binding renamings :key '
          first)))
        ;; and the binding occurs already in the list of renamings:
        (if binding-in-renamings
          ;; Push it into the renamings with the same variable
          (push `(,binding ,variable ,(third binding-in-renamings))
              renamings)
          ;; Else, push it into the nenamings with a new variable
          (push `(,binding ,variable ,(make-var)) renamings)))))
   (if renamings
     ;; Return the renamings in the usual bindings format.
     (mapcar #'(lambda (r) (cons (second r) (third r))) renamings)
     +no-bindings+)))
```

Figure 6.2: FCG's implementation of the general pro-unification algorithm.

Figure 6.3: A minimal pro-unification example. The application of the lexical constructions 'a-cxn' and 'book-cxn' on the left yields the transient structure in the middle, containing two units with a feature '(number singular)'. The 'np-cxn' in the right upper corner matches on two units with the features '(number ?number-noun)' and '(number ?number-article)' respectively. The pro-unification algorithm detects that these two variables are bound to the same value in the transient structure and renames '?number-noun' and 'number-article' to 'number-noun-1' throughout the construction. The specialised construction is shown in the lower left corner.

through all grammatical constructions in the construction inventory and anti-unifies each construction with the transient structure. After that, it takes the construction that was anti-unified with the lowest cost and pro-unifies this anti-unified construction with the transient structure. The resulting construction is returned by the repair as a fix-cxn. The fix-cxn is then applied to the transient structure and routine processing can continue.

- **Consolidation Strategy: 'add-cxn'.** If a branch of the search tree leads to a solution, all constructions that were the result of pro-unification in the 'anti-unify-pro-unify' repair are added to the construction inventory. From that moment on, they are treated as normal constructions and can apply in routine processing.

A schematic representation of the integration of these anti-unification and pro-unification based diagnostics and repairs in FCG's meta-layer architecture is shown in Figure 6.4. In the bottom left corner, the initial transient structure TS_t is shown. It is not a solution (Goal-test: nil) and triggers no problems (diagnose: nil). 'Cxn_m' applies and a new node is created, containing transient structure TS_{t+1}. This construction is not a solution and the diagnostic 'no-match-or-solution' signals a problem of type 'no-match', as the node is fully expanded and not all elements of its meaning or unit structure are connected into a single network. This transient structure and diagnostic are shown in orange. The instantiation of a new problem triggers a jump to the meta-layer. There, the repair 'anti-unify-pro-unify' becomes active, as shown in the green box in the middle of the figure. All grammatical constructions of the construction inventory are anti-unified with TS_{t+1} and the anti-unified construction with the lowest score is '$a\text{-}u\text{-}cxn_k$'. This construction is then pro-unified with TS_{t+1}, yielding the construction '$p\text{-}u\text{-}a\text{-}u\text{-}cxn_k$'. This construction is than applied to TS_{t+1} and a new node TS_{t+2} is created. This node is not a solution and the diagnostic signals no problems. Then, routine processing applies Cxn_l to TS_{t+2} yielding TS_{t+3}. This node triggers no problems and qualifies as a solution (Goal-test: t). As this branch of the search tree leads to a solution, the consolidation phase adds the pro-unified anti-unified construction '$p\text{-}u\text{-}a\text{-}u\text{-}cxn_k$' to the construction inventory.

6.4 Demonstration: Learning Word Order Constraints

I will now demonstrate how the combination of generalisation and specialisation using anti- and pro-unification can be used to learn new constructions based on existing constructions and novel observations. I will use an example very similar to the one used in Section 5.5.1 above. In this example, an agent observes the utterance "un dîner formidable" (*a splendid dinner*). His construction inventory contains the lexical

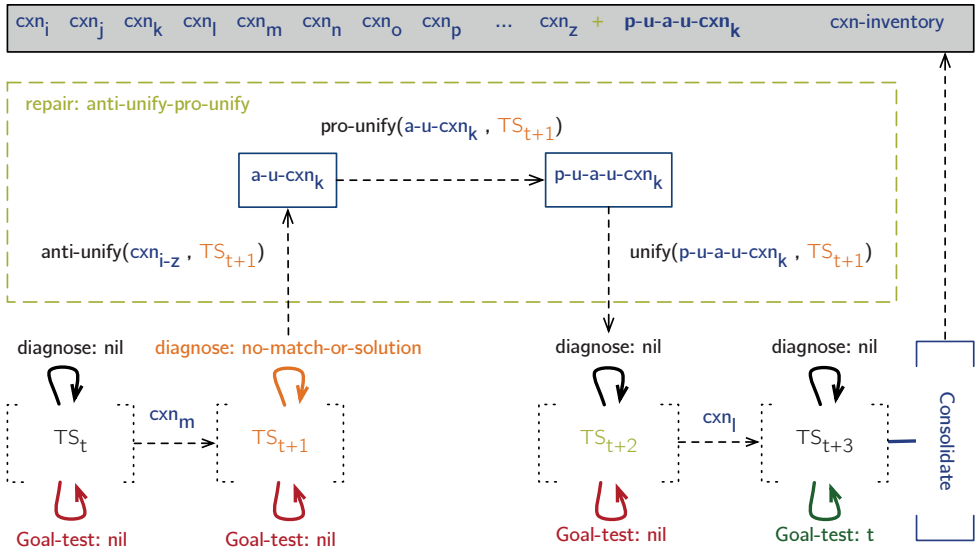


Figure 6.4: A schematic representation of how the anti-unification and pro-unification operators are integrated in FCG's meta-layer architecture. When a diagnostic triggers a problem of the type 'no-match-or-solution' (shown in orange), the repair 'anti-unify-pro-unify' (shown inside the green box) will become active. It will loop though the different grammatical constructions of the construction-inventory and anti-unify them with the transient structure. The construction that could be anti-unified with the lowest cost is then pro-unified with the transient structure. This pro-unified construction is then unified with the transient structure, in order to create a new transient structure. Routine processing is then resumed and if that branch in the search tree leads to a solution, the pro-unified construction is consolidated by adding it to the construction inventory.

constructions for 'un' (*a*), 'dîner' (*dinner*) and 'formidable' (*splendid*), and an 'np-cxn' that groups an article, an adjective and a noun, in that order, into a noun phrase. The lexical constructions can apply to the utterance, but the 'np-cxn' cannot, as its word order constraints are in conflict with the word order that was observed. After the application of the lexical constructions, the 'no-match-or-solution' diagnostic instantiates a problem of the type 'no-match'. This is the case because no more grammatical constructions can apply and the meaning predicates in the transient structure are not connected into a single network. At the meta-layer, the 'anti-unify-pro-unify' repair becomes active. It loops through the different grammatical constructions in the construction inventory and anti-unifies them with the transient structure. The anti-unification cost is the lowest for the 'np-cxn', as only two variables need to be decoupled in the 'meets' constraints. The anti-unified 'np-cxn' is then pro-unified with the transient structure, binding again the decoupled variables to unit names. The pro-unified construction then applies to the transient structure and the resulting transient structure qualifies as a solution. Finally, the pro-unified construction is added to the construction inventory.

The transient structure and constructions involved in this example are shown in Figure 6.5. The transient structure for the observation "un dîner formidable" is shown at the bottom. The black circle highlights the 'meets' constraints that indicate that the article immediately precedes the noun and that the noun immediately precedes the adjective. At the left side, the existing 'np-cxn' is shown. The red circle highlights the 'meets' constraints that indicate that the article should immediately precede the adjective and that the adjective should immediately precede the noun. Because of these conflicting 'meets' constraints, the 'np-cxn' cannot apply to the transient structure. The anti-unified 'np-cxn' is shown at the top. The 'meets' constraints, highlighted by a grey circle, have been relaxed by variable decoupling and now contain two variables that are not coupled to unit names: 'art-27' and 'noun-42'. This construction can apply to the transient structure, but its 'meets' constraints do not constrain the word order any more. The pro-unified construction is shown at the right. The 'meets' constraints, highlighted by a green circle, have now been coupled again to unit names. They now require that the article immediately precedes the noun, and that the noun immediately precedes the adjective, as was observed in the utterance.

Note that, while the anti-unification process has relaxed the conflicting word order constraints in the construction, and while the pro-unification process has effectively incorporated constraints encoding the word order that was observed into the construction, the algorithms were not informed by any grammar- or feature-specific information. In fact, they did not even need to know that they were dealing with features that affect word order. It is this generality that makes anti- and pro-unification powerful as learn-

ing operators in emergent grammars, in which the the type and function of features is not known on beforehand and can evolve over time.

6.5 Conclusion

In this chapter, I have presented an operator that specialises FCG constructions towards observations captured in transient structures, as well as an integration of this specialisation operator and the generalisation operator presented in the previous chapter into FCG's meta-layer architecture. I have first argued that the specialisation of constructions towards observations is a crucial step in learning new constructions. It is complimentary to the generalisation step that was discussed in chapter 5 of this dissertation. While an impasse during processing can often be overcome by generalising over the conflicting element in the construction, the generalised construction is often too unconstrained to be added to the construction inventory. Therefore, a subsequent specialisation step that constrains the generalised construction again towards the observation is required. This construction can then be added to the construction inventory, reducing the risk of overgeneralisation.

Then, I have introduced a pro-unification operator that performs the specialisation of a construction towards a given transient structure. Like in the case of the anti-unification-based generalisation operator introduced in chapter 5, the pro-unification operator is very general. It works on symbolic structures only, and does not require any grammar-specific or problem-specific information. This makes the algorithm suitable for use in emergent grammars, in which the specific features and there functions are not known on beforehand and can evolve over time.

Finally, I have presented the integration of the anti-unification and pro-unification operators as a powerful repair in FCG's meta-level architecture. I have demonstrated the generality of this repair with an example in which an unknown word order occurred. The generalisation operator first created a construction in which the values of features that encode word order were generalised into free variables. After that, the specialisation operator created a construction in which these variables were bound to the correct unit names, as observed in the transient structure. Apart from the word order constraints, which it took from the observation, the resulting construction contained the exact same information as the original construction. It is worth noting that the generalisation and specialisation operator works without any knowledge about the features that it is handling. New word orders, agreement features, semantic categories and syntactic functions are all handled in exactly the same way.

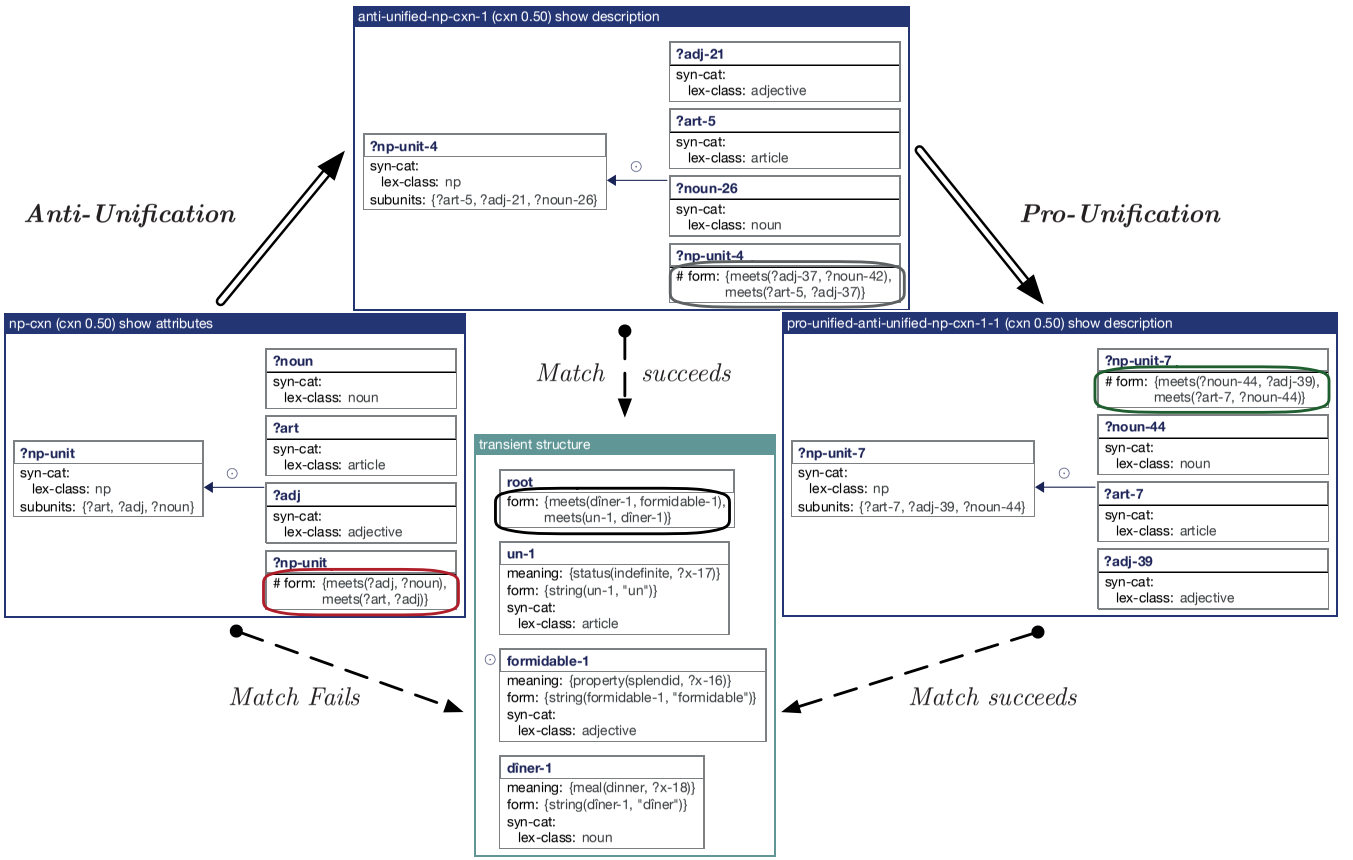


Figure 6.5: Relaxing and learning word order constraints through anti- and pro-unification. The cxn at the left cannot apply to the transient structure at the bottom because of conflicting word order constraints (art-noun-adj vs. art-adj-noun). They are first relaxed through anti-unification (cxn at the top) and then constrained through pro-unification (cxn at the right).

Chapter 7

Case Study: the Origins of Syntax

7.1 Introduction

This chapter presents a case study in which the representations and mechanisms that were introduced in the previous chapters are applied in a multi-agent experiment on the origins of syntax. I will focus in particular on the following aspects:

- The use of anti-unification as a general meta-level operator for expanding the coverage of a grammar by incrementally learning a type hierarchy.
- The learning advantages of capturing grammatical information in a grammar's type hierarchy instead of in its individual constructions.
- The learning advantages of shifting the competition between individual constructions to the links that connect different categories in the type hierarchy.

The case study consists in an experiment that studies the emergence and evolution of early grammatical patterns in a population of autonomous, artificial agents. The experiment follows the language game paradigm (Steels, 1995, 1997, 1998, 2012b), which is a well-established methodological framework for studying language emergence and evolution through agent-based models. Each agent in the experiment is equipped with the representations (Fluid Construction Grammar including type-hierarchies) and mechanisms (meta-layer diagnostics and repairs including anti-unification) that were introduced earlier in this dissertation. The experiment studies how a shared system of syntactic patterns can emerge and evolve in a population of autonomous agents. The primary function of the shared syntactic patterns is to minimize the referential ambiguity of the language in the world, leading to a more effective and efficient communication system.

The chapter is structured as follows. The first section presents a high-level overview of the experiment, sketches its background and specifies its main objectives (7.2). The second section provides a detailed description of the design and implementation of the experiment (7.3). The third section presents different learning strategies that the agents in the experiment use to introduce and adopt syntactic structures in their language (7.4). The last section compares the performance of the four strategies and discusses their results (7.5). Interactive visualisations of actual experimental runs are included in the web demonstration that accompanies this dissertation (`https://www.fcg-net.org/demos/vaneecke-phd`).

7.2 The Origins of Syntax

In human languages, utterances are not simple bags of words, but highly structured entities. There are two main mechanisms that are crucial for structuring utterances. The first mechanism concerns the *linear ordering* of the elements that constitute the utterance. In most languages, words, phrases and morphological entities such as affixes do not appear in a random order[1]. In English for example, nouns precede their derivational affixes (e.g. 'luck' + 'y') and main verbs follow their auxiliaries (e.g. 'has' + 'spoken'). The second mechanism for structuring utterances concerns the use of *markers* that are shared between different elements in the utterance. In Latin for example, it is the marking system, more strongly than the word order, that structures utterances such as 'Mari-*a* pulchr-*am* puell-*ae* ros-*am* dat. (*'Maria gives the girl a beautiful flower'*)'.

The existence of word order and markers and their importance in structuring utterances is acknowledged in probably all theories of language, and has been extensively studied in the linguistic literature (for a high-level overview, see Valin and LaPolla, 1997; Blake, 2001; Cinque and Kayne, 2005; Corbett, 2006; Malchukov and Spencer, 2009). Their origins however, remain heavily debated. The view that was dominant until two decades ago argues that these structures are innate and therefore constitute a stable universal grammar that underlies all natural languages (Chomsky, 1986). The opposing view argues that grammar is not innate or a priori present in the human brain, but that it is a dynamic system that emerges through the communicative interactions of interlocutors (Hopper, 1987; Jasperson et al., 1994). The experiment that we describe in this chapter contributes to the latter view, as it presents a model of how word order can emerge and evolve in a population of artificial agents, through repeated communicative interactions.

Previous experiments on the emergence of syntactic structures have often focussed on the learner's bias, also called induction bias or generalisation bias (Batali, 1998; Kirby, 1999, 2002b; Briscoe, 2000). These experiments aim to show that syntactic structures are introduced by language learners, whose learning algorithms are biased towards generalising and structuring any input that they get. In this view, generations of learners iteratively impose more structure and regularity onto the language, until the system stabilises. Steels and Garcia Casademont (2015a) on the other hand, argue that the emergence of syntactic structures is motivated by the need to dampen the combinatorial explosions that arise when comprehending and interpreting utterances in

[1] In many languages, the linear order these elements is considered so important that they are often classified according to it: prefixes, infixes and suffixes; prepositions, circumpositions and postpositions; prenominal and postnominal modifiers; preverbal and postverbal subjects; etc.

the world. Their experiments show that shared syntactic structures can emerge in a population of agents in a single generation, based on the outcome of communicative interactions that drive the "stepwise invention, adoption and alignment of linguistic conventions" (Steels and Garcia Casademont, 2015a, p. 38).

The experiment that is presented in this case study builds further on the findings of Steels and Garcia Casademont (2015a). First, it replicates the three baseline strategies that are introduced in their paper, but provides a more detailed analysis of the results. Then, it presents an improved version of the fourth, more realistic strategy. Apart from confirming the results of Steels and Garcia Casademont (2015a), and presenting a more powerful learning strategy, the case study also shows that the framework of representations and mechanisms introduced in this dissertation allows a more general and elegant implementation of this kind of evolutionary linguistics experiment.

7.3 Experimental Design and Implementation

I have implemented the experiment in Common Lisp using the Babel2 framework[2] (Loetzsch et al., 2008). Babel2 is an open source software library that groups different technologies that were specifically designed to be used in agent-based experiments on the emergence and evolution of language. Specifically, the experiment employs Babel2's modules for running multi-agent interactions (*experiment framework*), for language comprehension and production (*Fluid Construction Grammar*), for learning (*meta-layer learning*), and for tracking and visualising the dynamics and results of the experiment (*monitors* and *web interface*).

The experiment consists of three basic components, of which the design and implementation are presented in the following three sections: the *world* (7.3.1), the *population* (7.3.2), and the *interaction script* (7.3.3). Once these three parts are in place, the agent-based model is ready to run. However, a shared language will only start to emerge and evolve when the agents are equipped with appropriate learning mechanisms (7.4).

7.3.1 World

The world of the experiment consists of a number of *objects*. The objects can be perceived through a number of *dimensions*, which can have different *values*. The possible dimensions and values are completely open-ended. For clarity reasons however, the objects in the version of the experiment described here are geometrical figures,

[2] `https://github.com/EvolutionaryLinguisticsAssociation/Babel2`

which are made up of dimensions such as *shape*, *color* and *size*. The values for these dimensions can for example be 'square' or 'circle' for shape, 'red' or 'blue' for color and 'small' or 'large' for size. The world itself consists of all unique objects that can be formed based on the given dimensions and values. The number of objects in the world is thus equal to v^d, in which d stands for the number of dimensions and v stands for the number of possible values per dimension. An example world is shown in Figure 7.1. In this world, the objects have three dimensions with two values each (shape: square, circle; color: red, blue; size: small, huge), which means that the world consists of 8 unique objects (2^3).

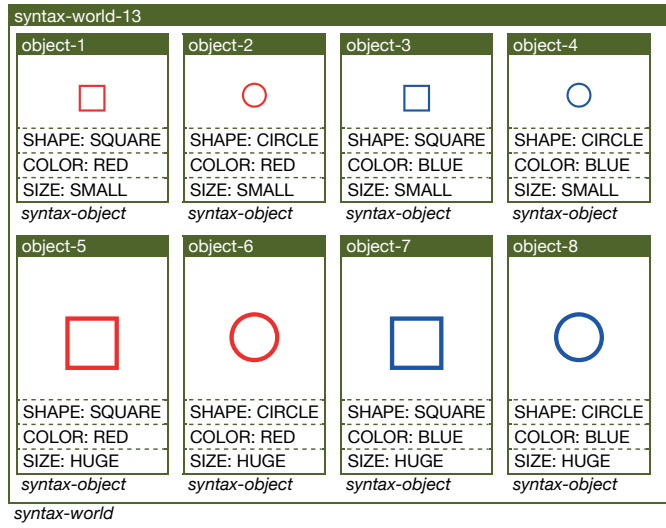


Figure 7.1: An example world: three dimensions (shape, color and size) with two possible values (square-circle, red-blue and small-huge) yields a world of 8 unique objects.

Unless otherwise indicated, the world in the experimental runs described in this chapter consists of 64 objects, with the dimensions and values specified in Table 7.1. The ontology of dimensions and values to be used for generating the world can be specified by setting the `:ontology` key in the configuration of the experiment.

7.3.2 Population

The population of the experiment consists of a number of *autonomous agents*. The agents are either embodied in physical robots or simulated in software. They are autonomous in the sense that each individual agent perceives the world through its own sensors (vision and hearing) and acts upon the world using its own actuators (speech

Shape	Color	Size
Square	Red	Tiny
Circle	Blue	Small
Rectangle	Yellow	Large
Triangle	Green	Huge

Table 7.1: The default dimensions and values used for creating a world of 64 objects. Unless otherwise indicated, this is the world used in the experimental runs presented in this chapter.

and pointing). Each agent has its own grammar, represented by an FCG construction inventory. At the beginning of the experiment, the inventory contains all lexical constructions that are needed to communicate about the dimensions of the objects in the world, but not yet any grammatical constructions. Each lexical construction maps between a particular value for a dimension that occurs in the world (e.g. square(?x)) and a Dutch word (e.g. 'vierkant')[3]. An example of a lexical construction is shown in Figure 7.2. This construction maps between the string "blauw" and its meaning 'blue(?x)'. The value of the lex-class feature is a unique symbol for each construction of each agent and will later be used when creating grammatical patterns. The number of agents in the population can be specified by setting the `:population-size` key in the configuration of the experiment. Unless otherwise specified, the experiments described below are run with a population of 10 agents.

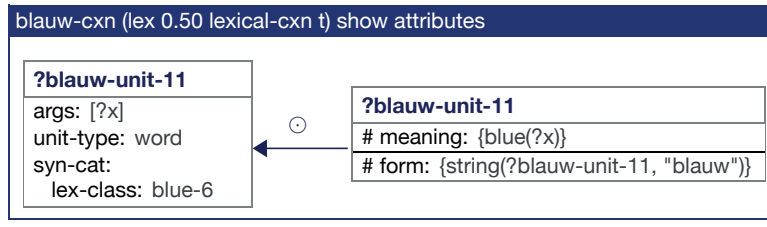


Figure 7.2: A lexical construction that maps between the string "blauw" and the meaning "blue(?x)". Each lexical construction of each agent has a unique symbol as the value of its lex-class feature.

[3]In this experiment, I focus on the emergence and evolution of early grammar only, and assume that a shared vocabulary is already in place at the start. There has been a large body of previous experiments that have studied the concrete mechanisms through which this kind of shared vocabulary can emerge and evolve in a population of agents. For an overview, see Steels (2015).

7.3.3 Interaction Script

The agents in the population participate in repeated communicative interactions, in which one agent (the speaker) tries to draw the attention of another agent (the hearer) on a number of objects in their world. These communicative interactions take place according to a fixed interaction script. A single communicative interaction involves the 6 steps described below. A schematic visualization of the interaction script is shown in Figure 7.5.

1. Agent and Role Selection (speaker and hearer)

Two agents are randomly selected from the population. One agent is randomly assigned the role of *speaker*, the other agent is assigned the role of *hearer*.

2. Scene Selection (speaker and hearer)

The two agents are placed in a *scene*. The scene consists in a random subset of the objects in the world. The speaker and hearer can directly perceive these objects, and only these, during the communicative interaction. The minimum and maximum number of objects in a scene can be specified using the `:min-nr-of-objects-in-scene` and `:max-nr-of-objects-in-scene` keys in the configuration of the experiment. The default values are 1 for the minimum number of objects in the scene and 64 (the number of objects in the default world) for the maximum. All objects in the scene are unique, in the sense that they differ in at least one dimension. Figure 7.3 shows an example of a scene, generated based on the world shown in Figure 7.1. The scene contains four objects: a small blue square, a small red circle, a huge blue square and a huge red square.

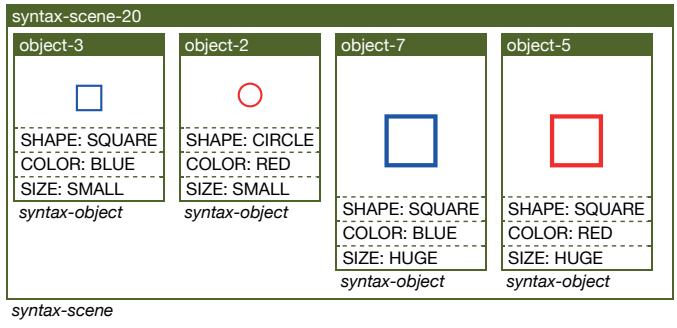


Figure 7.3: An example scene of four objects, based on the world from Figure 7.1.

3. Topic Selection and Conceptualisation (speaker)

The speaker randomly selects the *topic* of the interaction. The topic consists of one or more objects in the scene. It will be the task of the speaker to draw the attention of the hearer to these objects by describing them using language. In order to do this, the speaker needs to conceptualise the topic, i.e. to come up with a meaning - a selection of information - that he will convey to the other agent. Just like in human communication, the speaker will be economical. He will not describe all properties of the objects to which he refers, but only those that are necessary (cf. the Gricean maxim of quantity (Grice, 1989)). In order to conceptualise the topic, the speaker computes the minimal set of features that distinguishes the objects in the topic from all other objects in the scene. Then, these discriminatory features are transformed into a semantic network that represents the meaning that the speaker will convey to the hearer.

The left side of Figure 7.4 shows a topic that is drawn from the scene shown in Figure 7.3. The topic consists here of two objects. The possible number of objects in the topic is bounded by the `:min-nr-of-objects-in-topic` and `:max-nr-of-objects-in-topic` parameters in the configuration of the experiment. The default values are 1 for the minimum number of objects in the topic and 2 for the maximum. The discriminatory features, i.e. the features that distinguish the objects from all other objects in the scene, are highlighted in bold. For the small blue square and the huge red square respectively, {small(object-3), blue(object-3)} and {red(object-5), huge(object-5)} are sufficient to unambiguously identify these objects in the scene. The corresponding semantic network is shown at right side of the Figure.

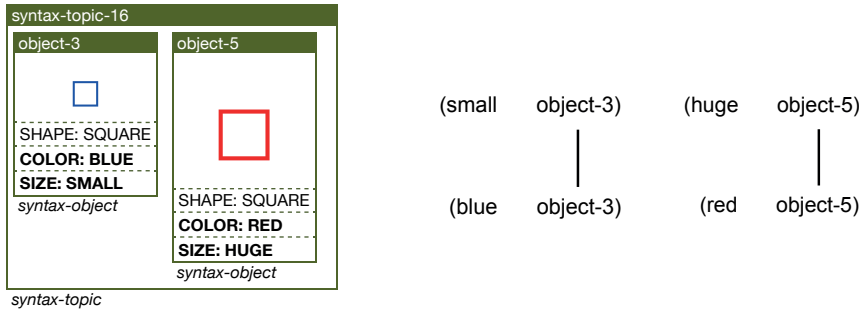


Figure 7.4: The left side of the Figure shows an example topic with two objects, drawn from the scene presented in Figure 7.3. The minimal set of discriminatory features, i.e. features that distinguish the objects from all other objects in the scene, is highlighted in bold. The right side of the Figure shows the corresponding semantic network that will be conveyed to the hearer.

4. Formulation (speaker)

The speaker now formulates an utterance that expresses the semantic network that was the result of the conceptualisation process. He formulates the utterance using the constructions in his FCG grammar (cf. Section 3.4.5). After each construction application, a goal test checks whether the resulting transient structure qualifies as a solution (cf. Section 3.4.6). The goal test is here a re-entrance test (Steels, 2003). Conceptually, re-entrance consists in the speaker reflecting whether he would have been able to correctly comprehend and interpret the utterance if he would have been the hearer. Technically, after each construction application, the speaker will use his grammar to comprehend the utterance that he has constructed so far, and interpret the resulting meaning representation in the scene. If the interpretation of the meaning representation in the scene only yields a single hypothesis, unambiguously identifying the objects in the topic, the current node in the search tree is considered a solution, and the utterance is passed on to the hearer. If the search tree has been exhaustively explored and no solution has been found, a diagnostic signals the problem, which triggers a jump to the meta-layer. At the meta-layer, the learning mechanisms described in Sections 7.4.1 to 7.4.4 will become active and repair the problem by adding grammatical constructions or type hierarchy links that disambiguate the utterance. Then, routine processing continues and when the goal test succeeds, the resulting utterance is passed on to the hearer.

5. Comprehension and interpretation (hearer)

The hearer perceives the utterance formulated by the speaker and parses it using his own FCG grammar. After each construction application, a goal test checks (i) whether all strings have been processed and if so, (ii) whether the meaning extracted from this node unambiguously identifies a number of objects in the scene. This is done by unifying the semantic network with the agent's model of the scene. If the unification returns only one set bindings, the goal test succeeds and the hearer points to the objects that correspond to these bindings. If the complete search tree has been explored and no solution has been found, the hearer signals to the speaker that he could not understand the utterance.

6. Feedback and Alignment (speaker and hearer)

After the hearer has either pointed to a number of objects or signalled that he could not understand the utterance, the speaker gives feedback to the hearer. Two cases

need to be distinguished:

- If the hearer pointed to the right objects, the speaker signals success. Both the speaker and the hearer will reward the constructions or links in the type-hierarchy that they have used, and punish competing constructions and links. The exact updating rule and definition of competitors depends on the concrete learning mechanisms that are used, and will be discussed in more detail below.
- If the hearer did not point to the right objects, the speaker signals failure and points to the actual objects that form the topic. The speaker will punish the constructions or links in the type hierarchy that he has used. Based on the feedback, the hearer learns one or more grammatical constructions or type hierarchy links that disambiguate the utterance in the current scene.

7.4 Learning Strategies

With the world, the population and the interaction script in place, the language game is ready to be played in a multi-agent experiment. However, the agents are at this point not yet equipped with any mechanisms for inventing and adopting syntactic structures. In the next sections, four different learning strategies are introduced. Each strategy provides the agents with increasingly more powerful invention and adoption mechanisms.

7.4.1 Lexical Strategy

The first strategy is called the *lexical strategy*. The lexical strategy does not put any syntactic constraints on the words that constitute the utterances and only relies on the words' lexical meaning. This strategy serves as a baseline for the experiment, as it investigates the properties of a non-syntactic language, to which the other strategies will later be compared.

Diagnostics and Repairs

The agents are not provided with any diagnostics and repairs, as they do not need to invent or adopt grammatical structures. In formulation, the speaker just applies his lexical constructions to all meaning predicates that need to be expressed, and utters the corresponding word forms in any order. The hearer will comprehend the

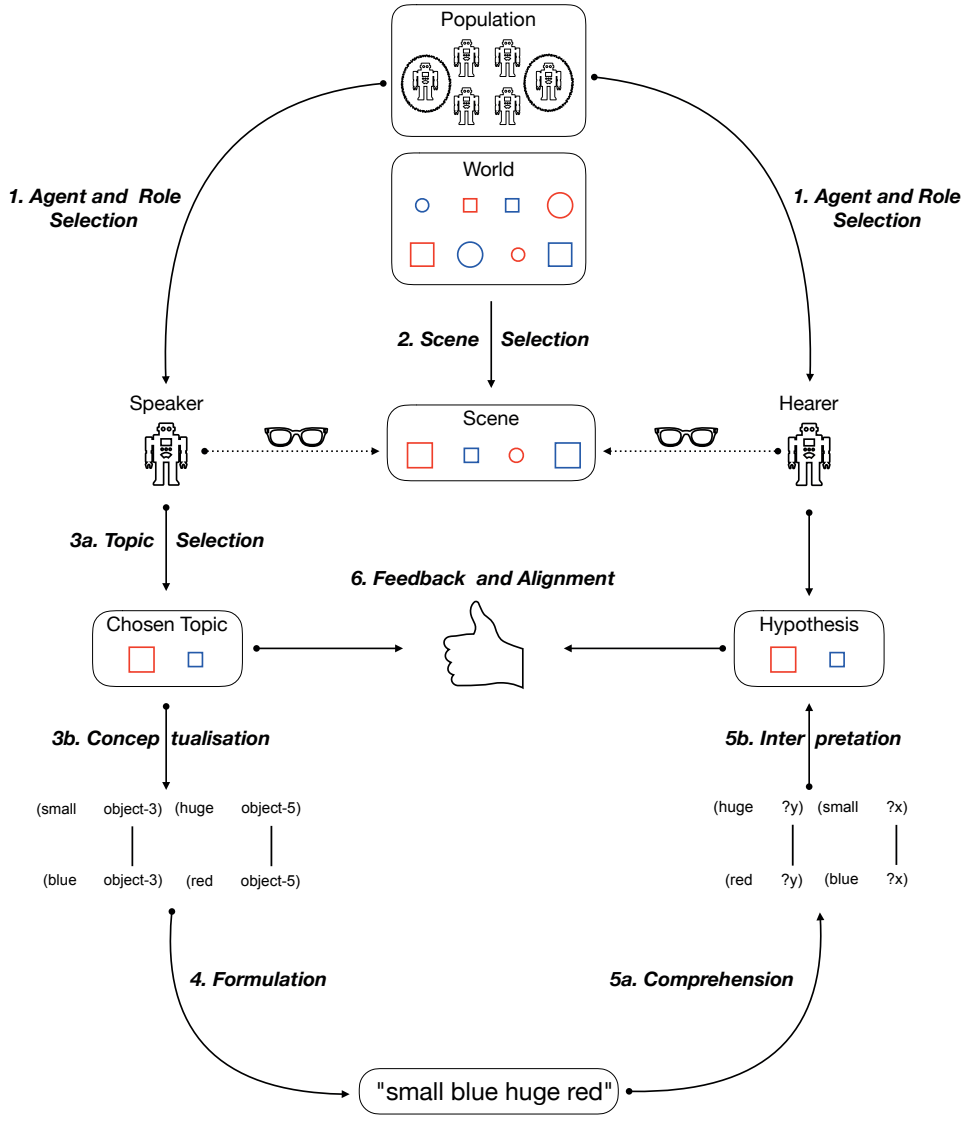


Figure 7.5: A schematic visualization of the interaction script that is used in the origins-of-syntax experiment. The six steps are explained in detail in Section 7.3.3.

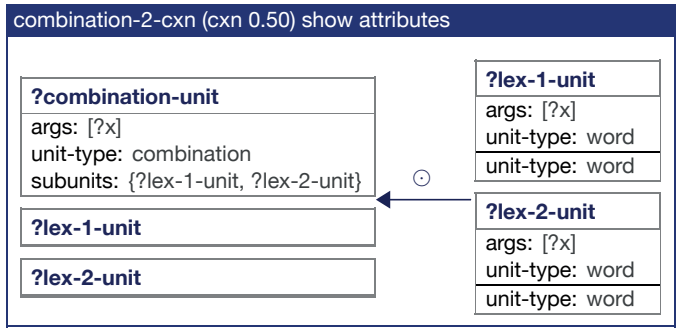


Figure 7.6: An example of a combination construction used in the lexical strategy. The construction matches on two lexical items, which can occur anywhere in the utterance and can have any 'lex-class' value.

utterance of the speaker using his lexical constructions, and make hypotheses about the bindings of the variables in the resulting meaning network. Each time that a hypothesis is generated, it is immediately tested. When its unification with the world returns exactly one set of bindings, the hearer points to the corresponding objects. In order to construct the hypotheses, we add two constructions to the construction inventories of the agents. These constructions are called *combination constructions* and their only purpose is to generate hypotheses about variable bindings. An example of a combination construction is shown in Figure 7.6. It matches on two units, with the only constraint that the units need to be lexical units, as expressed through the 'unit-type' feature, and that they need to have the same referent, as expressed through the 'args feature'. There are no constraints on where the words corresponding to these units are located in the utterance.

Experimental Results

Figure 7.7 shows the results of a simulation in which the agents make use of the lexical strategy. The graph aggregates over the outcome of all of games played by all agents in the experiment. The x-axis represents the temporal dimension of the experiment, with the ticks indicating the average number of games that an individual agent has played at that point in time (either as speaker or hearer). The turquoise line indicates on the left y-axis the extent to which the games were successful. Communicative success is recorded as 1 (success) or 0 (failure) after each game, and is averaged over the last 250 games. The graph shows that the average communicative success stays at around 0.9 throughout the experiment. This means that the lexical strategy leads to communicative success in 9 out of 10 games, whereas in 1 out of 10 games,

the lexical strategy is not sufficient. The green line indicates the coherence of the language, also on the left x-axis. The coherence is registered as 1 if for each object in the scene, the hearer would have expressed the same meaning network as the one expressed by the speaker using the exact same utterance, and as 0 otherwise. The coherence is averaged over the last 500 games. The graph shows that in average only 20% of the utterances were coherent, which is no surprise as the lexical strategy does not impose any constraints on the word order. The yellow line indicates the average number of grammatical constructions in the construction inventories of the agents. Using the lexical strategy, their construction inventories contain at any moment only the two combination constructions that were provided to them at the beginning of the experiment. Communicative success, coherence and number of grammatical constructions all remain constant over time, as there is no learning going on in the agents.

While comprehending an utterance, the hearer will navigate through the search space created by all grammatical constructions that can be applied. He will apply one construction at a time and each time, a goal test will check whether the parsed meaning has a single interpretation in the scene. The more ambiguous the language is, the larger the search space will be. I will quantify here the ambiguity of the language given the communicative task in terms of search effort, calculated by dividing the total number of grammatical constructions that were applied (i.e. the total number of nodes in the search tree) by the number of grammatical constructions that were applied in the path to the solution (i.e. the depth of the solution node). A graph visualising this measure for the lexical strategy is shown in Figure 7.8. Only successful games are included and the values are averaged using a sliding window of 500 interactions. The graph shows that the average size of the search space remains constant throughout the experiment at about 11, which means that only 1 in 11 construction applications brings the agent closer to the solution.

The huge search space is due to the enormous *referential ambiguity* of the language. The referential ambiguity of an utterance in a scene corresponds to the number of possible interpretations in that scene, of the semantic network that results from the hearer's comprehension process of that utterance. In other terms, it corresponds to the number of different sets of objects in the scene that can be described using the utterance. While the exact number depends on the specific scene, this number directly correlates with the number of different variable bindings that can be made between the predicates in the semantic network. When using the lexical strategy, the utterance "blue square large" for example, allows for the 5 different possibilities listed in (2) to (6) below.

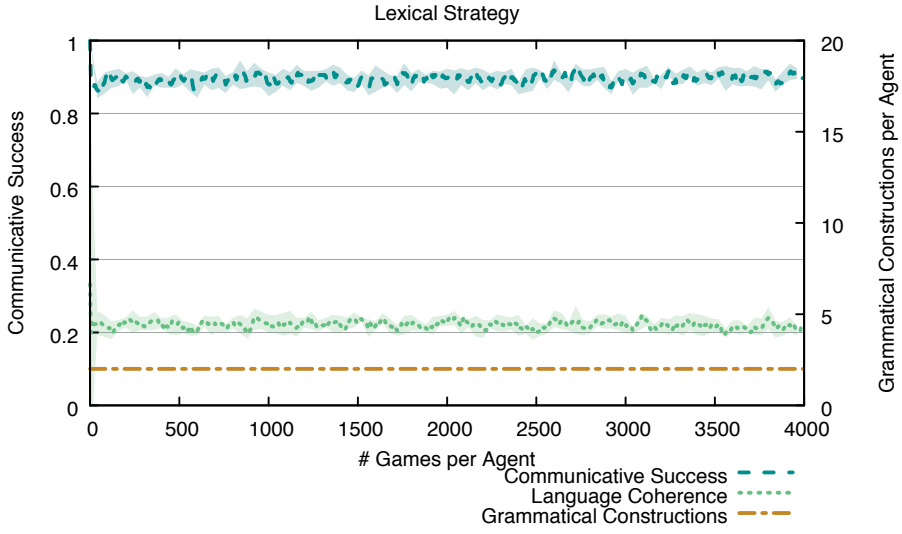


Figure 7.7: Results of a simulation in which the agents use the lexical strategy. The turquoise line indicates the communicative success (left y axis), the green line indicates the language coherence, and the yellow line indicates the average number of combination constructions in the inventories of the agents.

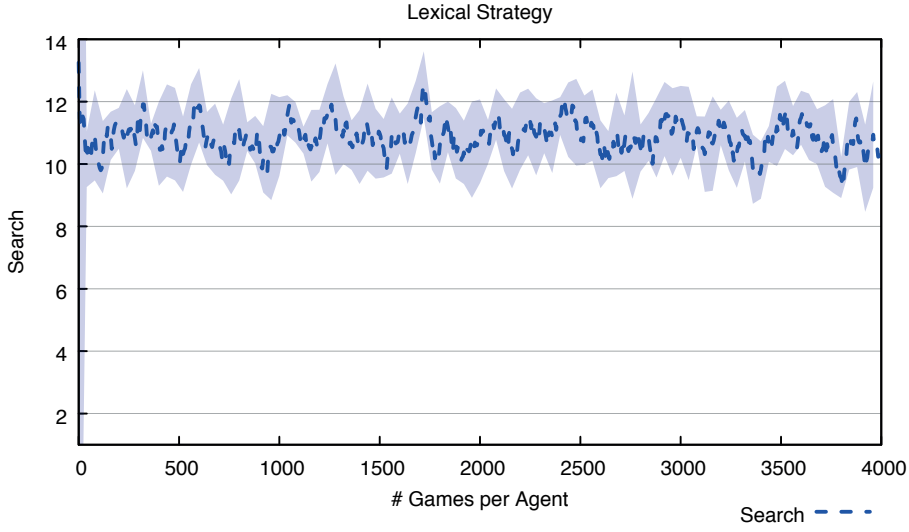


Figure 7.8: Size of the search space that was explored using the lexical strategy. The y-axis indicates that total number of nodes created by combination-cxns, divided by the depth of the solution node.

(2) {blue(?obj-1), square(?obj-2), large(?obj-3)}

(3) {blue(?obj-1), square(?obj-2), large(?obj-1)}

(4) {blue(?obj-1), square(?obj-1), large(?obj-2)}

(5) {blue(?obj-1), square(?obj-2), large(?obj-2)}

(6) {blue(?obj-1), square(?obj-1), large(?obj-1)}

In general, the number of possible hypotheses corresponds to the number of possible partitions of a set with as cardinality the number of variables in the semantic network, which, in this experiment, also corresponds to the number of words in the utterance. This number for a set of cardinality n is called the n^{th} Bell number and can be calculated using the formula in (7.1), in which $\left\{ {n \atop k} \right\}$ stands for the stirling number of the second kind, i.e. the number of ways in which a set of cardinality n can be partitioned into k non-empty subsets.

$$B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\} \tag{7.1}$$

The Bell number grows double exponentially, which means that while an utterance of 3 words yields 5 possiblities, utterances of 6, 9 and 12 words yield 203, 21147 and 4213597 hypotheses respectively. Although the world certainly imposes certain restrictions on the hypotheses (e.g. the same objects cannot be large and small at the same time), it is clear that an efficient communication system will require an effective way to deal with this combinatorial explosion.

7.4.2 Grouping Strategy

The *grouping strategy* offers a first way to dampen the referential ambiguity of the language by introducing constraints on the linear ordering of the words that constitute the utterances. The grouping strategy implies that words that refer to the same object in the scene are linearly grouped together in the utterance. The order of the groups and of the words within a group is not fixed. For example, the utterance corresponding to {blue(obj-1), square(obj-2), large(obj-1)} could be formulated as "blue large square", "large blue square', "square blue large" or "square large blue". "Blue square large" and "large square blue" would not be allowed, as blue and large have the same referent, but do not belong to the same group in the utterance. Grouping together the co-referent words reduces the number of possible variable bindings within a semantic network

from the double exponentially growing Bell numbers to the simple exponential 2^{n-1}, in which n stands for the number of variables in the network. Not considering ontological restrictions, utterances of 3, 6, 9 and 12 words now correspond to 4, 32, 256 and 2048 possible semantic networks. For example, the three-word utterance "blue square large" that was discussed in the previous section now gives rise to the four semantic networks shown in (2), (3), (5) and (6) above. When the grouping strategy is used, the utterance is not compatible with the semantic network shown in (4), as blue and large are not grouped together in the utterance and can therefore not have the same referent.

Diagnostics and Repairs

In order to be able to use the grouping strategy, the agents in the experiment need to be endowed with the necessary mechanisms to invent and adopt constructions that group co-referent words. These mechanisms are implemented in the form of meta-level diagnostics and repairs. There are slight differences between the diagnostics and repairs that an agent uses when he is the speaker or when he is the hearer.

- **Speaker:** A diagnostic checks whether the formulation process leads to a solution, as defined by the re-entrance goal test described in Section 7.3.3 above. If no solution is found, the diagnostic triggers a jump to the meta-layer. At the meta-layer, a repair creates the necessary grammatical constructions, called *grouping-cxns*. The grouping-cxns ensure that co-referent words are grouped together in the formulated utterance. They are added to the construction inventory of the speaker so that the formulation process can continue. Formulation will now succeed and the resulting utterance is passed on to the hearer.

- **Hearer:** In the case of the hearer, a diagnostic becomes active at the very end of the interaction, after feedback has been provided by the speaker. The diagnostic checks whether the interaction was successful, and if this was not the case, it triggers a jump to the meta-layer. At the meta-layer, a repair will create new grouping-cxns based on the utterance and the topic that was provided as feedback by the speaker. The constructions are then added to the construction inventory of the hearer.

An example of a grouping construction is shown in Figure 7.9. This construction groups together two lexical units. One unit matches on the lex-class 'green-19' and the other on the lex-class 'large-19'. These lex-classes correspond to the lex-classes of the units that were created by the agent's lexical constructions (cf. Figure 7.2), and are thus used to match on specific words. The construction ensures that the value of the 'args'

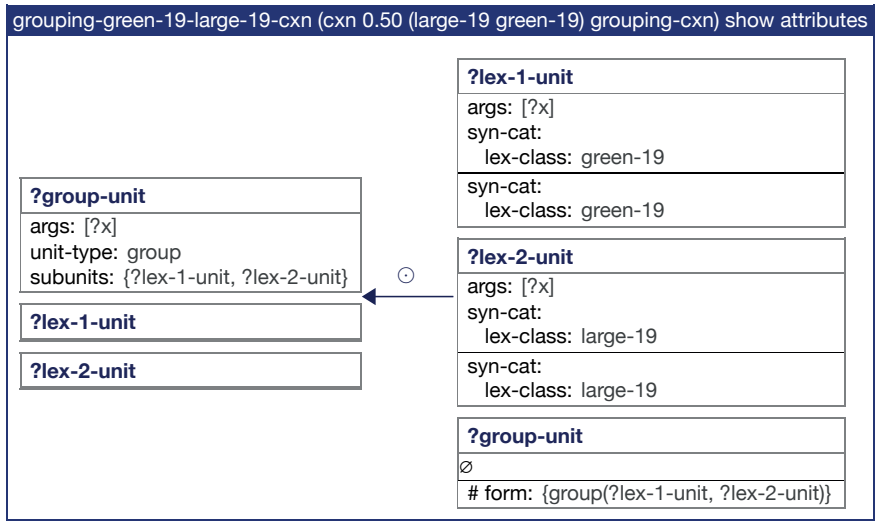


Figure 7.9: An example of a grouping construction that matches on two lexical items (lex-classes 'green-19' and 'large-19'). In formulation, the construction ensures that the two co-referent words are grouped together in the utterance. In comprehension, it ensures that the two lexical items in the group are co-referent.

feature of the two units is the same, meaning that they refer to the same object. The 'group' constraint in the form feature of the group unit ensures that the two lexical items occur in the same group in the utterance, i.e. they cannot be separated by a word that refers to a different object. Grouping constructions can group any number of co-referent words.

The number of constructions that is needed to be able to group each combination of values for any number of dimensions can be computed using the formula shown in (7.2), in which a stands for the number of dimensions of the objects in the world, and v stands for the number of values that each dimension can have. For the experiment described here ($a = 3$ and $v = 4$), this means that 112 different grouping constructions are needed.

$$\sum_{n=2}^{a} \binom{a}{n} \cdot v^n \tag{7.2}$$

Experimental Results

Figure 7.10 shows the results of a simulation in which the agents make use of the grouping strategy. The turquoise line indicates again the communicative success on the

left y-axis. The communicative success starts at 0.1 [4] and increases until it reaches 1 after about 800 interactions per agent. From this moment on, all games succeed. The yellow line indicates on the right y-axis the average number of grouping constructions in the construction inventories of the agents. The number of grouping constructions starts at 0 and climbs to 112, which is indeed the number predicted by the formula in (7.2) ($112 = \sum_{n=2}^{3} \binom{3}{n} \cdot 4^n$). The green line indicates the coherence of the language on the left x-axis. The coherence starts at 0 and reaches 0.3 by the time that the communicative success is at its maximum and remains constant after that. It is not surprising that the coherence stays low, as the grouping constructions do not impose an order on the lexical items that they group together.

The language that emerges when the agents employ the grouping strategy is powerful enough to achieve full communicative success in the experiment. This means that given an utterance, a scene and a complete set of grouping constructions, the utterance can unambiguously be interpreted in the scene. It does however not mean that there is no ambiguity in the language any more. In order to illustrate this, let us return to the scene and topic that were shown in 7.3.3. The scene consisted of a small blue square, a small red circle, a huge blue square and a huge red square. The topic consisted of a small blue square and a huge red square. The minimal set of discriminatory features computed by the speaker was {small(object-3), blue(object-3), huge(object-5), red(object-5)}. One possible formulation of this semantic network using the grouping strategy is "small blue huge red". For the hearer, the grouping strategy could in theory lead to $2^{(4-1)} = 8$ possible semantic networks, as illustrated in (7) - (14). In practice however, the grouping constructions that would license (12) - (14) will never be created by an agent. The reason for that is that they are ontologically impossible, e.g. an object cannot be red and blue at the same time. As a consequence, they will never lead to communicative success and will never be created by a repair. Of the remaining five hypotheses, four ((12) - (14)) lead to multiple interpretations in the world and will thus not be considered a solution by the agent. Only hypothesis (11) leads to a single set of bindings in the scene, and is thus considered a solution.

(7) {small(?obj-1), blue(?obj-2), huge(?obj-3), red(?obj-4)}
→ *multiple sets of bindings*

(8) {small(?obj-1), blue(?obj-2), huge(?obj-3), red(?obj-3)}
→ *multiple sets of bindings*

[4] The communicative success starts at 0.1 because about 10% of the utterances do not express more then 1 dimension of the objects in the topic and can, as a consequence, be comprehended using the lexical constructions only.

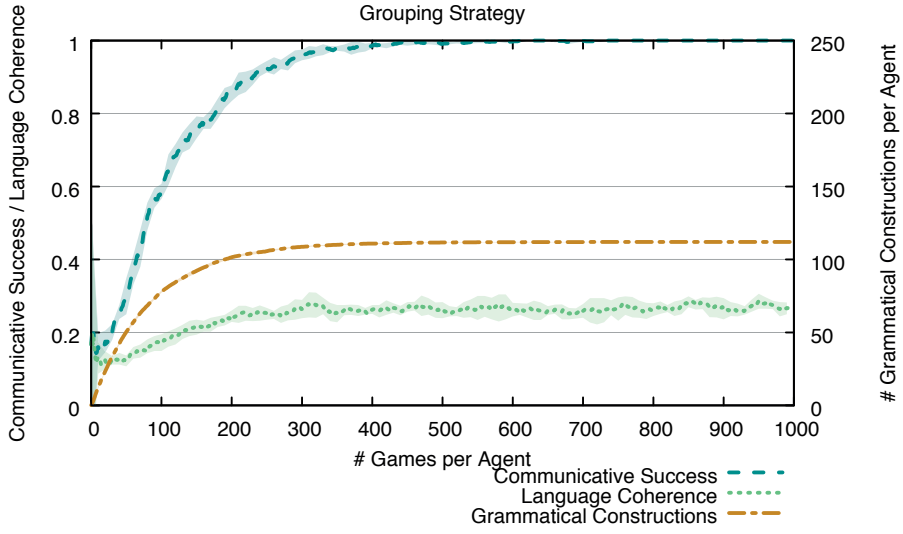


Figure 7.10: Results of a simulation in which the agents use the grouping strategy. The turquoise line indicates the communicative success (left y axis), the green line indicates the language coherence, and the yellow line indicates the average number of grouping constructions in the inventories of the agents.

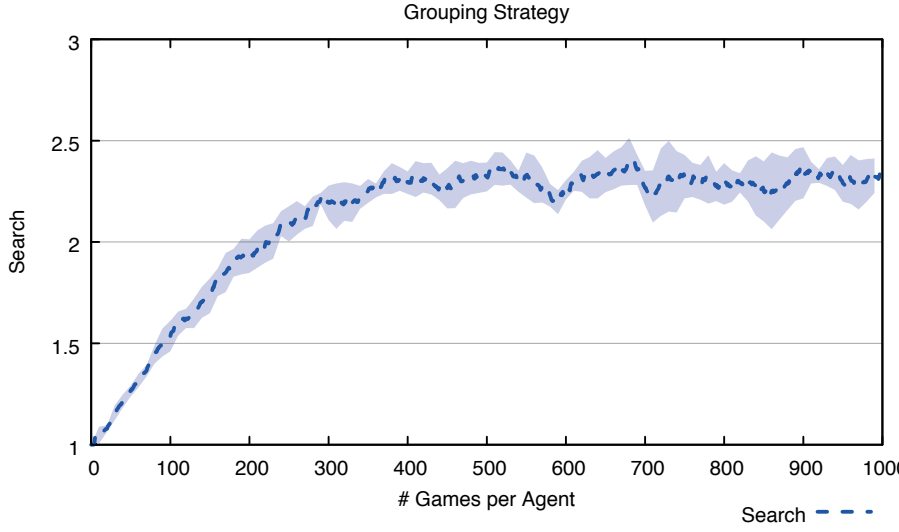


Figure 7.11: Size of the search space that was explored while applying the grouping-cxns. The y-axis indicates that total number of nodes created by grouping-cxns, divided by the depth of the solution node.

(9) {small(?obj-1), blue(?obj-2), huge(?obj-2), red(?obj-3)}
→ *multiple sets of bindings*

(10) {small(?obj-1), blue(?obj-1), huge(?obj-2), red(?obj-3)}
→ *Multiple sets of bindings*

(11) {small(?obj-1), blue(?obj-1), huge(?obj-2), red(?obj-2)}
→ **Possible Solution**

(12) {small(?obj-1), blue(?obj-2), huge(?obj-2), red(?obj-2)}
→ *Ontologically impossible - incompatible with the world*

(13) {small(?obj-1), blue(?obj-1), huge(?obj-1), red(?obj-2)}
→ *Ontologically impossible - incompatible with the world*

(14) {small(?obj-1), blue(?obj-1), huge(?obj-1), red(?obj-1)}
→ *Ontologically impossible - incompatible with the world*

Figure 7.11 shows the search effort required for processing the grouping strategy. The graph shows that the amount of search needed starts at 1 (no search) and increases with the growing number of grouping constructions that are created. When the maximum number of constructions is reached, the amount of search needed stabilises at about 2.4.

7.4.3 N-gram Strategy

The third strategy is called the *n-gram strategy*. The n-gram strategy does not only group together the co-referent words within an utterance, like the grouping strategy, but also imposes a sequential order on the words inside the group. This has the potential advantage of further reducing the number of parsing hypotheses, as the borders between the groups become clearer. For example, imagine that an agent hears the utterance "square large blue". The baseline lexical strategy would give rise to the five hypotheses shown in (15) - (19). Imagine now that the agent's construction inventory contains constructions that can combine "large blue square", "large blue", "large square" and "blue square". Using the grouping strategy, in which the groups are internally unordered, only hypotheses (15) - (18) remain. When the n-gram strategy is used, in which the words are ordered, only hypotheses (15) and (16) remain. Hypotheses (17) and (18) are ruled out, as they are not licensed by the constructions of the agent's inventory, e.g. (17) would require a construction for "square large", but only a construction for "large square" is available. N-gram constructions have thus the

potential to reduce the referential ambiguity of the language to a greater extent then grouping constructions.

(15) {square(?obj-1), large(?obj-2), blue(?obj-3)}
→ *Lexical Strategy, Grouping Strategy, N-gram Strategy*

(16) {square(?obj-1), large(?obj-2), blue(?obj-2)}
→ *Lexical Strategy, Grouping Strategy, N-gram Strategy*

(17) {square(?obj-2), large(?obj-2), blue(?obj-1)}
→ *Lexical Strategy, Grouping Strategy*

(18) {square(?obj-1), large(?obj-1), blue(?obj-1)}
→ *Lexical Strategy, Grouping Strategy*

(19) {square(?obj-1), large(?obj-2), blue(?obj-1)}
→ *Lexical Strategy*

Diagnostics and Repairs

The diagnostics and repairs that the agents need in order to be able use the n-gram strategy closely resemble those that they needed for using the grouping strategy. The only difference lies in the grammatical constructions that are created by the repairs. The 'group(?lex-1-unit, ?lex-2-unit)' constraints, which ensured that the two lexical items appeared in the same group in the utterance, are now replaced by 'meets(?lex-1-unit, ?lex-2-unit)' constraints. This kind of constraint ensures that the lexical item that matches '?lex-1-unit' will appear immediately left-adjacent to the lexical item that matches '?lex-2-unit'. Constructions that match on three units will have two meets constraints, namely 'meets(?lex-1-unit, ?lex-2-unit)' and 'meets(?lex-2-unit, ?lex-3-unit)' . When the speaker needs to invent a new construction, the order of the lexical items is chosen randomly. When the hearer needs to create a new construction, it will adopt the word order that was observed in the speaker's utterance. An example of an n-gram construction for "large red" is shown in Figure 7.12.

The minimum number of n-gram constructions that is needed to be able to express any number of values for any number of dimensions, is the same as the minimum number of grouping constructions that was needed (cf. formula (7.2)). However, more constructions are possible now, as for example "large square" and "square large" would be covered by the same grouping construction, but give rise to two different n-gram constructions. The maximum number of possible n-grams is bounded by the

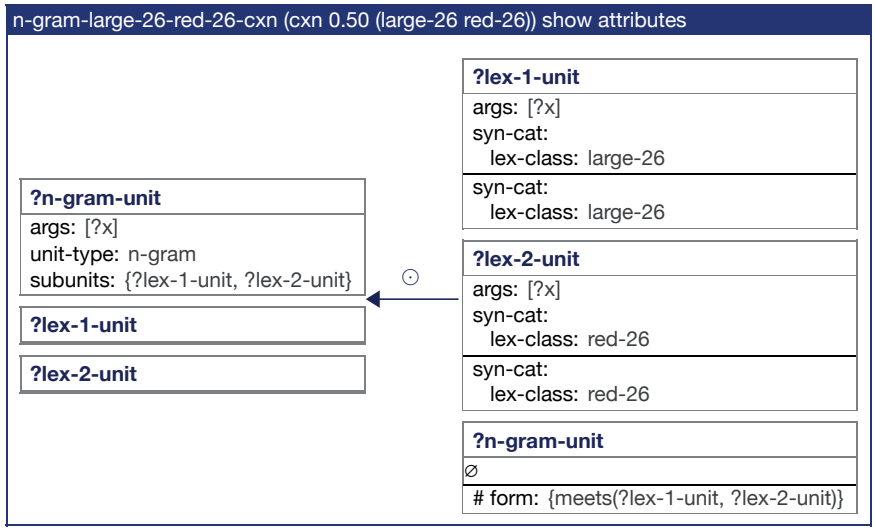


Figure 7.12: An example of an n-gram construction that matches on two lexical items (lex-classes 'large-26' and 'red-26'). The 'meets' constraint imposes that the '?lex-1-unit' immediately precedes the '?lex-2-unit' and the shared variable equality in the 'args' feature imposes the to lexical items to be co-referential.

formula in (7.3), in which a stands for the number dimensions and v for the number of possible values per dimension. In our experiment, $a = 3$ and $v = 4$, which means that the minimal number of n-gram constructions is 112 and the maximum number is 480.

$$\sum_{n=2}^{a} \binom{a}{n} \cdot v^n \cdot n! \tag{7.3}$$

Experimental Results

The results of a simulation in which the agents make use of the n-gram strategy are visualised by the red lines in Figures 7.13 and 7.14. The dark red line in Figure 7.13 shows that the communicative success starts at about 0.1 and increases to 1 after approximately 6000 interactions per agent. The bright red line in the same figure indicates that the agents start with no n-gram constructions at all, and end up with an inventory of over 320 such constructions. While the number of n-gram constructions stabilises quite a bit below the theoretical maximum of 480, it goes far above the optimal number of 112. The reason that it goes over 112 is that different word orders are introduced by different agents, and that all these word orders propagate in the population. The reason that it stays below 480 is that with a population of 10 agents,

the constructions spread quite fast and the communicative task is solved before all word orders have been invented. The larger the population is, the more variation in word order is introduced, and the more constructions the agents end up with. The dashed red line indicates the coherence of the language, which stabilises just below 0.4. The red line in Figure 7.14 shows the amount of search that is performed by the agents, starting at 1 and climbing to about 2.5 by the time at which the maximum number of constructions is reached. While the amount of search is slightly lower than in the case of the grouping strategy, and the language is slightly more coherent, there is no dramatic improvement. N-gram constructions can indeed lead to clearer borders between the groups and to a more stable word order, but this only holds when not all possible n-gram constructions for a particular combination of words are available. For example, the use of an n-gram construction for "large square" will only lead to a smaller search space than a grouping construction for "large square", if the construction inventory does not contain an n-gram construction for "square large". If all 480 possible n-gram constructions are available, the amount of search that is needed equals the amount of search that is needed in the case of the grouping strategy. Likewise, "large square" will only be consistently used in that order if the construction inventory does not include an n-gram construction for "square large". The 320 constructions that are created in this experimental run only avoid a limited number of n-gram constructions, which leads to a slightly smaller search space and a slightly more coherent language.

The key to a more significant reduction of the search space and a more coherent language is to minimize the word order variation among the agents of the population. Let's have a look at an improved version of the n-gram strategy, in which this is achieved by the *alignment* step in the interaction script. After each interaction, the speaker and hearer update the scores of the constructions in their construction inventory, based on the outcome of the interaction. The particular updating rule that is used here is based on lateral inhibition (De Vylder and Tuyls, 2006). The score of a construction ranges from 0 (not usable) to 1 (well entrenched). Whenever an agent, whether it is the speaker or the hearer, creates a new construction, this construction enters the construction inventory with an initial score of 0.5. After each interaction, the scores of the constructions are updated as follows.

- If the interaction succeeded, the speaker and hearer will both:
  - increase the scores of the constructions that they used by the value of the `:li-incf-score` parameter (default 0.1).
  - decrease the scores of any competing constructions by the value of the `:li-decf-score` parameter (default 0.2).

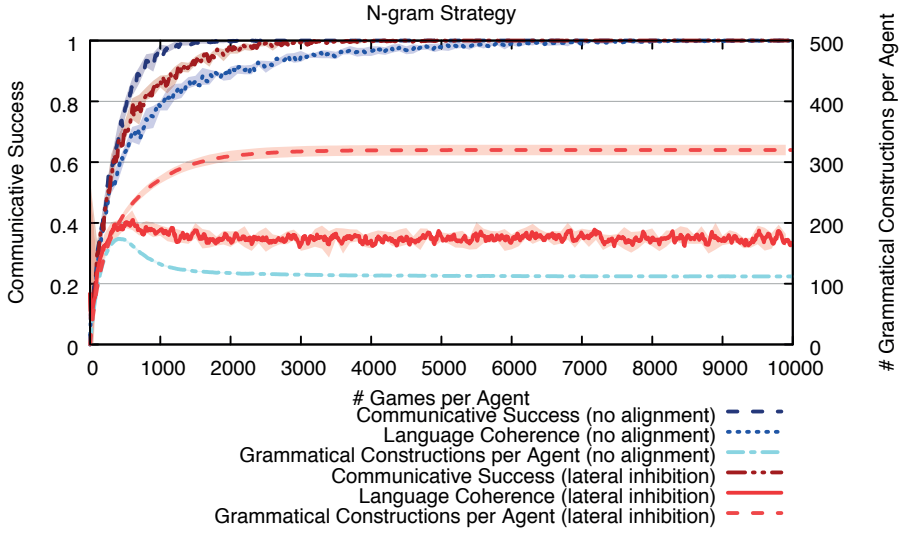


Figure 7.13: Results of a simulation in which the agents use the n-gram strategy. The dark lines indicate the communicative success with and without alignment mechanisms (left y axis), and the lighter lines indicate the average number of n-gram constructions in the inventories of the agents, with and without alignment mechanisms.

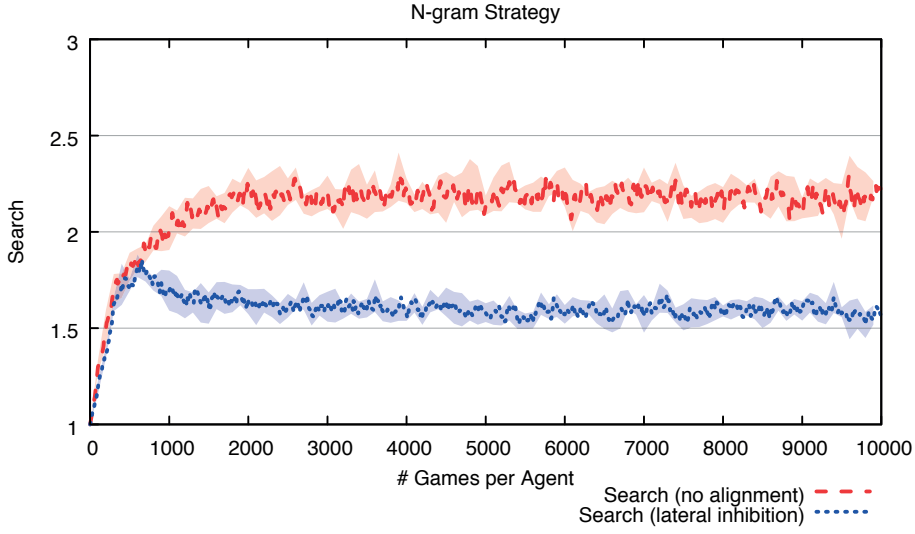


Figure 7.14: Size of the search space that was explored while applying the n-gram-cxns with and without alignment mechanisms. The y-axis indicates that total number of nodes created by n-gram-cxns, divided by the depth of the solution node.

- If the interaction failed, the speaker will:
  - decrease the scores of the constructions that he used by the value of the `:li-decf-score` parameter (default 0.2).

Constructions with a higher score have a higher priority in the construction application process. This is especially important in the case of the speaker, as he is the one who formulates the utterance and thus chooses the order of the words that are used in that interaction. Competing constructions are defined as constructions which match on the exact same lexical units, but in a different order, for example constructions matching on *large square* and *square large*.

The blue lines in Figures 7.13 and 7.14 visualise the results of a simulation in which the agents use the updating rule above. The dark blue line in Figure 7.13 indicates the communicative success, which goes from 0.1 to 1 in about 3000 interactions per agent. This is about twice as fast as in the version of the experiment in which no alignment took place. The reason for this is that the word order conventions are shared much faster, and as a consequence, fewer n-gram constructions need to be learned by the agents. The number of n-gram constructions (with a non-zero score) is indicated by the bright blue line. It starts at zero and increases to about 180 after 400 interactions per agent. Then, it starts to decrease and stabilises at 112, i.e. the minimal number of n-gram constructions that is needed to cover all scenes that can be drawn from the world. The royal blue line indicates the coherence of the language. It shows that full coherence is reached after about 10000 interactions. This version of the experiment effectively leads to a shared language in which all agents in the population consistently use the same order for the same words. As for the amount of search, indicated by the blue line in Figure 7.14, we can see that it follows the same dynamics as the number of n-gram constructions. It starts at 1 and grows to about 1.9, after which it decreases again, stabilising at about 1.7. Using lateral inhibition, the n-gram strategy leads thus to the emergence of a coherent language with fewer constructions and a lower referential ambiguity, which can be processed more efficiently.

7.4.4 Pattern Strategy

The *pattern strategy* is the fourth and final learning strategy. The main property that differentiates the pattern strategy from the grouping and n-gram strategies is that the constructions that are created by the pattern strategy do not match on individual words. Instead, the constructions represent more general and abstract patterns that combine words based on their syntactic categories. This reduces the number of constructions

that is needed, while keeping the referential ambiguity of the language low. Importantly, the syntactic categories of the words are not a given, but an outcome of the experiment.

The experiment is initialised in exactly the same way as before. At the start of the experiment, the construction inventories of the agents contain an empty type hierarchy, and one lexical construction for each word, i.e. for each possible value of each dimension that occurs in the world. The lexical constructions are the same as those that were used with the other strategies. An example construction for the word 'blauw'/blue(?x) was shown in Figure 7.2 above. In comprehension, this construction matches on the string "blauw" and in formulation, it matches on the meaning predicate 'blue(?x)'. The construction creates a new unit ('?blauw-unit-11') and merges a feature 'syn-cat: lex-class: blue-6' into this unit. The value of this feature is a unique symbol for each word of each agent. The initial lexicon is thus not (yet) structured according to any syntactic or semantic properties. Note that the fact that the value of the 'lex-class' symbol contains 'blue' has a purely mnemotechnic reason, and that the symbol has no external meaning at all (see Section 4.2).

Diagnostics and Repairs

The diagnostics that are used by the speaker and the hearer remain the same as in the previous strategies: in the case of the speaker, a diagnostic triggers based on the result of the re-entrance goal test, and in the case of the hearer, a diagnostic triggers when the interaction failed. The meta-layer repairs however, now make use of FCG's type hierarchy system and anti-unification operator, two of the main contributions of this dissertation (see chapters 4 and 5). For the speaker as well as for the hearer, the repair process looks as follows:

1. The agent goes through the objects in the topic, and for each object, he looks up whether his construction inventory contains a pattern-cxn that matches on as many lexical units as the number of dimensions that need to be expressed for that object (in the case of the speaker) or that are expressed for that object (in the case of the hearer). For example, if the agent needs to express {small(obj-5), triangle(obj-5), green(obj-5)} (as speaker), or needs to comprehend "small green triangle" (as hearer), he will inspect whether his construction inventory contains a pattern construction that matches on three lexical units. If this is not the case, he proceeds to step 2. If it already contains such a construction, he immediately proceeds to step 3.

2. The agent creates a pattern-cxn that matches on as many lexical units as the number of dimensions that need to be expressed (in the case of the speaker)

or that are expressed (in the case of the hearer). An example pattern-cxn with three slots is shown in Figure 7.15. This construction matches on 3 lexical units that are adjacent ('meets' constraints) and co-referent ('args' features), and that have the lex-classes 'first-slot-14', 'second-slot-14' and 'third-slot-14' respectively. Again, the names of these symbols are chosen for mnemotechnic reasons only. For the creation of a new pattern-cxn that matches on n units, three cases need to be distinguished:

- The construction inventory does not yet contain any constructions for shorter or longer patterns. In this case, the agent creates a new construction matching on n adjacent units. The value of the 'lex-class' feature of each unit is a new and unique symbol.

- The construction inventory already contains at least one construction for a longer pattern. In this case, the agent creates all possible pattern constructions of length n that match on linearly adjacent units of which the values of the 'lex-class' features are an ordered subset of the values of those features in the longest construction. For example, imagine that a pattern-cxn matching on 2 units needs to be created and that the longest construction contained in the construction inventory matches on 3 units with lex-classes 'slot-a', 'slot-b' and 'slot-c'. 3 new pattern constructions matching on 2 units are created, namely constructions matching on units with the lex-classes 'slot-a' and 'slot-b', the lex-classes 'slot-a' and 'slot-c' , and the lex-classes 'slot-b' and 'slot-c'.

- The construction inventory does not yet contain any constructions for longer patterns, but already contains one or more constructions for shorter patterns. In this case, the agent creates a construction matching on n units by adding one or more units to a copy of the longest pattern-cxn in the construction inventory. The new units can either be added to the front or the back of the pattern and the values of their 'lex-class' features are new, unique symbols. Apart from this new pattern-cxn of length n, additional pattern-cxns of length $< n$ are also created if a construction of this length was already present in the construction inventory. These additional constructions ensure that also for the shorter patterns, combinations including the new 'lex-class' values are created. For example, imagine that a pattern-cxn with length 3 needs to be created and that there is already a pattern-cxn with length 2 in the construction inventory, which matches on the lex-classes 'slot-a' and 'slot-b'. First, a new pattern-cxn of length 3 is created. This construction matches on lex-classes 'slot-a', 'slot-b' and

'slot-c' (or alternatively on 'slot-c', 'slot-a' and 'slot-b'). Additionally, two more constructions of length 2 are created, matching on 'slot-a' and 'slot-c' and on 'slot-b' and 'slot-c' (or alternatively on 'slot-c' and'slot-a' and on 'slot-c' and 'slot-b').

This way of creating new constructions might seem complex at a first glance, but it is actually equivalent to maintaining a single construction in which units are made optional (when shorter patterns occur) or in which additional optional units are added (when longer patterns occur). For example, the extension of a pattern A-B to A-B-(C) is equivalent to the addition of patterns A-B-C, A-C and B-C. This technical solution is used because FCG's routine processing does not support the application of constructions with optional units, for efficiency reasons.

3. For each object in the topic, the agent anti-unifies the lexical units in the transient structure that refer to the object, with the pattern construction that matches on the same number of units. The anti-unification process returns the minimal set of new nodes and links that need to be added to the type hierarchy of the construction inventory for the pattern-cxn to be able to apply. If there are multiple pattern constructions that match on the required number of units, the anti-unification process is performed for each construction and the result of the anti-unification process that could be performed with the lowest cost is selected (see Section 5.4.2). Once the resulting nodes and links are added to the type-hierarchy of the agent, the construction can apply and routine processing continues. For the example above, in which the speaker needed to express {small(obj-5), triangle(obj-5), green(obj-5)} or the hearer needed to comprehend "small green triangle", imagine that the agent created the new pattern-cxn in Figure 7.15, and that his type hierarchy is still empty. The anti-unification of this construction with the lexical units for 'small', 'triangle' and 'green' will return 6 new nodes and 3 new links. The nodes are the 'lex-class' values of the 3 lexical units in the transient structure and of the 3 lexical units in the pattern construction. The links connect each 'lex-class' value in the transient structure to a 'lex-class' value in the construction. The resulting type-hierarchy is shown in Figure 7.16. The arrows indicate that 'first-slot-14', 'second-slot-14' and 'third-slot-14' in the construction can respectively match on 'small-23', 'green-23' and 'triangle-23' in the transient structure (which were the lex-classes introduced by lexical constructions). The weights on the edges represent the entrenchment of the individual links. The weights range from 0 to 1, with 0 meaning that the association between the two symbols is very strong (completely entrenched), and 1 meaning that the association is very weak. When a link is added to the

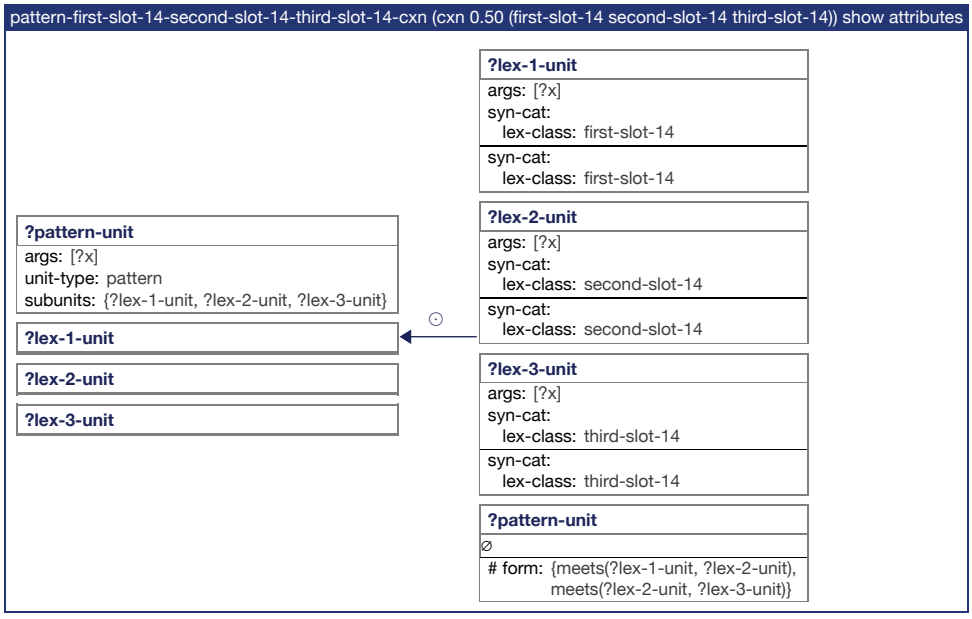


Figure 7.15: An example of an pattern construction with three slots. The 'meets' constraints impose that 'lex-1-unit', 'lex-2-unit' and 'lex-3-unit' occur in a sequence. The shared variables in the 'args' features of these units ensure that they are co-referential. The 'lex-class' features will be matched through the type hierarchy.

type hierarchy, it receives an initial weight of 0.50. If more than one new type hierarchy link needs to be added, the speaker will randomly assign the lex-classes in the transient structure to the slots in the construction. The hearer will of course use the word order that he observed.

Type Hierarchy Build-Up

Using the diagnostics and repairs described above, the agents rapidly establish a set of pattern constructions matching on 2 to *a* lexical units, with *a* being the maximum number of properties that need to be expressed about a single object in the topic. In our experimental set-up, this number equals the number of dimensions of the objects in the world. The exact number of patterns can be calculated using the formula in (7.4).

$$\sum_{n=2}^{a} \binom{a}{n} \tag{7.4}$$

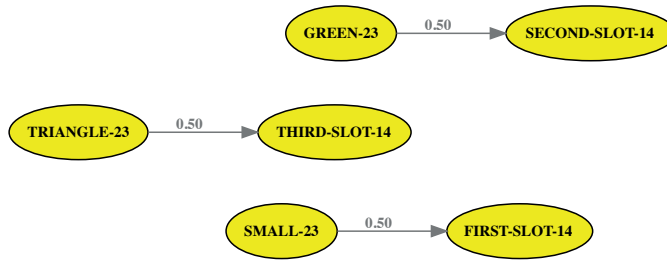


Figure 7.16: Through the type hierarchy, the values of the 'lex-class' features in the pattern-cxn ('first-slot-14', 'second-slot-14', 'third-slot-14') can match on the values of the 'lex-class features' in units created by lexical constructions ('small-23', 'green-23', 'triangle-23').

Let us now have a closer look at how these pattern constructions interact with the lexical constructions in the grammar. As explained above, each lexical construction introduces a unit with as value for its 'lex-class' feature a symbol that is unique to that construction. This means that there exist v^a different 'lex-class' values in the lexicon. The pattern constructions match, as a result of step 2 of the repair above, on units which can have a different 'lex-class' values. These values are symbols that only occur in the pattern constructions, and not in any other constructions of the grammar. As a consequence, the application of a pattern construction to any units that were created by lexical constructions will always need to make use of the type hierarchy of the grammar. In other terms, it is the type hierarchy of the grammar that specifies which lexical items can fill which slots of which pattern constructions.

The build-up of the type hierarchy is handled by the anti-unification process described in step 3 of the repair above. Where it is needed, the repair will add new nodes, which are 'lex-class' values, to the type hierarchy, as well as links from the a^v 'lex-class' values that occur in the lexical constructions to the a 'lex-class' values that occur in the pattern constructions. When the experiment is run as such, the type hierarchies of the agents will eventually contain all 'lex-class' values from the lexical and pattern constructions, and equal weight links from each 'lex-class' value that occurs in the lexicon to almost all 'lex-class' values that occur in the pattern constructions. A type hierarchy of this kind is shown in Figure 7.17. The (almost) fully connected graph and the equal weight links visualise that the agent has no preferential associations between his lexical items and the specific slots in his pattern constructions. This makes the language incoherent, in the sense it will never converge on a single word order.

For a coherent language to emerge and evolve, an alignment rule like the one that was used with the n-gram strategy is put in place. The updating rule is again based on lateral inhibition, but the weights of the links in the type hierarchy are now updated,

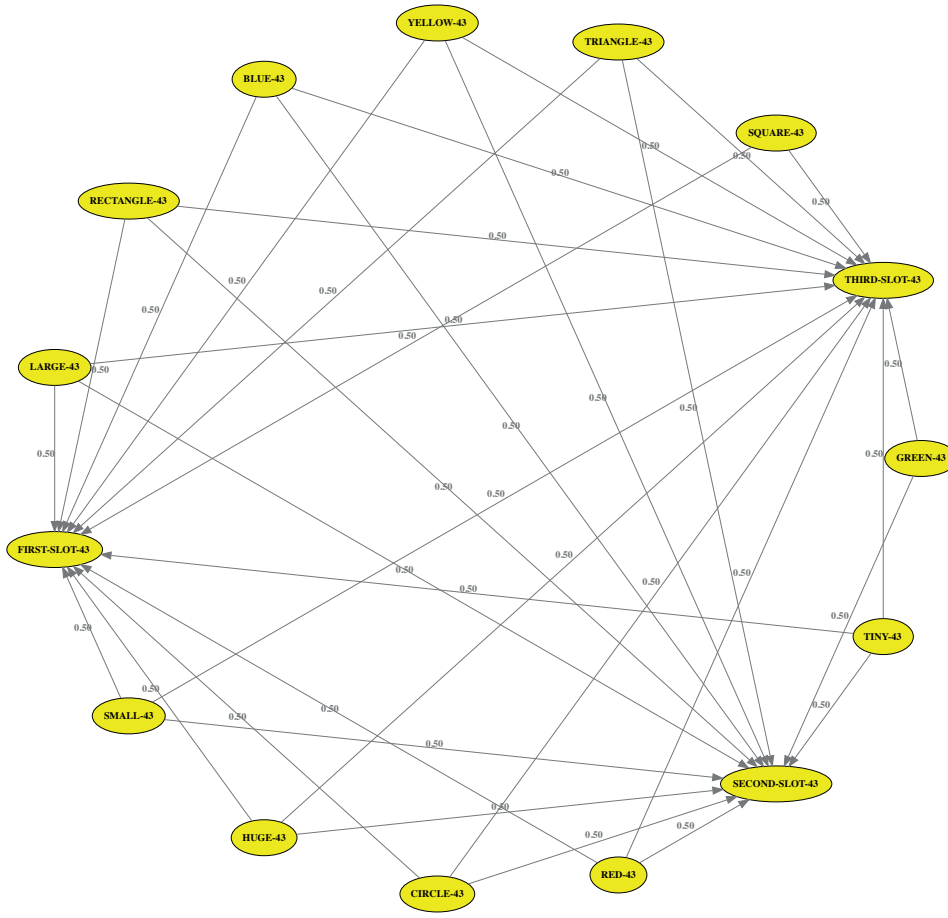


Figure 7.17: Type hierarchy of an agent using the pattern strategy without alignment mechanisms. The network is almost fully connected with equal weight links, which means that there exist no preferential associations between the lexical items and specific slots in the pattern constructions. This leads to a language that is incoherent in the sense that it does not converge on a single word order.

rather then the weights of the constructions themselves. The following rule is used:

- If the interaction succeeded, the speaker and hearer will both:
  - increase the scores of the links in the type hierarchy that they used in the matching phase by the value of the `:li-decf-weight` parameter (default 0.1).
  - decrease the scores of any competing links in the type hierarchy by the value of the `:li-incf-weight` parameter (default 0.2).
- If the interaction failed, the speaker will:
  - decrease the links links in the type hierarchy that he used in the matching phase by the value of the `:li-decf-weight` parameter (default 0.2).

Competing links are defined in relation to the construction that they were used in. For each link that was used during the construction application process, any link that goes from the same symbol in the transient structure to a different symbol in the construction is considered a competitor. For example, imagine that a pattern construction matches on two units with 'lex-class' values 'first-slot-3' and 'second-slot-3' and that these values are matched through the type hierarchy with the symbols 'green-12' and 'square-12' respectively. Any links that go from 'green-12' or 'square-12' to any other symbols in the construction would in this case be considered competitors. Taking into account the repair strategy that was used, the only links that could possibly have been added to the type hierarchy are those that connect 'green-12' with 'second-slot-3' and 'square-12' with 'first-slot-3'. If these links are present in the type hierarchy, they will be subject to punishment.

During the construction application process, type hierarchy links with a lower weight are prioritised over links with a higher weight. Concretely, construction application results (which correspond to nodes in the construction application process) of which the average weight of the type hierarchy links that were used is lower, get a higher priority in the queue and are thus explored earlier. This is especially important in formulation, as it ensures that the speaker uses his preferred word order.

Experimental Results

The results of a simulation in which the agents make use of the pattern strategy are shown in Figures 7.18 and 7.19. The turquoise line in Figure 7.18 shows that the population reaches full communicative success after about 225 interactions per agent. The green line indicates that the language of the agents reaches full coherence after

about 400 interactions per agent. The yellow line indicates that the average number of pattern constructions in the construction inventories of the agents goes from 0 to 4 in only a few interactions, and then stays constant throughout the rest of the experiment. The number of pattern constructions is indeed in accordance with the formula in (7.4), as the number of dimensions in the experimental set-up is 3 ($4 = \sum_{n=2}^{3} \binom{3}{n}$). The fact that the agents have acquired all four constructions after only a few interactions is not surprising, as they only need to have participated in one interaction in which 2 dimensions of an object are expressed, and one interaction in which 3 dimensions of an object are expressed, in order to have learned all 4 constructions. All further learning is achieved by creating new links in the type hierarchy and updating the weights of these links. If the world and population remain the same, no new links will be added after the communicative success reaches 1, as the diagnostics only trigger in case of communicative failure. Likewise, if the world and population remain the same, the word order that the agents use will not change any more after the coherence of the language has reached 1, as there is no longer competition between the agents. From that moment on, the weights on the links will only be reinforced to 0 for preferential associations or 1 for non-preferential associations, while the preferential associations themselves remain the same.

The blue line in Figure 7.19 visualises the amount of search that is needed during the hearer's comprehension process, as calculated by dividing the number of explored nodes created by the application of pattern constructions by the number of nodes in the shortest path to the solution. Starting at 1, this number quickly rises to about 5,5 as more constructions and type hierarchy links become available. After reaching a peak at about 50-100 interactions per agent, the amount of search that is needed decreases as the language starts to get more coherent. Once the language is fully coherent, the average amount of search needed stabilises at about 3.9.

Let us now have a closer look at the language that emerges during simulations in which the agents use the pattern strategy. In the experiment, objects have 3 observable dimensions, with 4 possible values each. The agents start with 12 lexical constructions, namely one construction for each value. When a lexical construction applies, it introduces a unit with as 'lex-class' value a symbol that is unique to the construction (e.g. 'green-70'). Note that the construction contains no information at all about the dimension of the value that it expresses. Once the experiment starts, the diagnostics and repairs of the agents rapidly construct 4 grammatical constructions, which match on 2 or 3 units (called 'slots') with as 'lex-class' values all ordered subsets of three different symbols (e.g. 'first-slot-54', 'second-slot-54' and 'third-slot-54'). Over the course of further communicative interactions, the diagnostics and repairs add all these

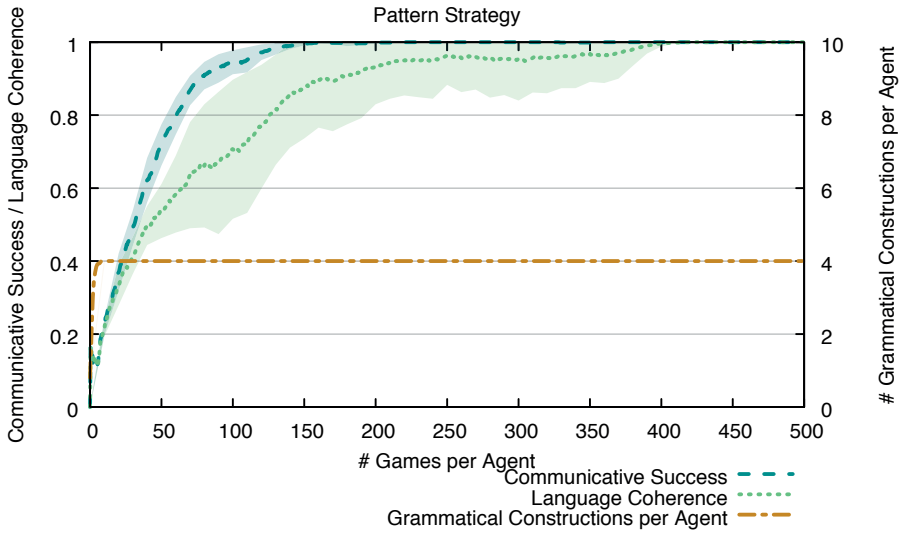


Figure 7.18: Results of a simulation in which the agents use the pattern strategy. The turquoise and green line indicate communicative success and lexical coherence respectively (left x-axis). The yellow line indicates the average number of grammatical constructions in the construction inventories of the agents.

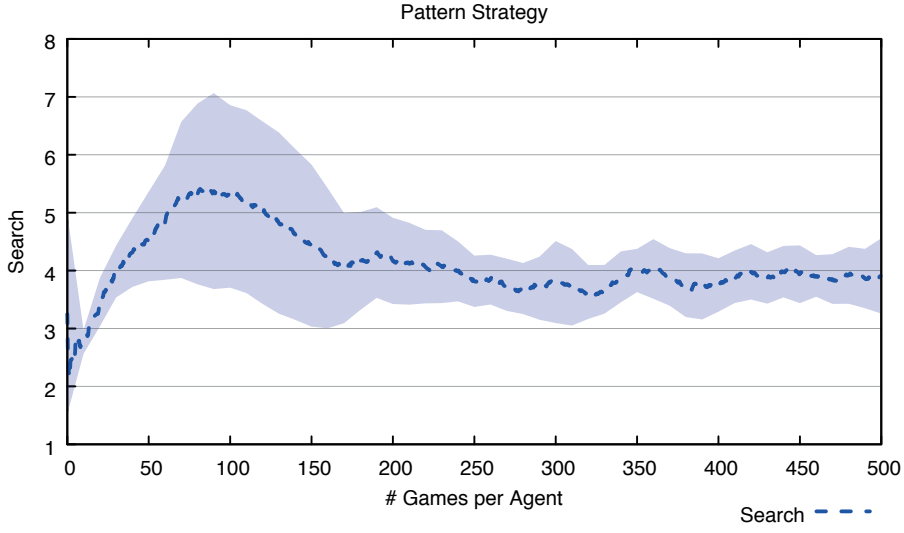


Figure 7.19: Size of the explored search space during a simulation in which the agents use the pattern strategy.

symbols to the type hierarchy of the agent, and construct links from 'lex-class' values in the lexical constructions to 'lex-class' values in the grammatical constructions (e.g. 'green-70 → third-slot-54'). As there is a large amount of freedom in the creation of new links, there is initially a lot of variation in the population and the type hierarchies of the agents are almost fully connected graphs, like the one that was shown in Figure 7.17 above. At this point, the population reaches full communicative success. However, the language is still incoherent and its processing requires a lot of search effort, as almost each lexical item is equally likely to fill almost each slot in the pattern constructions. In order to make the language more coherent, the agents adjust the weights on the type hierarchy links after each interaction using a lateral inhibition updating rule. This effectively leads to a language in which each lexical item is strongly associated with a single slot in the pattern constructions.

A graph showing the type hierarchy of an agent after 400 interactions is presented in Figure 7.20. The lower the weight of a link is, the darker it is visualised. The graph shows that each 'lex-class' value from a lexical construction has indeed one link of maximum strength with a 'lex-class' value that represents a slot in the pattern constructions. All other links have become minimally strong. While the type hierarchy of every agent is different, as it contains the agent's own categories and is shaped by the agent's own past interactions, the preferential links of each agent are equivalent when the language reaches coherence. Remarkably, the lexical items associated to the different slots are grouped per dimension. In this case, all size values are linked to the first slot, all shape values to the second slot, and all color values to the third slot. Note that this is solely a consequence of the fact that the objects in the world can never be discriminated using two values on the same dimension (e.g. an object is never 'large' and 'small') and that this information is never explicitly provided to the agents.

7.5 Comparison and Discussion

This section compares the results of the simulations using the four learning strategies that were presented in the previous section and discusses their strengths and weaknesses in terms of communicative success (7.5.1), language coherence (7.5.2), required number of grammatical constructions (7.5.3) and required search effort (7.5.4). Finally, a more general discussion closes the section (7.5.5).

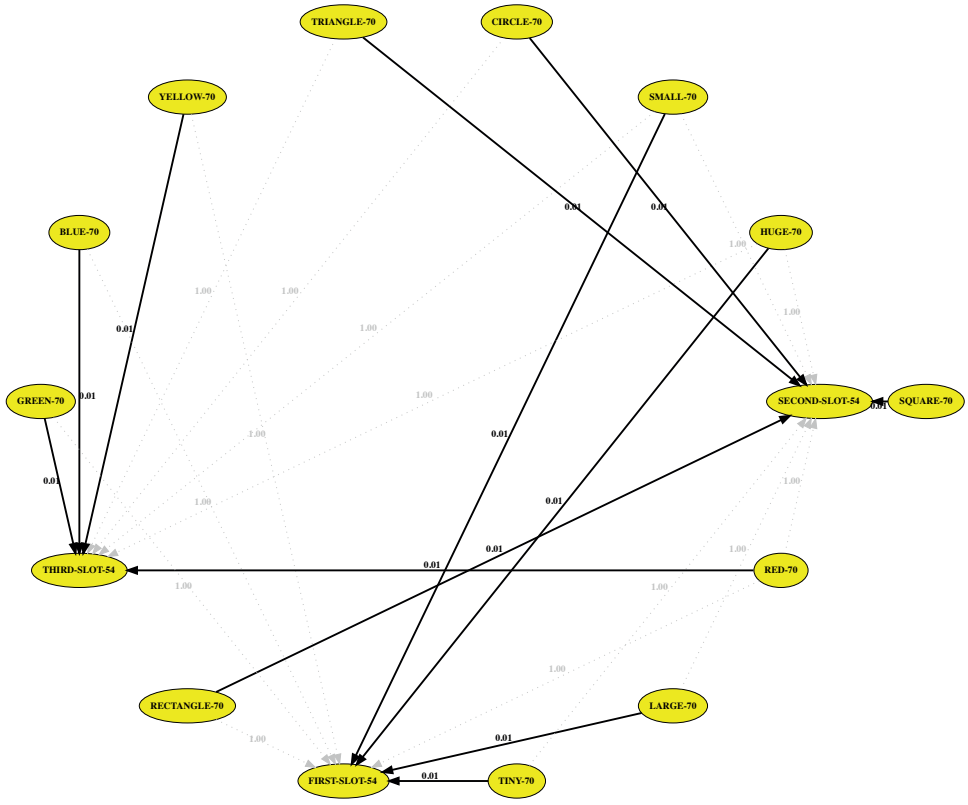


Figure 7.20: Type hierarchy of an agent using the pattern strategy and type hierarchy alignment mechanisms after 400 interactions under the default experimental conditions. Edges with a lower weight are visualised darker. The graph clearly shows the preferred associations between the 'lex-cat' values of the words and the 'lex-cat' values of the different slots in the pattern construction. Not only is there a single preferred association of each word with each slot, the associations are also semantically relevant (size, shape, color).

7.5.1 Communicative Success

The primary goal of the learning strategies is to allow the agents in the population to emerge and evolve a language with which they can successfully communicate. Hence, the percentage of interactions in which the agents achieve communicative success can be seen as the most important metric for evaluating the learning strategies. Apart from the success rate itself, also the number of interactions that it takes until a certain rate of success is attained is an important metric. The faster the maximum success rate is attained, the better the learnt structures generalise to novel observations. Clearly, the generality of the learnt structures and consequently their applicability in novel situations are important factors for any learning system that has to interact with the real world. A graph comparing the experimental results for the four strategies in terms of communicative success is shown in Figure 7.21.

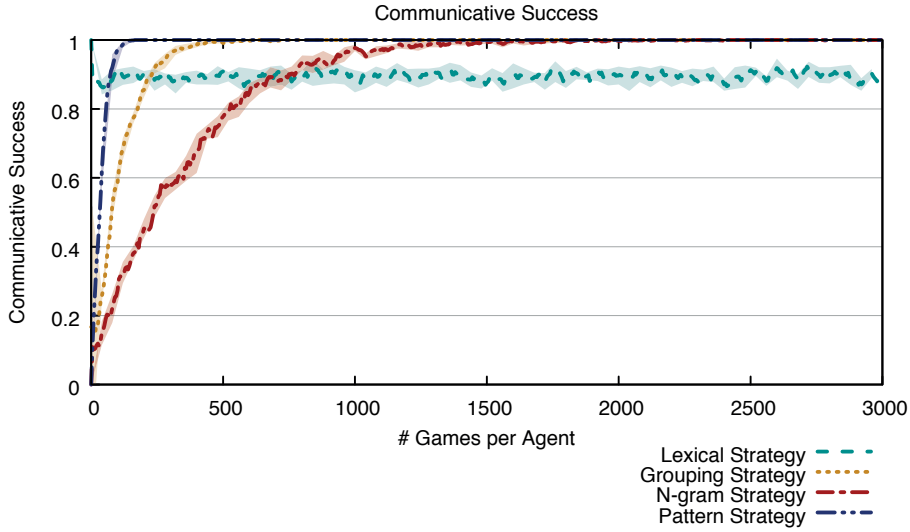


Figure 7.21: Graph comparing the four learning strategies in terms of communicative success.

The graph shows that the pattern strategy, the grouping strategy and the n-gram strategy all three attain a success rate of 100%. In the case of the lexical strategy, the success rate stagnates at about 90%. This means that under the current experimental conditions, the disambiguating power of first three strategies is sufficient to completely solve the communicative task, whereas the disambiguating power of lexical strategy is not sufficient in 1 out of 10 situations. The number of interactions that it takes in order to reach the maximum success rate is very different for the four strategies. The lexical strategy achieves its success rate of 90% immediately. The pattern strategy

achieves 100% communicative success after only 250 interactions per agent. The grouping strategy and n-gram strategy achieve a 100% success rate after 700 and 3000 interactions per agent respectively. In terms of the generality of the learnt structures, this means that the constructions used in the lexical strategy are the most general ones, followed by those used in the pattern, grouping and n-gram strategies respectively. Indeed, the lexical strategy only needs 2 constructions, which are applied as such in all cases. The pattern strategy needs to learn at least one type hierarchy link for each word that is used. The grouping strategy needs to learn a different grouping construction for each combination of words that is used, and the n-gram strategy needs to learn a different n-gram construction for each combination of words in each order that is used.

7.5.2 Coherence of the Language

A second important factor for the evaluation of the learning strategies is the degree of coherence of the language that emerges. I have defined coherence above as the probability that the hearer would have expressed the same conceptualisations for the same objects, with the exact same utterances as the speaker. Coherence is an important property of an efficient language, because it increases the predictability of its utterances. The effects of language coherence and predictability are not directly studied in this experiment, but there is a wide consensus in the psycholinguistic and neurolinguistic literature that prediction plays an important role in human language processing (see e.g. DeLong et al., 2005; Pickering and Garrod, 2013). A graph comparing the experimental results for the four strategies in terms of language coherence is shown in Figure 7.22.

The graph shows that the lexical strategy and the grouping strategy do not lead to a coherent language. The probability of the hearer using the exact same utterance as the speaker is between 0.2 and 0.3. The pattern strategy and the n-gram strategy on the other hand, both lead to a fully coherent language. In the case of the pattern strategy, full coherence is reached after about 400 interactions. At this moment, all preferential associations between the individual words and the slots in the pattern constructions are shared by all agents in the population. In the case of the n-gram strategy, full coherence is only reached after about 10000 interactions. The reason that the n-gram strategy needs many more interactions to reach coherence is due to the fact that for each combination of words, a preferential word order needs to propagate in the population.

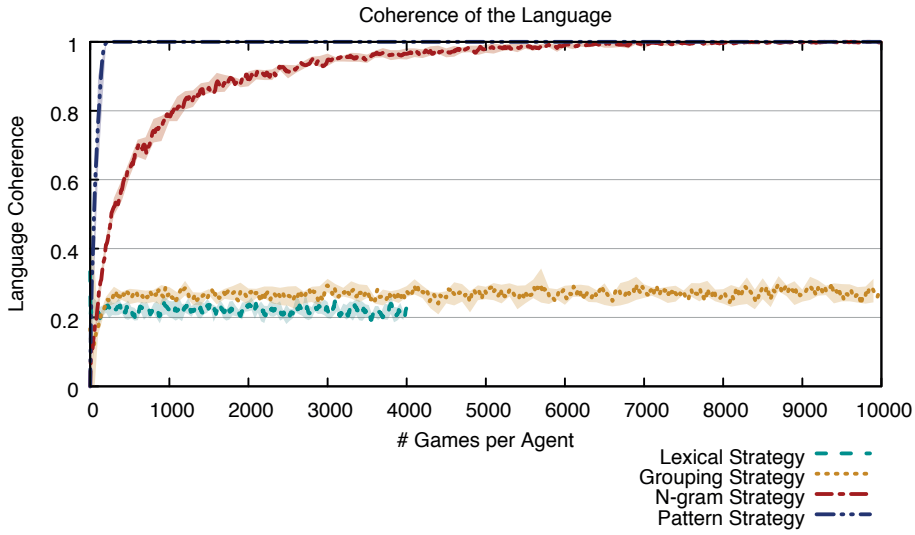


Figure 7.22: Graph comparing the four learning strategies in terms of coherence of the emerged language.

7.5.3 Number of Grammatical Constructions

The number of grammatical constructions that are created using the different learning strategies is again related to the generality of the learnt structures. The more general the structures are, the fewer constructions are needed for covering the same expressions, and the more specific the structures are, the more constructions are needed. A graph comparing the experimental results for the four strategies in terms of number of grammatical constructions is shown in Figure 7.23.

The graph shows that the lexical strategy requires the lowest number of grammatical constructions, namely 2. These very general constructions are not learned, but provided to the agents at the beginning of the experiment. The pattern strategy needs 4 constructions for covering all expressions. These constructions are already learnt during the first interactions. The pattern constructions are very specific, but the addition of links to the type hierarchy gradually expands their coverage to novel words. The grouping strategy needs 112 constructions, namely one construction per possible combination of words. All grouping constructions are learnt by the time the communicative success reaches 100%, after about 700 interactions. Finally, the n-gram strategy initially leads to an inventory of almost 180 constructions, after which the inventory is also reduced to 112 constructions. The final number of constructions is only attained after the language has reached full coherence, after about 10000 interactions.

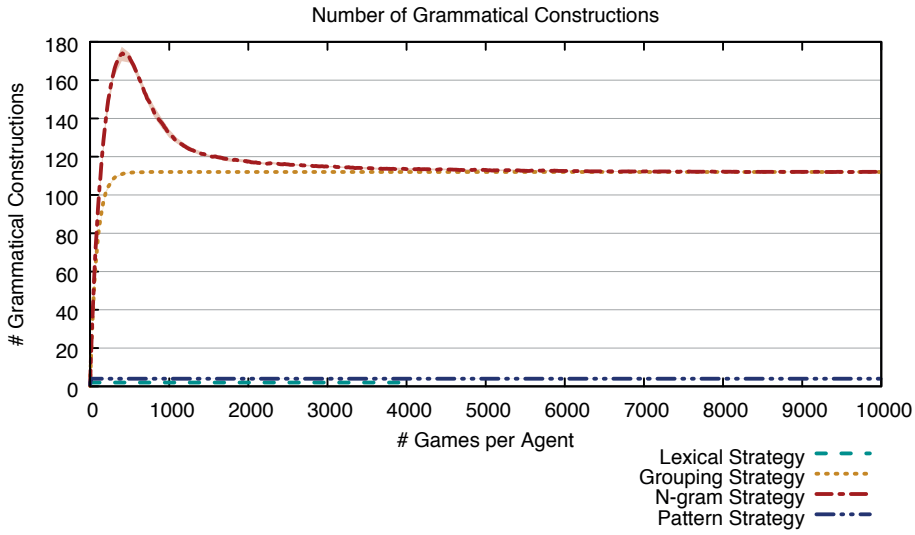


Figure 7.23: Graph comparing the four learning strategies in terms of the required number of grammatical constructions.

7.5.4 Search Effort

The last measure for comparing the learning strategies is the amount of search that the comprehension process requires. As explained above, the required search effort is calculated by dividing the number of visited nodes in the search tree created by all possible construction applications by the depth of the solution node. A lower number means that the language is less ambiguous and can therefore be processed more efficiently. A graph comparing the experimental results for the four strategies in terms of search effort is shown in Figure 7.25.

The graph shows that the lexical strategy requires the largest search effort. In average, 11 construction applications need to be performed for each construction application that brings the agent closer to the solution. This is not surprising, as the lexical strategy puts no restrictions on which words in an utterance can refer to the same or different objects. The n-gram strategy can be processed most efficiently. Once the language has reached full coherence, the search effort stabilises just under 2. The search effort required by the grouping strategy is a little bit higher, about 2.5. This is in line with the larger number of hypotheses that the grouping strategy yields. Finally, the search effort required for processing the pattern strategy stabilises between 3 and 4.

The search effort required for processing the n-gram strategy and the pattern strategy is higher in this experimental set-up than it theoretically necessary. The reason

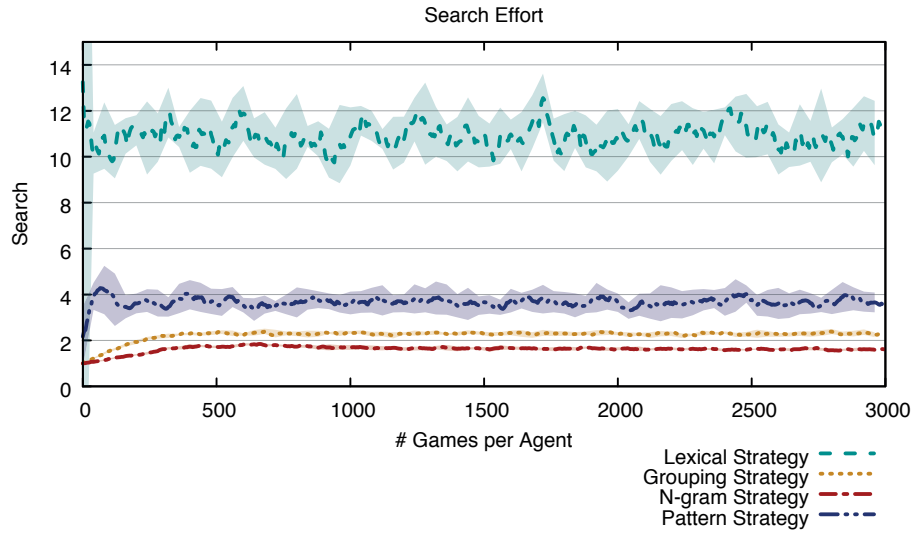


Figure 7.24

Figure 7.25: Graph comparing the four learning strategies in terms of required search effort.

has to do with constructions with score 0 in the case of the n-gram strategy, and with type hierarchy links with weight 1 in the case of the pattern strategy. In this experimental set-up, constructions that reach the lowest score and type hierarchy links that reach the highest weight do not disappear from the grammar. As the priority of these constructions and links is as low as it can be, they will not be used by the agents in formulation. In comprehension however, they sometimes become part of the search space. For example, imagine that the speaker uses the pattern strategy and refers to a large, yellow square and to something small with the words "large yellow square small". If the hearer first combines 'large yellow', which is possible as it has the same cost as combining 'large yellow square' (namely 0), he will still try to apply all 3 pattern constructions of length 2 to 'square small' before backtracking and combining 'large yellow square'. The three pattern constructions can apply, as the required links are part of the type-hierarchy, albeit with weight 1 for two of the three constructions. Simply removing constructions with score 0 or type hierarchy links with weight 1 reduces indeed the search space, but also limits the flexibility of the language. Good heuristics and optimisation strategies, such as applying constructions that span more words earlier, can certainly be designed and implemented, not only for the pattern and n-gram strategies, but for all strategies that were discussed. These would however justify a study in their own right and fall outside the scope of this case study.

7.5.5 Final Discussion

The results of this case study can be summarised as follows. The lexical strategy is the simplest strategy and requires the fewest grammatical constructions, but scores far worse than the other strategies in terms of communicative success, language coherence and required search effort. The grouping strategy attains full communicative success fast and allows for relatively efficient processing, but leads to an incoherent language and requires a large construction inventory. The n-gram strategy allows for very efficient processing and leads to a coherent language, but it requires a large construction inventory and, due to its very specific constructions, full communicative success and coherence are only attained after a very large number of interactions. Finally, the pattern strategy reaches full communicative success and coherence fast, requires a small number of grammatical constructions, and can still be processed relatively efficiently, although it requires somewhat more search than the grouping and n-gram strategies. These results are in line with the hypothesis that natural languages employ phrasal structure in order to reduce their referential ambiguity, which brings important advantages in terms of communicative success and processing effort.

The results show that the pattern strategy outperforms the other strategies in terms of the number of constructions that is needed and in terms of the number of interactions that it takes until full communicative success and coherence are attained, while still allowing to be processed efficiently. This was achieved by (i) shifting the competition from the grammatical constructions, like in the grouping and n-gram strategies, to the associations between the lexical items and the individual slots in the grammatical constructions, and (ii) making pattern constructions of different length share the same slots. This facilitates the reuse of the information on the association between lexical items and slots in the grammatical patterns, that is learned after each interaction, for the processing of other word combinations of any length that share at least one of these words.

The pattern strategy could be elegantly implemented thanks to the hierarchical type system and anti-unification operator that I implemented for FCG. The type hierarchy allows capturing the associations between the lexical items and the slots in the pattern constructions, as well as the competition between these associations, and the anti-unification operator is crucial for adding new links to the type hierarchy. As the type hierarchies that are build-up during the experiment, such as the one shown in Figure 7.20, represent the associations between words and slots in grammatical constructions, they actually capture something close to syntactic categories. An experiment that makes use of the pattern strategy can therefore also be seen as an experiment on the emergence of syntactic categories.

The results obtained by Steels and Garcia Casademont (2015a) for the lexical, grouping and n-gram strategies are in line with the results obtained in this case study. For the pattern strategy, the authors report that it leads to fewer patterns and faster convergence than the n-gram strategy, which is also in line with our results. The impact of the coercion and reuse operations that are used in their experiment is difficult to determine, which makes a further comparison difficult. However it is clear that our pattern strategy leads to considerably fewer patterns and a faster convergence as compared to the n-gram strategy than their pattern strategy. Also, the number of pattern constructions and type hierarchy links that emerge in our experiment is stable and predictable, whereas the number of constructions that emerges in their experiment varies widely over different experimental runs.

7.6 Conclusion

This chapter reported on a case study in which the representations and mechanisms that were introduced in the previous chapters, in particular the high-level FCG notation, the type hierarchy system and the anti-unification operator, were applied in a multi-agent experiment on the origins of syntax. The experiment studied four different strategies that a population of autonomous agents could use for reducing the referential ambiguity of their language by introducing primitive syntactic structures. The baseline lexical strategy did not make use of any syntactic constraints, which led to an inefficient language that was not powerful enough to solve the communicative task. The grouping strategy grouped together the co-referential words inside the utterances, and stored the individual groups in the form of constructions, so that they could be reused later. This led to a language that was powerful enough to solve the communicative task, and that could be processed efficiently, but that was incoherent. Like the grouping strategy, the n-gram strategy also grouped the co-referential words inside the utterances, but stored the groups including their internal word order. This led to a language that was powerful enough to solve the task, that could be processed efficiently and that was coherent, but that emerged only after a very large number of interactions. Finally, the pattern strategy focussed on learning associations between individual words and slots in pattern constructions. This led to a language that could be processed efficiently, that was powerful enough to solve the task, that was coherent and that emerged fast.

The power of the pattern strategy stems from the fact that the competition is shifted from entire constructions to associations between individual symbols, in this case the lexical categories that are contributed by the lexical constructions and the lexical categories on which the lexical units of the pattern constructions match. The type hierarchy

system can elegantly capture these associations and their competition, and the anti-unification operator offers a general way to detect new associations that need to be added to the type hierarchy. The case study is meant in the first place as a demonstration of the application of type hierarchy system and the anti-unification operator in an evolutionary linguistics experiment. However, it also makes an important contribution to the field of evolutionary linguistics by proposing a new methodology for modelling the emergence of syntactic categories and presenting a first experiment that makes use of this methodology.

Chapter 8

Conclusions

8.1 Introduction

The primary objective of this dissertation, as formulated in the introduction, was to improve the representations and mechanisms that are used in agent-based models of language evolution, and extend them with more powerful and general learning operators. These improved representations and mechanisms are intended to constitute a general framework that provides powerful building blocks for conducting more advanced experiments on the emergence and evolution of grammar. A secondary objective was to make use of this framework to conduct an agent-based experiment that studies how early syntactic structures can emerge and evolve in a population of agents, and

how these structures can improve the expressiveness, coherence and efficiency of the language.

I have situated these objectives within the state of the art in the field of cultural language evolution, in particular in relation to the language game paradigm (Steels, 1995). The language game paradigm employs agent-based models to determine the exact invention, adoption and alignment mechanisms with which the agents in a population need to be endowed, so that a shared language exhibiting human language-like properties can emerge and evolve through local, communicative interactions. In the past, the language game paradigm has been extensively applied to the emergence and evolution of vocabularies and conceptual systems, which has led to a good understanding of the mechanisms that are involved (Steels, 2011b). More recently, the same paradigm has also been successfully applied to the emergence and evolution of grammatical structures, but the understanding of the mechanisms that are involved there is still much more limited. A major challenge in these experiments is to endow the agents with a powerful and flexible grammar processing engine, and with general operators for inventing, adopting and aligning grammatical structures. This has not been achieved in the experiments on grammar evolution that have previously been conducted, as the learning operators that were used were always very specific and ad hoc.

The framework that I have introduced in this dissertation provides high-level representations and general and powerful meta-level operators for inventing, adopting and aligning grammatical structures. This framework is integrated in Fluid Construction Grammar (FCG), the grammar formalism that is most widely used in agent-based experiments on the emergence and evolution of grammar. The case study that I have conducted shows that the framework can be directly used to model the emergence and evolution of early syntactic structures, and the experiment confirms that these structures effectively improve the expressiveness, coherence and efficiency of the language.

In the remainder of this chapter, I will first discuss the achievements of this dissertation in more detail (8.2) and then present a number of future research routes that build further on these achievements (8.3).

8.2 Achievements

The objectives of this dissertation have materialised into five concrete achievements, which are discussed in the next sections. These achievements consist in the implementation of a high-level notation for FCG (8.2.1), the integration of a meta-level architecture (8.2.2), the design and implementation of a type hierarchy system for

FCG symbols (8.2.3), the design and implementation of meta-level generalisation and specialisation operators (8.2.4), and the design and implementation of an agent-based experiment on the origins of syntax (8.2.5).

8.2.1 A High-Level Notation for Fluid Construction Grammar

I have implemented a high-level notation for Fluid Construction Grammar (FCG), a computational platform that provides the basic building blocks for representing and processing construction grammars. The implementation of this notation consists in an interface between the notation introduced by Steels (2017) and the processing engine that was build using the FCG notation described by Steels (2011a). During the process of implementing this notation, I have also made major contributions to the design of the notation itself. The high-level notation is more intuitive and easier to read and write, as it does not need inline special operators, does not separate transient structures into poles, automatically handles footprints, and graphically distinguishes between conditional and contributing units instead of relying on an obscure 'J-unit' notation. The high-level notation is used by the FCG user for all interfacing with the processing engine. The constructions are written in the high-level notation, and the visualisations of constructions, transient structures and construction application processes are presented in the high-level notation to the user.

The implementation of the high-level notation has already had a considerable impact in the computational construction grammar and evolutionary linguistics communities. It has in the meantime become FCG's standard notation and has been used in various publications by different researchers (i.a. Marques and Beuls, 2016; Beuls et al., 2017; van Trijp, 2017; Cornudella Gaya, 2017). The source code is distributed via the Babel2 github page[1], and a few example grammars are made available online via the FCG Interactive web service (`https://www.fcg-net.org/fcg-interactive`).

8.2.2 Integration of a Meta-Level Architecture

I have integrated a general meta-level architecture into Fluid Construction Grammar. The meta-level architecture separates an agent's routine processing abilities from his problem solving and learning capacities. In FCG, routine processing is implemented as a search process, in which the constructions of a grammar are applied until a solution is found. The meta-layer monitors the routine processing by running a set of diagnostics after each construction application. If a diagnostic triggers a problem, routine

[1] `https://github.com/EvolutionaryLinguisticsAssociation/Babel2`

processing is interrupted and meta-level processing becomes active. At the meta-level, repair strategies try to solve the problem, producing a fix (e.g. a new construction or type hierarchy link). The fix is then applied and routine processing resumes. If the branch of the search tree in which the fix was applied leads to a solution, the fix is consolidated by storing it in the construction inventory of the agent, so that it can later be reused in routine processing.

The separation between routine processing and meta-level reasoning is common in cognitive architectures such as Soar (Laird et al., 1987) and MIDCA (Cox et al., 2016). There also exists ample psycholinguistic and neurolinguistic evidence for such a distinction, for example the P600 and N400 event-related potentials that are associated with syntactic and semantic anomalies respectively (for an overview, see Kutas et al., 2006). The tight integration of a meta-layer architecture into FCG provides a concrete operationalisation of the distinction between routine processing and meta-layer problem solving in computational construction grammar (Van Eecke and Beuls, 2017).

8.2.3 A Type Hierarchy System for FCG Symbols

I have designed and implemented a type hierarchy system for FCG symbols. While standard FCG represents all information inside the individual constructions of a grammar, the type hierarchy system allows capturing systematic, hierarchical relations between symbols that occur in different constructions of the grammar. In order to operationalise the type hierarchy system, a graph is added to the construction inventory. The nodes in the graph are the symbols that occur in the constructions, and the edges represent the relations between these symbols. FCG's matching and merging algorithms are adapted to take these relations into account. The edges are weighted in order to reflect the strength of the association between two symbols, very much like the scores of constructions reflect their entrenchment. The weights on the edges allow modelling competing associations, which is useful in evolutionary linguistics experiments, in which the type hierarchies of the different agents of the population need to be build up and need to get aligned through communicative interactions.

The type hierarchy system allows capturing hierarchical relations between symbols in a very fine-grained way. This avoids the need to duplicate certain information that should otherwise systematically occur in the constructions of a grammar. For example, if every 'mass noun' is also a 'common noun' and every 'common noun' is also a 'noun', this systematic relationship can be included in the type hierarchy of the grammar instead of being explicitly expressed in the individual lexical constructions. Another advantage of capturing relations in the type hierarchy system instead of in individual constructions

is that new associations can immediately be used by multiple constructions. This can considerably speed up the learning and alignment processes, as was shown in the case study in chapter 7.

8.2.4 Generalisation and Specialisation operators

I have designed and implemented two general meta-level operators for generalising and specialising FCG constructions with respect to novel observations. The generalisation operator is based on anti-unification and finds the least general generalisation of an FCG construction that matches a given transient structure. The generalisations can either be captured in the construction itself, for example by decoupling two variables or by replacing a constant with a variable, or they can be captured in the type hierarchy of the grammar by adding one or more links from constants in the transient structure to constants in the construction. The fact that fine-grained generalisations can be captured in the type hierarchy of a grammar greatly reduces the risk of overgeneralisation. The generalisation operator also includes a flexible cost calculation system. The cost reflects the number and kind of generalisations that were needed during the anti-unification process, and is crucial in determining whether a generalised construction is a good candidate to be used or not. The specialisation operator implements a process that is called pro-unification. It specialises a construction towards an observation, for example by coupling different variables in a construction that are bound to the same constants in a transient structure. This way, constraints that are systematically observed can be incorporated into a construction.

Together with the high-level notation, the integrated meta-level architecture and the type hierarchy system, the generalisation and specialisation operators provide a general framework that implements powerful building blocks for conducting agent-based experiments on language evolution. This contrasts with previous experiments on the emergence and evolution of grammar, which relied on specific and ad hoc invention, adoption and alignment mechanisms.

8.2.5 An Agent-Based Experiment on the Origins of Syntax

I have conducted an agent-based experiment that investigates how first-order syntactic structures can emerge and evolve in a population of agents through local, communicative interactions. The experiment demonstrates how the representations and mechanisms introduced in this dissertation, in particular the high-level FCG notation, the type hierarchy system and the general meta-level operators, can be directly used

to model the invention, adoption and alignment of syntactic patterns. The results of the experiment show how shared grammatical patterns can emerge and evolve in a population and confirm the hypothesis that these patterns improve the expressiveness, coherence and efficiency of the language.

The case study built further on previous experiments by Steels and Garcia Casademont (2015a). In a first step, I have reimplemented the lexical, grouping and n-gram strategies and measured the expressiveness, coherence and efficiency of the languages that emerged. The lexical strategy led to an incoherent language that was not expressive enough to solve the communicative task and that required a lot of processing effort. The grouping strategy led to a more expressive and efficient language that emerged fast, but that was still incoherent. Finally, the n-gram strategy led to an efficient language, that was coherent and expressive, but that emerged and converged very slowly.

In a second step, I have designed and implemented an improved version of the pattern strategy introduced in the same paper. Instead of using the highly intricate and ad hoc learning mechanisms that were used in the original experiment, I have implemented a pattern strategy that makes use of the general framework that was presented in this dissertation. This strategy relies on the meta-level generalisation operator to detect the minimal generalisations that need to be made to a construction in order to expand its coverage to a novel observation. In this case, these generalisations allow a specific word to be used in a specific slot of a pattern construction. They are stored in the type hierarchy of the grammar and can be used by all constructions. After each game, the links in the type hierarchy are updated based on the outcome of the interaction, which causes the type hierarchies of the different agents to align. The pattern strategy leads to an expressive, coherent and relatively efficient language that emerges after only a limited number of interactions.

8.3 Future Research

Four of the five achievements of this dissertation together form a framework that provides improved representations, processing mechanisms and learning operators for computational construction grammars. The fifth achievement is a case study that shows how this framework can be used in a concrete evolutionary linguistics experiment that studies the emergence of early syntactic structures. The contributions presented in this dissertation open many possibilities for further research.

A first path that will be pursued is to make use of this framework to conduct more

advanced experiments on the evolution of grammar. As a first step, an experiment will be set up, in which the pattern strategy described in the case study is extended from first-order syntactical structures to recursive hierarchical structures that express relations between objects. As a next step, the pattern strategy will be combined with a marker strategy such as the one introduced by Beuls and Steels (2013). This experiment will allow investigating the interplay between different language strategies, and shed light on how a certain strategy can emerge, become dominant and disappear again as a different strategy emerges. Further experiments will study processes of grammaticalisation, in which lexical words specialise in a specific function and become grammatical markers, like in English the lexical verb 'will' specialised into an auxiliary that marks future tense (Heine and Kuteva, 2002; Traugott and Trousdale, 2013). The results of these experiments will contribute to a better understanding of the mechanisms through which language can emerge, evolve and adapt to the communicative needs of the language users.

Secondly, the framework presented in this dissertation will be used to build intelligent systems that solve concrete communicative tasks, for example question answering. In a first project, it will be used in a visual question answering system that comprehends the question of a user, interprets it in an image and formulates an answer. The grammar is used to map between utterances and functional programs that represent the meaning of these utterances as a combination of primitive operations that are implemented as modular symbolic or subsymbolic operations (Andreas et al., 2015; Johnson et al., 2017). For example, the question "are there more cats than dogs in the picture" might correspond to a program that calls a neural network that returns a mask indicating the cats in the picture, calls a neural network that returns a mask indicating the dogs in the picture, passes these masks to a neural network that counts objects, and finally calls a function that compares the resulting numbers. FCG is a very good candidate to be used in such a system, as the meaning representation can be designed to directly correspond to the primitive functions that are available. The combination of symbolic and subsymbolic approaches allows exploiting at the same time the explainability and higher-level reasoning capabilities of symbolic approaches, and the strong pattern recognition capacities of subsymbolic approaches. The modularity of the system also increases its explainability, and allows combining the same primitive operations for solving many different problems, which contrasts with large end-to-end neural network architectures. Once a grammar is in place, two kinds of multi-agent experiments will be set up. The first kind will investigate which mechanisms a population of agents needs in order to learn this grammar, and the second kind will study which mechanisms a population of agents needs in order to develop its own grammatical system that can map between natural language utterances and these

functional programs.

8.4 Final Remarks

This dissertation has introduced a framework that provides powerful building blocks for representing, processing and learning robust and flexible construction grammars, and has presented a case study that uses this framework in an evolutionary linguistics experiment on the emergence of syntactic structures. While the results of this dissertation already provide some insight into how grammatical systems can emerge and evolve, I am convinced that the tools that were introduced will greatly help further, more advanced experiments, which will ultimately lead to an understanding of how languages with human language-like expressiveness, robustness, flexibility and adaptiveness can emerge and evolve. In the future, this understanding will be crucial in building truly intelligent artificial systems.

Bibliography

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. (2015). Deep compositional question answering with neural module networks. *CoRR*, abs/1511.02799.

Appelt, D. E. (1985). *Planning English Sentences*. Cambridge University Press, New York, NY, USA.

Arbib, M. (2012). *How the Brain Got Language: The Mirror System Hypothesis*. Studies in the evolution of language. Oxford University Press, USA.

Armengol, E. and Plaza, E. (2012). Symbolic explanation of similarities in case-based reasoning. *Computing and informatics*, 25(2-3):153–171.

Baronchelli, A., Felici, M., Loreto, V., Caglioti, E., and Steels, L. (2006). Sharp transition towards shared vocabularies in multi-agent systems. *Journal of Statistical Mechanics: Theory and Experiment*, 2006(06):P06014.

Barres, V. (2017). Template construction grammar: A schema-theoretic computational construction grammar. In *2017 AAAI Spring Symposium Series*, pages 139–146.

Batali, J. (1998). Computational simulations of the emergence of grammar. In Hurford, J., Studdert-Kennedy, M., and Knight, C., editors, *Approaches to the Evolution of Language*, pages 405–426. Cambridge University Press.

Bergen, B. and Chang, N. (2005). Embodied construction grammar in simulation-based language understanding. In Fried, M. and Östman, J., editors, *Construction grammars: cognitive grounding and theoretical extensions*, pages 147–190. John Benjamins, Amsterdam.

Berthouze, L., Prince, C., Littman, M., Kozima, H., and Balkenius, C. (2007). Generalization and specialization in reinforcement learning. In *Proceedings of the Seventh International Conference on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*.

Beuls, K. (2012). Inflectional patterns as constructions: Spanish verb morphology in fluid construction grammar. *Constructions and Frames*, 4(2):231–252.

Beuls, K. (2017). An open-ended computational construction grammar for Spanish verb conjugation. *Constructions and Frames*, 9(2):278–301.

Beuls, K., Gerasymova, K., and van Trijp, R. (2010). Situated learning through the use of language games. In *Proceedings of the 19th Annual Machine Learning Conference of Belgium and The Netherlands (BeNeLearn)*.

Beuls, K., Knight, Y., and Spranger, M. (2017). Russian verbs of motion and their aspectual partners in Fluid Construction Grammar. *Constructions and Frames*, 9(2):302–320.

Beuls, K. and Steels, L. (2013). Agent-based models of strategies for the emergence and evolution of grammatical agreement. *PloS one*, 8(3):e58960.

Beuls, K., Van Eecke, P., and Marques, T. (2018). Bidirectional language processing and planning using a construction-based architecture. *In Preparation*.

Beuls, K., Van Trijp, R., and Wellens, P. (2012). Diagnostics and repairs in Fluid Construction Grammar. In Steels, L., editor, *Language Grounding in Robots*, pages 215–234. Springer, Berlin.

Bickerton, D. and Szathmáry, E. (2009). *Biological foundations and origin of syntax*. Strungmann Forum Reports. Mit Press.

Blake, B. J. (2001). *Case*. Cambridge Textbooks in Linguistics. Cambridge University Press, 2 edition.

Bleys, J. (2016). *Language strategies for the domain of colour*. Language Science Press.

Bleys, J., Loetzsch, M., Spranger, M., and Steels, L. (2009). The grounded colour naming game. In *Proceedings of the 18th IEEE International Symposium on Robot and Human Interactive Communication (Ro-man 2009)*.

Bleys, J. and Steels, L. (2009). Linguistic selection of language strategies. In *European Conference on Artificial Life*, pages 150–157. Springer.

Boas, H. C. and Sag, I. A. (2012). *Sign-based construction grammar*. CSLI Publications/Center for the Study of Language and Information.

Boullier, P. (2000). Range concatenation grammars. In *Proceedings of the Sixth International Workshop on Parsing Technologies (IWPT2000)*, pages 53–64.

Briscoe, T. (2000). Grammatical acquisition: Inductive bias and coevolution of language and the language acquisition device. *Language*, 76(2):245–296.

Bulychev, P. and Minea, M. (2008). Duplicate code detection using anti-unification. In *Spring Young Researchers Colloquium on Software Engineering*, pages 51–54.

Bundy, A. and Wallen, L. (1984). Procedural attachment. In *Catalogue of Artificial Intelligence Tools*, pages 98–99. Springer.

Cangalovic, V. S. (2018). Cooking up a grammar. incremental grammar extension for domain-specific parsing. Bachelor Report. Department of Linguistics, University of Bremen.

Carpenter, B. (2005). *The logic of typed feature structures: with applications to unification grammars, logic programs and constraint resolution*, volume 32. Cambridge University Press.

Chang, N. C.-L. (2008). *Constructing grammar: A computational model of the emergence of early constructions*. PhD thesis, University of California, Berkeley.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.

Chomsky, N. (1957). *Syntactic structures*. Mouton.

Chomsky, N. (1986). *Knowledge of language: Its nature, origin, and use*. Greenwood Publishing Group.

Christiansen, M. H., Chater, N., and Reali, F. (2009). The biological and cultural foundations of language. *Communicative & integrative biology*, 2(3):221–222.

Cinque, G. and Kayne, R. S. (2005). *The Oxford handbook of comparative syntax*. Oxford University Press.

Ciortuz, L. (2002a). A framework for inductive learning of typed-unification grammars. *Grammatical Inference: Algorithms and Applications*, pages 334–338.

Ciortuz, L. (2002b). Light–a constraint language and compiler system for typed-unification grammars. In *KI 2002: Advances in Artificial Intelligence: 25th Annual German Conference on AI, KI 2002, Aachen, Germany, September 16-20, 2002. Proceedings*, volume 25, page 3. Springer Science & Business Media.

Ciortuz, L.-V. (2003). Inductive learning of attribute path values in typed-unification grammars. *Sci. Ann. Cuza Univ.*, 13:105–126.

Copestake, A. (2002). *Implementing typed feature structure grammars*, volume 110. CSLI publications Stanford.

Corbett, G. G. (2006). *Agreement*, volume 109 of *Cambridge Textbooks in Linguistics*. Cambridge University Press.

Cornudella, M., Poibeau, T., and Van Trijp, R. (2016). The role of intrinsic motivation in artificial language emergence: a case study on colour. In *26th International Conference on Computational Linguistics (COLING 2016)*, pages 1646–1656.

Cornudella Gaya, M. (2017). *Autotelic Principle: the role of intrinsic motivation in the emergence and development of artificial language*. Theses, Université de recherche Paris Sciences et Lettres.

Cox, M. T., Alavi, Z., Dannenhauer, D., Eyorokon, V., Munoz-Avila, H., and Perlis, D. (2016). Midca: A metacognitive, integrated dual-cycle architecture for self-regulated autonomy. In *AAAI*, pages 3712–3718.

Cox, M. T. and Raja, A. (2011). Metareasoning: An introduction. In Cox, M. T. and Raja, A., editors, *Metareasoning: Thinking about thinking*, pages 3–14. MIT Press, Cambridge, MA.

Croft, W. (2001). *Radical construction grammar: Syntactic theory in typological perspective*. Oxford University Press.

De Beule, J. (2007). *Compositionality, hierarchy and recursion in language. A case study in fluid construction grammar*. PhD thesis, Vrije Universiteit Brussel, Unpublished PhD thesis.

De Beule, J. (2012). A formal deconstruction of Fluid Construction Grammar. In Steels, L., editor, *Computational Issues in Fluid Construction Grammar*. Springer Verlag, Berlin.

De Raedt, L. and Bruynooghe, M. (1992). A unifying framework for concept-learning algorithms. *The Knowledge Engineering Review*, 7(3):251–269.

De Vylder, B. and Tuyls, K. (2006). How to reach linguistic consensus: A proof of convergence for the naming game. *Journal of theoretical biology*, 242(4):818–831.

Dediu, D. (2007). *Non-spurious Correlations Between Genetic and Linguistic Diversities in the Context of Human Evolution*. University of Edinburgh.

DeLong, K. A., Urbach, T. P., and Kutas, M. (2005). Probabilistic word pre-activation during language comprehension inferred from electrical brain activity. *Nature neuroscience*, 8(8):1117.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

Dominey, P., Mealier, A.-L., Pointeau, G., Mirliaz, S., and Finlayson, M. (2017). Dynamic construction grammar and steps towards the narrative construction of meaning. In *2017 AAAI Spring Symposium Series*, pages 163–170.

Dor, D., Knight, C., and Lewis, J. (2014). *The Social Origins of Language*. Oxford Studies in the Evolution of Language. Oxford University Press.

Dubois, V. and Quafafou, M. (2002). Concept learning with approximation: Rough version spaces. In *International Conference on Rough Sets and Current Trends in Computing*, pages 239–246. Springer.

Fanselow, G. (1993). *The parametrization of universal grammar*, volume 8. John Benjamins Publishing.

Feldman, J., Dodge, E., and Bryant, J. (2009). Embodied construction grammar. In Heine, B. and Narrog, H., editors, *The Oxford Handbook of Linguistic Analysis*, pages 121–146. University Press, Oxford.

Feng, C. and Muggleton, S. (2014). Towards inductive generalisation in higher order logic. In *Proceedings of the ninth international workshop on Machine learning*, pages 154–162.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208.

Fillmore, C. J., Kay, P., and O'connor, M. C. (1988). Regularity and idiomaticity in grammatical constructions: The case of let alone. *Language*, pages 501–538.

Fitch, W. (2010). *The Evolution of Language*. Cambridge University Press.

Flach, P. (1994). *Simply Logical. Intelligent Reasoning by Example*. John Wiley & Sons.

Flach, P. (2012). *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press.

Fodor, J. and Sakas, W. (2017). Learnability. In Roberts, I., editor, *The Oxford Handbook of Universal Grammar*. University Press, Oxford.

Galitsky, B., De La Rosa, J. L., and Dobrocsi, G. (2011). Mapping syntactic to semantic generalizations of linguistic parse trees. In *Proceedings of the Twenty-Fourth International Florida Artificial Intelligence Research Society Conference*.

Galitsky, B. A., Ilvovsky, D., Kuznetsov, S. O., and Strok, F. (2014). Finding maximal common sub-parse thickets for multi-sentence search. In *Graph Structures for Knowledge Representation and Reasoning*, pages 39–57. Springer.

Garcia Casademont, E. and Steels, L. (2016). Insight grammar learning. *Journal of Cognitive Science*, 17(1):27–62.

Garoufi, K. and Koller, A. (2010). Automated planning for situated natural language generation. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1573–1582. Association for Computational Linguistics.

Gazdar, G., Klein, E., Pullum, G. K., and Sag, I. A. (1985). *Generalized phrase structure grammar*. Harvard University Press.

Gerasymova, K. and Spranger, M. (2010). Acquisition of grammar in autonomous artificial systems. In Coelho, M., Studer, R., and Woolridge, M., editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI-2010)*, pages 923–928. IOS Press.

Gianollo, C., Guardiano, C., and Longobardi, G. (2008). Three fundamental issues in parametric linguistics. *The limits of syntactic variation*, pages 109–142.

Goldberg, A. E. (1995). *Constructions: A construction grammar approach to argument structure*. University of Chicago Press.

Goldberg, A. E. (2006). *Constructions at work: The nature of generalization in language*. Oxford University Press.

Grice, H. P. (1989). *Studies in the Way of Words*. Harvard University Press.

Griffiths, T. L. and Kalish, M. L. (2007). Language evolution by iterated learning with bayesian agents. *Cognitive science*, 31(3):441–480.

Gust, H., Kühnberger, K.-U., and Schmid, U. (2006). Metaphors and heuristic-driven theory projection (hdtp). *Theoretical Computer Science*, 354(1):98–117.

Heine, B. and Kuteva, T. (2002). *World lexicon of grammaticalization*. Cambridge University Press.

Hobbs, J. R., Stickel, M. E., Appelt, D. E., and Martin, P. (1993). Interpretation as abduction. *Artificial Intelligence*, 63(1-2):69–142.

Hopper, P. (1987). Emergent grammar. In *Annual Meeting of the Berkeley Linguistics Society*, volume 13, pages 139–157.

Jablonka, E. and Lamb, M. (2005). *Evolution in Four Dimensions: Genetic, Epigenetic, Behavioral, and Symbolic Variation in the History of Life*. Life and mind. MIT Press.

Jain, S. and Sharma, A. (1998). Generalization and specialization strategies for learning re languages. *Annals of Mathematics and Artificial Intelligence*, 23(1):1–26.

Jasperson, R., Hayashi, M., and Fox, B. (1994). Semantics and interaction: Three exploratory case studies. *Text-Interdisciplinary Journal for the Study of Discourse*, 14(4):555–580.

Johnson, J., Hariharan, B., van der Maaten, L., Fei-Fei, L., Zitnick, C. L., and Girshick, R. (2017). Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*.

Joshi, A. K., Levy, L. S., and Takahashi, M. (1975). Tree adjunct grammars. *Journal of computer and system sciences*, 10(1):136–163.

Kallmeyer, L. (2010). *Parsing beyond context-free grammars*. Springer, Heidelberg.

Kaplan, R. M. and Bresnan, J. (1982). Lexical-functional grammar: A formal system for grammatical representations. In Bresnan, J., editor, *The Mental reprsentations of grammatical relations*, pages 173–281. MIT Press.

Kapur, S. and Bilardi, G. (1992). Language learning without overgeneralization. *STACS 92*, pages 245–256.

Kay, P. and Fillmore, C. J. (1999). Grammatical constructions and linguistic generalizations: the what's x doing y? construction. *Language*, pages 1–33.

Kirby, S. (1999). Syntax out of learning: The cultural evolution of structured communication in a population of induction algorithms. In Floreano, D., Nicoud, J.-D., and Mondada, F., editors, *Advances in Artificial Life*, pages 694–703, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kirby, S. (2001). Spontaneous evolution of linguistic structure-an iterated learning model of the emergence of regularity and irregularity. *IEEE Transactions on Evolutionary Computation*, 5(2):102–110.

Kirby, S. (2002a). Learning, bottlenecks and the evolution of recursive syntax. In *Linguistic Evolution Through Language Acquisition*. Cambridge University Press.

Kirby, S. (2002b). Learning, bottlenecks and the evolution of recursive syntax. In Briscoe, T., editor, *Linguistic Evolution through Language Acquisition: Formal and Computational Models*, pages 173–204. Cambridge University Press.

Kirby, S., Cornish, H., and Smith, K. (2008). Cumulative cultural evolution in the laboratory: An experimental approach to the origins of structure in human language. *Proceedings of the National Academy of Sciences*, 105(31):10681–10686.

Kirby, S., Griffiths, T., and Smith, K. (2014). Iterated learning and the evolution of language. *Current opinion in neurobiology*, 28:108–114.

Knight, C., Studdert-Kennedy, M., and Hurford, J. (2000). *The Evolutionary Emergence of Language: Social Function and the Origins of Linguistic Form*. Cambridge University Press.

Kübler, S., McDonald, R., and Nivre, J. (2009). *Dependency Parsing*. Morgan and Claypool.

Kutas, M., Van Petten, C. K., and Kluender, R. (2006). Psycholinguistics electrified ii (1994–2005). In *Handbook of Psycholinguistics (Second Edition)*, pages 659–724. Elsevier.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64.

Lestrade, S. (2015a). A case of cultural evolution: The emergence of morphological case. *Linguistics in the Netherlands*, 32(1):105–115.

Lestrade, S. (2015b). Simulating the development of bound person marking. In *Baayen, H.; Jäger, G.; Köllner, M.(ed.), Proceedings of the 6th Conference on Quantitative Investigations in Theoretical Linguistics*. Tuebingen: University of Tuebingen.

Lestrade, S. (2016). The emergence of argument marking. In Roberts, S., Cuskley, C., McCrohon, L., Barceló-Coblijn, L., Fehér, O., and Verhoef, T., editors, *The Evolution of Language: Proceedings of the 11th International Conference (EVOLANGX11)*. Online at `http://evolang.org/neworleans/papers/36.html`.

Lestrade, S. (2017). Mole: Modeling language evolution. R package. https://CRAN.R-project.org/package=MoLE.

Liu, R.-R., Jia, C.-X., Yang, H.-X., and Wang, B.-H. (2009). Naming game on small-world networks with geographical effects. *Physica A: Statistical Mechanics and its Applications*, 388(17):3615–3620.

Loetzsch, M., Bleys, J., and Wellens, P. (2009). Understanding the dynamics of complex lisp programs. In *Proceedings of the 2nd European Lisp Symposium*, pages 59–69.

Loetzsch, M., Wellens, P., De Beule, J., Bleys, J., and Van Trijp, R. (2008). The babel2 manual. *Technical Report AI-Memo 01-08*.

Lüngen, H. and Sporleder, C. (1999). Automatic induction of lexical inheritance hierarchies. *Multilinguale Corpora. Codierung, Strukturierung, Analyse. Prague: Enigma Corporation*.

Maes, P. and Nardi, D. (1988). *Meta-level architectures and reflection*. Elsevier Science Pub. Co. Inc., New York, NY.

Malchukov, A. and Spencer, A. (2009). *The Oxford handbook of case*. Oxford University Press.

Malik Ghallab, C. I., Penberthy, S., Smith, D., Sun, Y., and Weld, D. (1998). Pddl-the planning domain definition language. Technical report, Technical report, Yale Center for Computational Vision and Control.

Marques, T. and Beuls, K. (2016). A construction grammar approach for pronominal clitics in european portuguese. In *International Conference on Computational Processing of the Portuguese Language*, pages 239–244. Springer.

Michalski, R. S. (1983). A theory and methodology of inductive learning. *Artificial intelligence*, 20(2):111–161.

Mitchell, T. (1978). Version spaces: an approach to concept learning. Technical report, Stanford University Department of Computer Science.

Mitchell, T. (1982). Generalization as search. *Artificial intelligence*, 18(2):203–226.

Moens, M., Calder, J., Klein, E., Reape, M., and Zeevat, H. (1989). Expressing generalizations in unification-based grammar formalisms. In *Proceedings of the Fourth Conference on European Chapter of the Association for Computational Linguistics*, EACL '89, pages 174–181, Stroudsburg, PA, USA. Association for Computational Linguistics.

Muggleton, S. and De Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679.

Newell, A., Shaw, J. C., and Simon, H. A. (1957). Empirical explorations of the logic theory machine: A case study in heuristic. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), pages 218–230, New York, NY, USA. ACM.

Nilsson, N. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Pub. Co.

Nivre, J. (2006). *Inductive Dependency Parsing*. Springer.

Oates, T., Armstrong, T., Bonache, L. B., and Atamas, M. (2006). Inferring grammars for mildly context sensitive languages in polynomial-time. In *ICGI*, volume 4201, pages 137–147. Springer.

Pauw, S. et al. (2013). *Size Matters: Grouding Quantifiers in Spatial Perception*. ILLC.

Pauw, S. and Hilferty, J. (2012). The emergence of quantifiers. *Experiments in cultural language evolution*, 3:277.

Pednault, E. P. (1987). Formulating multiagent, dynamic-world problems in the classical planning framework. *Reasoning about actions and plans*, pages 47–82.

Pereira, F. C. and Warren, D. H. (1980). Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, 13(3):231–278.

Pickering, M. J. and Garrod, S. (2013). Forward models and their implications for production, comprehension, and dialogue. *Behavioral and Brain Sciences*, 36(4):377–392.

Plotkin, G. D. (1970). A note on inductive generalization. *Machine intelligence*, 5(1):153–163.

Plotkin, G. D. (1971). A further note on inductive generalization. *Machine intelligence*, 6(101-124).

Pollard, C. (1997). Lectures on the foundations of hpsg. https://www1.essex.ac.uk/linguistics/external/clmt/papers/hpsg/pollard-foundations.ps.

Pollard, C. and Sag, I. A. (1994). *Head-driven phrase structure grammar*. University of Chicago Press.

Powers, D. M., Matsumoto, T., and Jarrad, G. (2003). Application of search algorithms to natural language processing. In *Australasian Language Technology Workshop 2003*. Australian Language Technology Associations.

Puglisi, A., Baronchelli, A., and Loreto, V. (2008). Cultural route to the emergence of linguistic categories. *Proceedings of the National Academy of Sciences*, 105(23):7936–7940.

Rendell, L. (1986). A general framework for induction and a study of selective induction. *Machine Learning*, 1(2):177–226.

Reynolds, J. C. (1970). Transformational systems and the algebraic structure of atomic formulas. *Machine intelligence*, 5:135–152.

Roberts, I. (2017). *The Oxford Handbook of Universal Grammar*. University Press, Oxford.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

Schmill, M., Oates, T., Anderson, M. L., Josyula, D., Perlis, D., Wilson, S., and Fults, S. (2008). The role of metacognition in robust ai systems. In *Workshop on Metareasoning at the Twenty-Third AAAI Conference on Artificial Intelligence*.

Schueller, W. and Oudeyer, P.-Y. (2016). Active control of complexity growth in naming games: Hearer's choice. In Roberts, S., Cuskley, C., McCrohon, L., Barceló-Coblijn, L., Fehér, O., and Verhoef, T., editors, *The Evolution of Language: Proceedings of the 11th International Conference (EVOLANGX11)*. Online at `http://evolang.org/neworleans/papers/105.html`.

Shieber, S. M. (1985). Evidence against the context-freeness of natural language. In *Philosophy, Language, and Artificial Intelligence*, pages 79–89. Springer.

Sierra-Santibáñez, J. (2012). A logic programming approach to parsing and production in Fluid Construction Grammar. In Steels, L., editor, *Computational Issues in Fluid Construction Grammar*. Springer Verlag, Berlin.

Sierra-Santibáñez, J. (2014). An agent-based model studying the acquisition of a language system of logical constructions. In *Proceedings of the National Conference on Artificial Intelligence*, pages 350–357.

Sierra-Santibáñez, J. (2018). An agent-based model of the emergence and evolution of a language system for boolean coordination. *Autonomous Agents and Multi-Agent Systems*, 32(4):417–458.

Smith, K. (2002). The cultural evolution of communication in a population of neural networks. *Connection Science*, 14(1):65–84.

Smith, K. (2004). The evolution of vocabulary. *Journal of theoretical biology*, 228(1):127–142.

Smith, K., Brighton, H., and Kirby, S. (2003). Complex systems in language evolution: the cultural emergence of compositional structure. *Advances in Complex Systems*, 6(4):537–558.

Smolka, G. and Ait-Kaci, H. (1989). Inheritance hierarchies: Semantics and unification. *Journal of Symbolic Computation*, 7(3-4):343–370.

Spranger, M. (2012). The co-evolution of basic spatial terms and categories. In Steels, L., editor, *Experiments in Cultural Language Evolution*, number 3 in Advances in Interaction Studies, pages 111–141. John Benjamins.

Spranger, M. (2016). *The evolution of grounded spatial language*. Language Science Press.

Spranger, M., Loetzsch, M., and Steels, L. (2012a). A perceptual system for language game experiments. In Steels, L. and Hild, M., editors, *Language Grounding in Robots*, pages 89–110. Springer.

Spranger, M., Pauw, S., and Loetzsch, M. (2010). Open-ended semantics co-evolving with spatial language. In Smith, A. D. M., Schouwstra, M., de Boer, B., and Smith, K., editors, *The Evolution of Language (Evolang 8)*, pages 297–304, Singapore. World Scientific.

Spranger, M., Pauw, S., Loetzsch, M., and Steels, L. (2012b). Open-ended Procedural Semantics. In Steels, L. and Hild, M., editors, *Language Grounding in Robots*, pages 153–172. Springer.

Steedman, M. (2000). *The syntactic process*. MIT press.

Steels, L. (1995). A self-organizing spatial vocabulary. *Artificial life*, 2(3):319–332.

Steels, L. (1997). Self-organizing vocabularies. In *Artificial Life V*, pages 179–184.

Steels, L. (1998). Synthesising the origins of language and meaning using co-evolution, self-organisation and level formation. *Approaches to the Evolution of Language*, pages 384–404.

Steels, L. (1999). The puzzle of language evolution. *Kognitionswissenschaft*, 8(4):143–150.

Steels, L. (2000). The emergence of grammar in communicating autonomous robotic agents. In Horn, W., editor, *ECAI 2000: Proceedings of the 14th European Conference on Artificial Life*, pages 764–769, Amsterdam. IOS Publishing.

Steels, L. (2003). Language-reentrance and the 'inner voice'. *Journal of Consciousness Studies*, 10(4-5):173–185.

Steels, L. (2007). The recruitment theory of language origins. In Lyon, C., Nehaniv, C. L., and Cangelosi, A., editors, *Emergence of Language and Communication*, pages 129–151. Springer, Berlin.

Steels, L., editor (2011a). *Design Patterns in Fluid Construction Grammar*. John Benjamins, Amsterdam.

Steels, L. (2011b). Modeling the cultural evolution of language. *Physics of Life Reviews*, 8(4):339–356.

Steels, L., editor (2012a). *Computational Issues in Fluid Construction Grammar*. Springer Verlag, Berlin.

Steels, L. (2012b). *Experiments in cultural language evolution*, volume 3. John Benjamins Publishing.

Steels, L. (2012c). Self-organization and selection in cultural language evolution. In Steels, L., editor, *Experiments in Cultural Language Evolution*, pages 1–37. John Benjamins, Amsterdam.

Steels, L. (2012d). Self-organization and selection in cultural language evolution. In Steels, L., editor, *Experiments in Cultural Language Evolution. John Benjamins, Amsterdam*. John Benjamins, Amsterdam.

Steels, L. (2015). *The Talking Heads experiment: Origins of words and meanings*, volume 1. Language Science Press.

Steels, L. (2017). Basics of fluid construction grammar. *Constructions and frames*, 9(2):178–225.

Steels, L., Belpaeme, T., et al. (2005). Coordinating perceptually grounded categories through language: A case study for colour. *Behavioral and brain sciences*, 28(4):469–488.

Steels, L. and De Beule, J. (2006). Unify and merge in fluid construction grammar. In *Symbol grounding and beyond*, pages 197–223. Springer.

Steels, L. and Garcia Casademont, E. (2015a). Ambiguity and the origins of syntax. *The Linguistic Review*, 32(1):37–60.

Steels, L. and Garcia Casademont, E. (2015b). How to play the syntax game. In *Proceedings of the European Conference on Artificial Life*, pages 479–486.

Steels, L. and Loetzsch, M. (2012). The grounded naming game. *Experiments in cultural language evolution*, 3:41–59.

Steels, L. and Spranger, M. (2008). The robot in the mirror. *Connection Science*, 20(4):337–358.

Steels, L. and Van Eecke, P. (2018). Insight grammar learning using pro-unification and anti-unification in fluid construction grammar. *Submitted*.

Steels, L. and van Trijp, R. (2011). How to make construction grammars fluid and robust. In Steels, L., editor, *Design Patterns in Fluid Construction Grammar*, pages 301–330. John Benjamins, Amsterdam.

Tesnière, L. (1965). *Eléments de syntaxe structurale*. éd. Klincksieck, Paris.

Thomas, B. (1999). Anti-unification based learning of t-wrappers for information extraction. In *Proc. AAAI-99 Workshop on Machine Learning for Information Extraction*.

Tomasello, M. (2003). *Constructing a Language: A Usage-Based Theory of Language Acquisition*. Harvard University Press.

Traugott, E. C. and Trousdale, G. (2013). *Constructionalization and constructional changes*, volume 6. Oxford University Press.

Valin, R. D. v. and LaPolla, R. J. (1997). *Syntax: Structure, Meaning, and Function*. Cambridge Textbooks in Linguistics. Cambridge University Press.

Van Eecke, P. (2015). Achieving robustness through the integration of production in comprehension. In *Proceedings of the EuroAsianPacific Joint Conference on Cognitive Science*, pages 187–192.

Van Eecke, P. (2017). Robust processing of the Dutch verb phrase. *Constructions and Frames*, 9(2):226–250.

Van Eecke, P. and Beuls, K. (2017). Meta-layer problem solving for computational construction grammar. In *2017 AAAI Spring Symposium Series*, pages 258–265.

Van Eecke, P. and Beuls, K. (2018). Exploring the creative potential of computational construction grammar. *Zeitschrift für Anglistik und Amerikanistik*, 66(3):341–355.

van Trijp, R. (2008). The emergence of semantic roles in fluid construction grammar. In *The Evolution Of Language*, pages 346–353. World Scientific.

van Trijp, R. (2011). Feature matrices and agreement: A case study for German case. In Steels, L., editor, *Design Patterns in Fluid Construction Grammar*, pages 205–235. John Benjamins, Amsterdam.

van Trijp, R. (2012). A reflective architecture for robust language processing and learning. In Steels, L. and Hild, M., editors, *Computational issues in Fluid Construction Grammar*, pages 51–74. Springer.

van Trijp, R. (2013). Linguistic assessment criteria for explaining language change: A case study on syncretism in german definite articles. *Language Dynamics and Change*, 3(1):105–132.

van Trijp, R. (2014). Long-distance dependencies without filler- gaps: a cognitive-functional alternative in fluid construction grammar. *Language and Cognition*, 6(02):242–270.

van Trijp, R. (2016). *The evolution of case grammar*. Language Science Press.

van Trijp, R. (2017). A computational construction grammar for English. In *2017 AAAI Spring Symposium Series*, pages 266–273.

Vera, J. (2018). An agent-based model for the role of short-term memory enhancement in the emergence of grammatical agreement. *Artificial life*, 24(02):119–127.

Vijay-Shanker, K., Weir, D. J., and Joshi, A. K. (1987). Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the 25th annual meeting on Association for Computational Linguistics*, pages 104–111. Association for Computational Linguistics.

Wellens, P. (2011). Organizing constructions in networks. In Steels, L., editor, *Design Patterns in Fluid Construction Grammar*, pages 181–201. John Benjamins, Amsterdam.

Wellens, P. (2012). *Adaptive Strategies in the Emergence of Lexical Systems*. PhD thesis.

Zock, M., Francopoulo, G., and Laroui, A. (1988). Language learning as problem solving: Modelling logical aspects of inductive learning to generate sentences in french by man and machine. In *Proceedings of the 12th conference on Computational linguistics-Volume 2*, pages 806–811. Association for Computational Linguistics.

Appendix A

List of Publications

The papers that I have published during my PhD project (2014-2018) are listed below. A complete list of my talks, posters and research activities is accessible via `https://ai.vub.ac.be/members/paul`.

- Beuls, Katrien, Steels, Luc & **Van Eecke, Paul**. (forthcoming). Strategies for interactive task learning and teaching. In: *Interactive Task Learning: Agents, Robots, and Humans Acquiring New Tasks through Natural Interactions*, edited by K. A. Gluck and J. E. Laird. Strüngmann Forum Reports, vol. 26, J. R. Lupp, series editor. Cambridge, MA: MIT Press.

- Shah, Julie, Gluck, Kevin, Belpaeme, Tony, Koedinger, Kenneth, Rohlfing, Katharina, van der Maas, Han, **Van Eecke, Paul**, VanLehn, Kurt, Vollmer, Anna-Lisa & Yee-King, Matthew. (forthcoming). Task Instruction. In: *Interactive Task Learning: Agents, Robots, and Humans Acquiring New Tasks through Natural Interactions*, edited by K. A. Gluck and J. E. Laird. Strüngmann Forum Reports, vol. 26, J. R. Lupp, series editor. Cambridge, MA: MIT Press.

- **Van Eecke, Paul** & Beuls, Katrien. (2018). Exploring the creative potential of computational construction grammar. *Zeitschrift für Anglistik und Amerikanistik* 66(3): 341-355.

- **Van Eecke, Paul**. (2017). Robust processing of the Dutch verb phrase. *Constructions and Frames* 9(2): 226-250.

- **Van Eecke, Paul** & Beuls, Katrien. (2017). Meta-Layer problem solving for computational construction grammar. *The 2017 AAAI 2017 Spring Symposium on Computational Construction Grammar and Natural Language Understanding.*

Technical Report SS-17-02: 258-265.

- Hoorens, Sébastien, **Van Eecke, Paul** & Beuls, Katrien. (2017). Constructions at work! Visualising linguistic pathways for computational construction grammar. *Proceedings of the 29th Benelux Conference on Artificial Intelligence*: 224-237.

- **Van Eecke, Paul** & Fernandez, Raquel. (2016). On the influence of gender on interruptions in multiparty dialogue. *Proceedings of Interspeech 2016*: 2070-2074.

  Van Eecke, Paul & Steels, Luc. (2016). The role of pro- and anti-unification in insight grammar learning. *BeNeLearn: Belgian and Netherlands Conference on Machine Learning*.

- Adrian, Kemo, Bilgin, Aysenur & **Van Eecke, Paul**. (2016). A semantic distance based architecture for a guesser agent in ESSENCE's location taboo challenge. *Diversity @ ECAI International Workshop on Diversity-Aware Artificial Intelligence Workshop Proceedings*: 33-39.

- Cornudella, Miquel, **Van Eecke, Paul** & van Trijp, Remi. (2015). How intrinsic motivation can speed up language emergence. *Proceedings of the European Conference on Artificial Life 2015*: 571-578.

- **Van Eecke, Paul**. (2015). Achieving Robustness through the Integration of Language Production in Comprehension. *Proceedings of the EuroAsianPacific Joint Conference on Cognitive Science. CEUR Workshop Proceedings 141*: 187-192.

Index