Centro de Informática U·F·P·E

Pós-Graduação em Ciência da Computação

FORMALIZATION OF CONTEXT-FREE LANGUAGE THEORY

by

Marcus Vinícius Midena Ramos

Ph.D. Thesis

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2016

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

Marcus Vinícius Midena Ramos

FORMALIZATION OF CONTEXT-FREE LANGUAGE THEORY

Ph.D. Thesis presented to the Centro de Informática of the Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Ciência da Computação.

Advisor: *Ruy J. G. B. de Queiroz*

RECIFE
2015

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

R175f   Ramos, Marcus Vinícius Midena.
Formalization of context-free language theory / Marcus Vinícius Midena Ramos. – 2016.
292 p.: il., fig., tab.

Orientador: Ruy José Guerra Barretto de Queiroz.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.
Inclui referências e apêndices.

1. Ciência da computação. 2. Teoria da computação. 3. Lógica matemática. I. Queiroz, Ruy José Guerra Barretto de. (orientador). II. Título.

004   CDD (23. ed.)   UFPE- MEI 2016-048

Marcus Vinícius Midena Ramos

Formalization of Context-Free Language Theory

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de **Doutor** em Ciência da Computação.

Aprovado em: 18/01/2016.

To Vinícius (in memoriam), for being a life example.
To Elizabeth, for a life of love and care.
To Leonardo, for whom I always want to make my best.

Acknowledgements

My thanks to Professor Ruy José Guerra Barretto de Queiroz for this support in all stages of this project, the weightings and guidance on the best ways to be followed when there was more than one possibility under consideration, and also the enthusiasm shown with each accomplishment.

I leave here registered my thanks to the Colegiado de Engenharia de Computação of UNIVASF, as well as to UNIVASF itself, who welcomed and authorized my departure from my regular activities during the period between July of 2013 and June of 2016, so that I could devote myself in full to my research.

I am grateful to the members of the board of the thesis proposal exam (formed by professors Mauricio Ayala Rincón, of the Departments of Computer Science and Mathematics of UNB, professor Flávio Leonardo Cavalcanti de Moura, of the Department of Computer Science of UNB, and professor Luiz Frederico Goncalves de Freitas, from UFPE's Centro de Informática) held in February of 2014 in the Centro Informática of UFPE. Their questions and critical reviews of the monograph presented at the time were very important to correct the course of the work, allowing me to fully understand the real complexity of the proposed task and to improve the presentation of the results. I could not fail to thank also to the many anonymous reviewers who, from their critical evaluations of the various articles submitted to events worldwide, have provided valuable suggestions for the improvement of the formalization and its presentation. Likewise, I thank the many members of the discussion forums `coq-club` and `stackoverflow`, which were of great value in solving problems and clarifying specific questions about the use of the Coq proof assistant.

I also thank Pierre-Louis Curien, Hugo Herbelin and Paul-André Melliès for the financial support offered, which enabled my participation in the event *Semantics of proofs and certified mathematics* held in Paris between April and July of 2014, and also in the *CIRM Summer Pre-school* held in Marseille in April of 2014. Finally, I express my gratitude to Yves Bertot, from INRIA Sophia-Antipolis in France, who kindly welcomed me to a meeting in April of 2014, heard about my project and gave me important guidelines for its development, as well as to Gianluigi Bellin, professor at the University of Verona (Italy), a fellow student of Coq in Marseille and Paris, and a great enthusiast and supporter of my research project from the beginning.

At least three people have played a direct and fundamental role in order to enable the achievement of the objectives of this project. Initially, I would like to express my sincere and deep thanks to Arthur Azevedo de Amorim, from the University of Pennsylvania, who, in Paris in April 2014, pointed out to me the ways for the development of the formalization, gave me the initial guidelines on the use of Coq and patiently made himself available to clarify doubts to me. After that, he even helped me via email at various times throughout the work, always with utmost kindness and great interest. Without his initial orientation and subsequent help this thesis would never exist.

The increasing size and complexity of the work made me reach a point where the need to share live experiences with researchers with expertise in the area became vital for its continuity. For this, I had the good fortune and privilege to be accepted by professor Nelma Moreira, of the Departmento de Ciência de Computadores of the Faculdade de Ciências of the Universidade do Porto (Porto, Portugal), for a period of co-supervision that extended between the months of March and August of 2015. During this period, I also had the honor of being introduced to and work under the supervision of professor José Carlos Bacelar Almeida, of the Departmento de Informática of the Universidade do Minho (Braga, Portugal). Thanks to them, I had months of hard work and high productivity, which resulted in achieving all the goals that were initially formulated. To professors Nelma and Bacelar I record here my eternal gratitude and recognition for the way they welcomed me, the availability shown at all times and for the interest and enthusiasm with which they guided me to the completion of the work.

I am deeply grateful to my friend and professor of the Centro de Informática of the UFPE, Alex Sandro Gomes, for the various and detailed critical reviews of this thesis, which allowed me to improve decisively and substantially the quality of the presentation of the process lived and the results obtained. Also to the friend, former undergraduate colleague and professor at Escola Politécnica of USP, Romero Tori, for the critical reading and important suggestions of improvement.

I would like to place on record a special thanks to my friend Roseli Tuan. It was in a night, during a long distance phone call, in which I expressed my disappointment with the then difficulty experienced by me in order to confirm my trip to Portugal, that she pointed me alternatives that finally resulted in the realization of it. Thus, the period in Portugal, the job completion and this thesis are the direct fruits of that phone call and, above all, the affection, friendship and concern always shown by her for me. For all this, my eternal thanks. Likewise, I thank my brother Paulo Ramos, who was enthusiastic about the possibility of my trip to Portugal from the beginning, encouraged me continuously to its achievement, even when the family situation recommended that it be at least postponed, and helped me in the initial arrangements relating to my arrival there. I am very grateful to Robson Medeiros Calado and Alexana Vilar Soares Calado for the long time friendship and the support offered in the various phases of this project, between 2011 and 2016, while I was in Recife. Thank you so much to Cecilio Ricardo Bastos and Flavia Ramos Bastos for taking care of my home while I was away in France in 2014, in Portugal in 2015 and in many other occasions. Lastly, thanks to my son Leonardo Ramos for his patience and understanding with the times we failed to enjoy together due to the numerous demands of this job.

“Never give up, never surrender”

— GALAXY QUEST (1999)

Resumo

Assistentes de prova são ferramentas de software que são usadas na mecanização da construção e da validação de provas na matemática e na ciência da computação, e também no desenvolvimento de programas certificados. Diferentes ferramentas estão sendo usadas de forma cada vez mais frequente para acelerar e simplificar a verificação de provas, e o assistente de provas Coq é uma das mais conhecidas e utilizadas. A teoria de linguagens e de autômatos é uma área bem estabelecida da matemática, com relevância para os fundamentos da ciência da computação e a tecnologia da informação. Em particular, a teoria das linguagens livres de contexto é de fundamental importância na análise, no projeto e na implementação de linguagens de programação de computadores. Este trabalho descreve um esforço de formalização, usando o assistente de provas Coq, de resultados fundamentais da teoria clássica das gramáticas e linguagens livres de contexto. Estes incluem propriedades de fechamento (união, concatenação e estrela de Kleene), simplificação gramatical (eliminação de símbolos inúteis, de símbolos inacessíveis, de regras vazias e de regras unitárias), a existência da Forma Normal de Chomsky para gramáticas livres de contexto e o Lema do Bombeamento para linguagens livres de contexto. Para alcançar estes resultados, diversas etapas precisaram ser cumpridas, incluindo (i) o entendimento das características, da importância e dos benefícios da formalização matemática, especialmente na ciência da computação, (ii) a familiarização com as teorias matemáticas fundamentais utilizadas pelos assistentes de provas, (iii) a familiarização com o assistente de provas Coq, (iv) a revisão das estratégias usadas nas provas informais dos principais resultados da teoria das linguagens livres de contexto e, finalmente, (v) a seleção e adequação das estratégias de representação e prova adotadas para permitir o alcance dos resultados pretendidos. O resultado é um importante conjunto de bibliotecas cobrindo os principais resultados da teoria das linguagens livres de contexto, com mais de 500 lemas e teoremas totalmente provados e verificados. Esta é provavelmente a formalização mais abrangente da teoria clássica das linguagens livres de contexto jamais feita no assistente de provas Coq, e inclui o importante resultado que é a formalização do Lema do Bombeamento para linguagens livres de contexto. As perspectivas para novos desenvolvimentos a partir deste trabalho são diversas e podem ser agrupadas em três áreas diferentes: inclusão de novos dispositivos e resultados, extração de código e aprimoramentos gerais das suas bibliotecas.

Palavras-chave: Assistentes de provas. Coq. Formalização. Teoria de linguagens. Linguagens livres de contexto. Gramáticas livres de contexto.

Abstract

Proof assistants are software-based tools that are used in the mechanization of proof construction and validation in mathematics and computer science, and also in certified program development. Different such tools are being increasingly used in order to accelerate and simplify proof checking, and the Coq proof assistant is one of the most known and used. Language and automata theory is a well-established area of mathematics, relevant to computer science foundations and information technology. In particular, context-free language theory is of fundamental importance in the analysis, design and implementation of computer programming languages. This work describes a formalization effort, using the Coq proof assistant, of fundamental results of the classical theory of context-free grammars and languages. These include closure properties (union, concatenation and Kleene star), grammar simplification (elimination of useless symbols, inaccessible symbols, empty rules and unit rules), the existence of a Chomsky Normal Form for context-free grammars and the Pumping Lemma for context-free languages. To achieve this, several steps had to be fulfilled, including (i) understanding of the characteristics, importance and benefits of mathematical formalization, specially in computer science, (ii) familiarization with the underlying mathematical theories used in proof assistants, (iii) familiarization with the Coq proof assistant, (iv) review of the strategies used in the informal proofs of the main results of the context-free language theory and finally (iv) selection and adequation of the representation and proof strategies adopted in order the achieve the desired objectives. The result is an important set of libraries covering the main results of context-free language theory, with more than 500 lemmas and theorems fully proved and checked. This is probably the most comprehensive formalization of the classical context-free language theory in the Coq proof assistant done to the present date, and includes the remarkable result that is the formalization of the Pumping Lemma for context-free languages. The perspectives for the further development of this work are diverse and can be grouped in three different areas: inclusion of new devices and results, code extraction and general enhancements of its libraries.

Keywords: Proof assistants. Coq. Formalization. Language theory. Context-free languages. Context-free grammars.

List of Figures

List of Tables

List of Acronyms

CC Calculus of Constructions

CIC Calculus of Inductive Constructions

CFG Context-Free Grammar

CFL Context-Free Language

CNF Chomsky Normal Form

GNF Greibach Normal Form

IDE Interactive Development Environment

PDA Pushdown automaton

PL Pumping Lemma for Context-Free Languages

Contents

1 Introduction

The fundamental mathematical activity of stating and proving theorems has been traditionally done by professionals that rely purely on their own personal efforts in order to accept or refuse any new proposal, after extensive manual checking. This style of work, which has been used for centuries by mathematicians all over the world, is now changing thanks to computer technology support.

The so called “proof assistants” are software tools that are executed in regular computers and offer their users a friendly and fully checked environment where one can describe mathematical objects and its properties, and then prove theorems about them. The main advantage of their use is to guarantee that proofs are correctly constructed and fully checked. Support for proof automation (that is, automatic proof construction) is also provided in different levels, but this is not the main focus. When applied to program development, these tools are also helpful in checking the correctness of an existing program and also in the construction of correct programs. In order to obtain these benefits, however, the user must first be familiar with the underlying mathematical theory of the chosen tool, as well as the associated languages and methodologies required to obtain these proofs.

Language theory is a fundamental area in mathematics and computer science, which started in the 1950s and was extensively developed during the 1960s and 1970s. New application domains, such as bioinformatics, web networking and model checking, have led to new reasearch and developments in this area in the last 20 years. Language theory provides important results about the study and development of artificial languages, as well as language processing technology, and relevant conclusions about the limits and properties of the computation process itself. In particular, context-free language theory is key in the areas of programming language analysis, design and implementation. Despite its huge number of results, in the form of notations, devices, lemmas and theorems, which are fundamental to both theoretical computer science and practical computer technology, this area has not received too much attention in respect to formalization and development of fully checked demonstrations. As it will become clear in Chapter 4, the first efforts date from mid-1980s but only since 2010 a few more comprehensive works in this direction have appeared.

This work is about the mathematical formalization of an important subset of the context-free language theory, including some of its most important results such as the Chomsky Normal Form and the Pumping Lemma. The formalization has been done in the Coq proof assistant, using its underlying mathematical theory, the Calculus of Inductive Constructions. This represents a novel approach towards formal language theory, specially context-free language theory, as virtually all textbooks, general literature and classes on the subject rely on an informal (traditional) mathematical approach. The objective of this work, thus, is to elevate the status of this theory to new levels in accordance with the state-of-the-art in mathematical accuracy, which is accomplished with the use of interactive proof assistants.

Before discussing the formalization, however, the mathematical framework on which it has been developed is introduced. This discussion starts with a brief overview of the characteristics, applications and importance of interactive proof assistants in general. Proof assistants, although available in different forms and flavors, share many characteristics, and this is the subject of Chapter 2.

The acceptance, usage and importance of proof assistants in computer formalizations and machine-checked developments are increasing very rapidly, as well as the complexity of the projects that are being addressed by such tools. This can be confirmed by the projects (mostly in Coq) presented and discussed in Chapter 3.

The mathematical framework and the tool used in the formalization are presented in Appendix A and Appendix B. In both cases, a tutorial style was adopted in order to ease the understanding for the readers not familiar with the topics, and can be skipped by the experts.

Appendix A presents the theoretical background required to understand and use modern proof assistants in general, and Coq in particular. Although this technology is recent (first experiments date from late 1960s), the theory behind them dates from the beginning of the 20th century and should be considered independently. The structure of this appendix follows more or less the logical development and interrelationship of its constituents: propositional and predicate logic, natural deduction, the lambda calculus of Church (both untyped and typed versions), the Curry-Howard Correspondence, and finally the Calculus of Inductive Constructions.

A brief description of the Coq proof assistant, which has been used in the present formalization, is presented in Appendix B. This includes discussions about the interactive environment used (CoqIDE), about its two languages Vernacular and Gallina and also about the commands and techniques used for reasoning and computing in a formalization.

Formal language and automata theory formalization is not a completely new area of research. In Chapter 4, a summary of these accomplishments is presented, which are then considered in the context of our own proposal. It shows that most of the formalization effort on general formal language theory up to date has been dedicated to the regular language theory, and not so much to context-free language theory. Thus, this constitutes the motivation for the present work.

Chapter 5 discusses the importance and scope of the theory formalized in this work, including its historical role in the sudy and implementation of artificial languages, specially programming languages, and its relation to other language classes (Chomsky Hierarchy). It is a brief review of the devices included in the theory, along with their properties and main theoretical results, which serves as a guideline for the formalization presented in Chapter 6.

Now, the idea of formalizing the context-free language theory in the Coq proof assistant can be addressed in Chapter 6. In particular, the presentation includes the strategy and the method adopted, the main definitions used, the lemmas and theorems that were proved, and insights into the demonstrations that were constructed. Intermediate results of this work were presented in the 9th and 10th Workshop on Logical and Semantic Frameworks with Applications (LSFA), respectively Ramos and Queiroz (2014) and Ramos and Queiroz (2015). A list of the most important lemmas and theorems of the main libraries of this work is presented in Appendix C, along with brief descriptions.

Final conclusions are presented in Chapter 7, and include a presentation and a discussion of the main contributions of this work. Among them, the fact that this is probably the most comprehensive formalization of the classical context-free language theory done to the present date in the Coq proof assistant, and also the remarkable result that is the second ever formalization of the important Pumping Lemma for context-free languages (the first in the Coq proof assistant).

Despite the extensive formalization results already obtained, this project can still evolve in many different directions, including code extraction for some operations, the formalization of other grammar-related results of context-free language theory, the consideration of pushdown automata and its relation to context-free grammars and the reconstruction and reorganization of the whole development using the SSReflect plugin for Coq. These are the starting points for new and related research projects, which are discussed also in Chapter 7.

2 Proof Assistants

Proof assistants are software tools that enable the construction of formal proofs, expressed in a specialized formal language, and the verification of their correctness (GEUVERS, 2009). They represent the implementation of some formal mathematics in an interactive environment where the user can formalize his theory or develop certified software.

Proof assistants enable, first of all, the user to describe the objects of his/her universe of discourse. Then, to describe their behaviour and, finally, to prove properties exhibited by these objects in the form of mathematical lemmas and theorems. All this, however, must be done in a proper mathematical language, with a friendly syntax and a consistent and precise semantics, as discussed in Appendix A.

This chapter is an introduction to proof assistants in general. In Section 2.1 we disccuss the general characteristics of a typical proof assistant, regardless of the choice. We include, in Section 2.2, a discussion about the benefits and drawbacks of using proof assistants and, in Section 2.3, an overview of the first and influentital Automath automated proof checker. We end, in Section 2.4, with a brief reference to the QED Manifesto, a historical document that expresses the most pure and ideal objectives concerning mathematical formalization in various aspects.

2.1 Characterization

Proof assistants (also known as "interactive theorem provers") are computer-based software tools that help users to state and proof results in a certain theory (GEUVERS, 2009). They date from as far as 1967, when de Bruijn launched the Automath project (BRUIJN, 1983), and their development follows closely the development of computers and computer software themselves. Since then, they have evolved a lot and many such systems are available today (WIEDIJK, 2006). Among the most known and influential, it is possible to mention Coq (BERTOT; CASTÉRAN, 2004; THE COQ DEVELOPMENT TEAM, 2015; INRIA, 2014), ACL2 (KAUFMANN; MOORE; MANOLIOS, 2000), Automath (VARIOUS, 1994), Agda (AGDA, 2015), HOL (VARIOUS, 1993), Isabelle (NIPKOW; WENZEL; PAULSON, 2002), LCF (GORDON, 2000), Matita (ASPERTI et al., 2011), Mizar (BONARSKA, 1990), Nqthm

(BOYER; MOORE, 1998), Nuprl (CONSTABLE et al., 1986) and PVS (PVS, 2015). Each has its own specific features and is based on a particular mathematical calculus. Some of them have inspired the development of the others. For most of them, a brief history of their main characteristics, usage, evolution and perspectives can be found in Geuvers (2009) and also in Wiedijk (2006).

Basically, a proof assistant is an interactive tool that allows the user to write expressions that represent propositions/types and proofs/terms in its input language. Proofs/terms can then be interactively built from propositions/types, and/or checked against them.

In the general case, proofs/terms can be built directly or indirectly. In the direct approach, the proof or term is written directly by the user in the formal language adopted by the proof assistant. In the indirect approach, the proof or term is constructed according to a set of high-level commands issued by the user (also called "tactics"). Tactics exist to ease the construction of proofs/terms, which may become quite long, complex and difficult to read very rapidly. In the indirect approach, proofs/terms are partially built by the result of applying tactics to the current goal in a proof session. New subgoals might be generated as the session continues. When no more subgoals are generated, the proof/term is complete. In any case (direct or indirect approach) the proof assistant checks that the constructed proof/term complies to the proposition/type expression by making extensive analysis of its structure against the rules of the system. Feedback to the user is given via error messages, if an error is found.

Proved theorems can then be saved and used later in other contexts. Libraries containing whole theories can be built, compiled and deployed elsewhere. Some proof assistants offer a graphical user interface, with windows, menus and so on, in order to increase user productivity.

In certain situations, the process of formal proof development might involve some level of automation (when the assistant tries to prove at least part of the theorem without any help of the user), but this is not usual since the problem of automatic proof construction is not decidable in the general case and thus proof assistants have very limited capabilities in this respect. For this reason, the process is normally guided by the strategy and hints interactively provided by the user. On the other hand, in all cases it involves "automated proof checking", which is a much simpler process of checking that a proof is correctly constructed according to the inference rules of the underlying theory.

The term "assistant" stresses the fact that such tools are not unrealistic automated theorem provers, but instead assistants for a human that needs to state and prove a theory with a number of lemmas and theorems, checking all steps of the development and offering some high level actions that ease the construction of the proofs. This is done in the setting of an underlying mathematical theory, whose characteristics drive the syntax and semantics of the language used and also shapes the style of reasoning and computing that can be used to achieve the results. For this reason, one must first consider the nature, the roots, the limits and the capabilities of this theory before considering its use in a computational environment.

"Proof irrelevance" is an important concept when it comes to using a proof assistant to automate, as much as possible, the construction of proofs and terms. Since the importance of a proof relies simply on its existence, no matter how long or complex it is, the use of a proof assistant should place no burden when some automation is done in the process. This means, essentially, that there are no important differences between two different proofs of the same proposition. On the other hand, finding a term that complies with a specification might have serious impact on its usage as a computer program, since readability and efficiency are standard concerns. Two different terms might verify the same specification, but for practical reasons there might be important differences between them. For this reason, proof assistants are not generally considered when pursuing automated program development.

Proof assistants enable the user to formally prove theorems and/or to build programs. But what is a formal proof anyway? As noted by Thomas Hales in Hales (2008):

> “A formal proof is a proof in which every logical inference has been checked all the way back to the fundamental axioms of mathematics. All the intermediate logical steps are supplied, without exception. No appeal is made to intuition, even if the translation from intuition to logic is routine. Thus, a formal proof is less intuitive, and yet less susceptible to logical errors.”

Thus, a formal proof is not only a proof that is written in a precise and non-ambiguous mathematical language, but also a construction that can be verified, in every step, according to the fundamental axioms of mathematics. As this involves a lot of effort, not usually fully undertaken by the practicing mathematician in the so-called informal proofs, the computer and the proof assistant play an important role in making this a feasible task.

2.2 Benefits and Applications

The main benefits of using a proof assistant to develop mathematical theories and computer software are:

- A powerful and uniform language is provided along the whole process of representation and proof building;
- Proofs and programs can be mechanically and efficiently checked;
- Program verification can be reduced to a simple type checking algorithm that can be very fast and reliable;
- Reduced user time and effort necessary to check proofs and programs, specially long and complex ones, as these are completely carried out by the machine;
- Because of the above, the checking process is also more reliable and less error-prone;

- Equivalent proofs can be considered quantitatively, that is, against criteria such as readability, level of abstraction, size, structure etc;
- Results can be stored for future reuse and can be easily retrieved in different contexts;
- Different proof strategies can be easily tested, saved and recovered, and results can be easily combined (for example, a simple lemma can be used to prove another more complex lemma);
- A proof assistant can interact with the user in order to ease the construction of the proof or the program, sometimes doing all or part of the work;
- The user gets a deeper insight into the nature of the proof or program, enabling a better understanding of the problem and usually leading to simpler or more efficient solutions.

As pointed out by Georges Gonthier in Gonthier (2008a), an important benefit of developing a formal proof is not only to become sure of its correctness. Due to the level and quantity of details involved, it is also an important exercise in getting a better understanding of the nature of the proof, possibly leading to further simplification.

Current applications of proof assistants include the formalization of important mathematical theorems, but are not limited to these. Relevant and more pragmatic uses include also the automatic review of large and complex proofs in papers submitted to specialized journals, and the verification of hardware and software components in computer systems, as noted by John Harrison in Harrison (2008). In his paper, Harrison also includes a concise presentation of the foundations, history, development and perspectives of the use of interactive proof assistants. Other more ambitious applications include the development of correct software, specially those used in critical applications, and the formalization of whole theories.

Not everything are advantages and benefits, however. A general claim against the use of proof assistants is the low reliability of the results, caused by possibility of errors in the underlying infrastructure. This includes bugs or failures in the code of the proof assistant itself, in the processor of the high level language used to implement it, as well as in the related libraries, in the operating system and in the hardware on which it is executed. This, however, can be virtually eliminated as these infrastructure items are not specific to proof assistants. On the contrary, they are usually commercial and large-scale products that are extensively used and tested in wide variety of situations and over long periods of time, thus making it very unlikely to confirm the worries of the claim.

Another important aspect of formal proofs is that their size can be much larger than the size of the corresponding original informal proofs. This is due to the very detailed nature of these proofs, and different researchers consider different expansion factors. If, on one hand, there is a general agreement that the size of formal proofs is larger than the size of the corresponding informal proofs, on the other hand the value of this expansion factor is a polemic subject, since different informal proofs of the same theorem may vary considerably in size (considered by number of characters, lines or pages) and different authors use different methodologies. As an example, however, we mention the work of Freek Wiedjik in Wiedijk (2012), for whom this expansion factor seems to be the constant and equal to 4. He came to this conclusion after comparing the size of the formal and informal proofs of different important projects in different proof assistants. The factor became since then known as the "de Bruijn factor" in honor to Nicolaas Govert de Bruijn, who first observed this phenomenon in the early 1980s while working in the Automath project. Anyway, this number should not be taken as final or definitive, but only as an evidence that formal proofs are indeed much larger than the corresponding informal proofs.

Other aspects that keep interactive proof assistants from being more popular are (i) the reduced number of professionals and researchers already using them, which some relate to the resamblance of their languages to computer code, keeping for example pure mathematicians uninterested in the technology, and (ii) the slowly learning curve, which makes it very difficult for newcomers to produce realistic and useful results in short to mid-term periods.

Proof assistants are normally implemented in some functional language (like Lisp, Haskell, Ocaml and ML) and differ in some fundamental features, such as the support for higher-order logic, dependent types, code extraction, proof automation and proof by reflection. Most of them are developed and maintained by research institutes and universities in Europe and USA, and can be freely used and distributed.

Freek Wiedijk compiled in 2006 a list of what he calls "The Seventeen Provers of the World", with the objective of gathering information on and comparing the features of the main provers available by then (WIEDIJK, 2006). For this, he invited researchers with different formal proof backgrounds and expertise to provide a formal proof of $\sqrt{2}$ being irrational (Pythagoras' Theorem) in different proof assistants, in order to highlight their usage style, expressive power, limitations and capabilities. Also, the paper includes a profile of each of these assistants and brings much insight into their nature and objectives, as well as their differences and unique characteristics.

2.3 Automath

The history of the modern proof assistants starts in 1967 when Nicolaas Govert de Bruijn, a Dutch mathematician, designed and later implemented the Automath proof checker (VARIOUS, 1994; GEUVERS, 2009). His idea was less ambitious than the ones behind current proof assistants: he only wanted computers to be able to check the correctness of mathematical proofs, instead of having to do it by hand. As a consequence, the process would be more secure, less error-prone and much faster and less tedious. To achieve his objective, he had to design also a language in which statements and proofs could be expressed. The user would then write down his theorems and proofs in this language, and the system would check their correctness on the basis of the inference rules of the underlying calculus of the language. Because of this, Automath is a term that refers both to the language and the checker of de Bruijn. In his own words (BRUIJN, 1983) we find a simple and yet rich explanation of what Automath is and what are its purposes:

> "Automath is a language for expressing detailed mathematical thoughts. It is not a programming language, although it has several features in common with existing programming languages. It is defined by a grammar, and every text written according to its rules is claimed to correspond to correct mathematics. It can be used to express a large part of mathematics, and admits many ways of laying the foundations. The rules are such that a computer can be instructed to check whether texts written in the language are correct. These texts are not restricted to proofs of single theorems; they can contain entire mathematical theories, including the rules of inference used in such theories."

Among the many contributions of the Automath project to the area of mathematical formalization and the development of proof assistants, it is possible to mention (GEUVERS, 2009): (i) the treatment of proofs as first class objects and the idea that proof checking could be reduced to type checking, which can be done in simple and efficient ways; (ii) the idea that the system should be minimal (a logical framework with only abstractions, bindings, substitutions etc) in order to allow the user to add his own inference rules and extensions and thus tailor the system for his own purposes and (iii) the introduction of the powerful and expressive dependent types (GEUVERS; NEDERPELT, 2013).

The idea of having independent proof objects allowed proofs to be easily checked by simple programs, which should, in his perspective, be enough to convince users of their correctness. This led to what is known today as the *De Bruijn criterion*, thanks to Barendregt and Geuvers (BARENDREGT; GEUVERS, 2001): the proof checker should be independent of other programs and as simple as possible, so that anyone could build his own if he so wishes. This way it should not be difficult to achieve a high level confidence in the checker itself, otherwise a critical component of the whole process.

De Bruijn was not aware of the Curry-Howard Isomorphism, and still reinvented it in his own terms while working on Automath. The following statement is a testimony of that (GEUVERS; NEDERPELT, 2013):

> "An important thing I got from Heyting is the interpretation of a proof of an implication $A \to B$ as a kind of mapping of proofs of A to proofs of B. Later this became one of the motives to treat proof classes as types."

Automath influenced a large number of proof assistants that were designed and implemented in the last 40+ years. Their designers usually recognize the pionerism of de Bruijn in many aspects and still use his ideas, one way or another, despite the great advances verified in software and computer technology since then. A nice discussion of the origins, the main characteristcs and the influence of Automath in other proof assistants can be found in Geuvers and Nederpelt (2013).

2.4 QED Manifesto

Proof assistants may lead, in the future, to the fulfillment of the dream described in "The QED Manifesto" (ANONYMOUS, 1994). Written and published by an anonymous group of researchers dedicated to automated reasoning in 1994, the text claims for a single automated repository of all mathematical knowledge and techniques, where the contents is previously mechanically checked and then made freely available to researchers and students all over the world.

Although important from an historical perspective, the practical consequences of this initiative were very limited: only a mailing list and two conferences (1994 and 1995) were organized following the publication. The reasons for this were analyzed by Freek Wiedijk 13 years later in Wiedijk (2007), and among the most relevant he lists (i) the reduced number of people working with formal mathematics by then and (ii) the fact that, according to his opinion, "formal mathematics does not resemble mathematics at all", since "formal proofs look like computer program source code", unsuitable for use by pure mathematicians.

Surely much still has to be done, but the recent developments in the area of formal mathematics, along with a wider use of related tools and specially the increasingly large and complex projects that were undertaken since then, all point to the relevant role that the area will play both in mathematics and computer science over the next years. The QED Manifesto dream might not be too far in the future after all.

The discussion in this chapter introduced the main characteristics of a general proof assistant, and showed that this technology has developed a lot in the last 50 years. The QED Manifesto has not been fulfiled (yet), but the use of interactive proof assistants in both mathematical formalization and certified software development is already impressive. To prove this we will now review, in Chapter 3, a few projects that confirm this impression.

After that, we will consider related formalizations in Chapter 4 and our own formalization in Chapter 5 and Chapter 6. The necessary background for the latter consists of Appendix A, for the formal mathematics implemented by the Coq proof assistant, and Appendix B, for the Coq proof assistant itself.

3 A Sampler of Formally Checked Projects

The increasing interest in computer-based formal mathematics over the recent years, the great diversity, availability and robustness of the proof assistants available today, the power and expressiveness of their underlying formal theories and the rapid evolution of computer systems in the last decades, as discussed in Chapter 2, has led to a fast increase in the number of researchers and professionals that are benefiting from them, while they work in larger projects of higher complexity.

In the case of the Coq proof assistant, for example, these include a wide range of projects in programming language semantics formalization (INRIA, 2015b), mathematics formalization (INRIA, 2015a) and teaching in both mathematics and computer science curricula (INRIA, 2015c). Most projects are academic, but there are also important industrial applications, specially in computer technology.

Besides that, Freek Wiedijk keeps a list that shows the formalization status of 100 important mathematical theorems that includes, for example, Fermat's last theorem. The list is updated continuously and is based on another list with the same theorems, chosen for "the place the theorem holds in the literature, the quality of the proof, and the unexpectedness of the result" (WIEDIJK, 2015). In 2008, Wiedijk reported that 80% of the theorems had already been formalized in some proof assistant (WIEDIJK, 2008). He also expressed his belief that this percentage would reach 99% in the next 3 years (that is, in 2011). As of December of 2015, however, according to Wiedijk in Wiedijk (2015)), 91% of the theorems in the list had been formalized in at least one proof assistant, thus showing that formalization is taking place at a slower pace than he thought. The most used proof assistants in this list are, in decreasing order, HOL Light, Mizar, Isabelle and Coq (both in 2008 and in 2015). Some simple theorems, such as the irrationality of $\sqrt{2}$, have more than a dozen different proofs developed in different proof assistants.

In order to show the nature and importance of some recent applications of interactive proof assistants, we have selected six projects that will be briefly described in the next sections. This selection includes three more theoretical and three more application-oriented projects, of which five use the Coq proof assistant and one uses the Isabelle/HOL proof assistant. This should convince the reader of the rapidly increasing importance of doing formal mathematics and using interactive theorem provers in current and future practice.

3.1 Four Color Theorem

It took one hundred and twenty four years since the Four Color Theorem was stated until it was finally proved (1852-1976). It is an important theorem because (i) it has a simple statement:

> The regions of any simple planar map can be coloured with only four colours, in such a way that any two adjacent regions have different colours.

and (ii) it was the first famous mathematical theorem to be proved with the aid of computers. These proofs date from 1976 and 1995, when different mathematicians provided, respectively, an initial proof that was very large and complex, and a more concise and readable demonstration of it. Both relied on the use of computers to analyze hundreds of cases in the demonstration. Because of this new way of proving mathematical theorems, the 1976 proof was very much criticized for those that considered computers unsuited for this task for a number of reasons, ranging from “unnatural” to “unreliable”.

Nevertheless, the 1995 proof still relied on partial human intervention, and for this reason it can not be considered a fully computer-checked proof. This happened only ten years later, in 2005, when Georges Gonthier, then a principal researcher from Microsoft Research, with the collaboration of Benjamin Werner of INRIA (*Institut National de Recherche en Informatique et en Automatique*, the French research institution fully dedicated to computational sciences), used the Coq proof assistant to develop a proof script that was completely verified by a machine (MICROSOFT RESEARCH, 2006; GONTHIER, 2008b; GONTHIER, 2008a; GONTHIER, 2015).

Considered a milestone in the history of computer assisted proving, the importance of this demonstration can be summarized in Gonthier’s own words (GONTHIER, 2015):

> “Although this work is purportedly about using computer programming to help doing mathematics, we expect that most of its fallout will be in the reverse direction — using mathematics to help programming computers.”

In the same reference, he expresses the fundamental role that is reserved for mathematics as a software development tool, with the aid of computers, and exposes new and exciting possibilities and directions for formal mathematics and computer technology evolution.

With approximately 60,000 lines of code and 2,500 lemmas in the proof script, Gonthier points that (GONTHIER, 2015):

> “As with most formal developments of classical mathematical results, the most interesting aspect of our work is not the result we achieved, but how we achieved it.”

The effort in this project also led to a byproduct, an extension to the Coq scripting language in the form of a new set of tactics designed to ease and reduce the development effort by implementing a style of proof known as "small scale reflection". This set turned out to be a plugin for the Coq system, which is now available under the name SSReflect (WHITESIDE; ASPINALL; GROV, 2012).

An important example of the cooperation between Microsoft Research and INRIA (among others), this work was finalized in the same period when these organizations decided to share efforts and create a new initiative dedicated to research on projects of common interest. This culminated in an association between them that was formalized with the inauguration of the Microsoft Research-INRIA Joint Centre in January of 2007 (MICROSOFT RESEARCH-INRIA JOINT CENTRE, 2015b). Almost a year before it was inaugurated, three formal methods projects were launched and, since then, a wide selection of projects is supported by the institution, mainly in the areas of the application of mathematics to increase the security and reliability of software and computer systems, and also on the development of new software tools and applications for complex scientific data. Georges Gonthier is nowadays a researcher of the Joint Centre.

3.2 Odd Order Theorem

Right after finishing the proof of the Four Color Theorem, Georges Gonthier engaged in an even more ambitious project: the formal proof of the Odd Order Theorem, also known as the Feit-Thompson Theorem. Conjectured by Burnside in 1911, it was first proved by Walter Feit and John G. Thompson in 1963.

The work of Gonthier, of enormous importance, started in May of 2006 and ended in September of 2012, thus consuming more than six years of a team led by him with full time dedication. Besides Gonthier, a number of other people from Microsoft Research and INRIA in several different locations worked together in order to achieve the result that rapidly gained attention throughout the world (MICROSOFT RESEARCH-INRIA JOINT CENTRE, 2012b; MICROSOFT RESEARCH-INRIA JOINT CENTRE, 2012a).

The theorem has deep implications both in mathematics (in the classification of finite simple groups) and also in computer science (specially in cryptography). Although the theorem states shortly ("any finite group of odd order is solvable"), its formal proof is approximately 150,000 lines long and contains some 13,000 theorems (almost three times the size of the Four Color theorem formal script). These big numbers are not due to the formal approach used, since the original demonstration by Feit-Thompson itself was quite long, with 255 pages of text.

The announcement by Microsoft Research (MICROSOFT RESEARCH, 2012) brings interesting opinions by Gonthier, stressing the context, the importance and the consequences of his work:

"The work is about developing the use of computers as tools for doing mathematics

> for processing mathematical knowledge. Computers have been gaining in math, but mostly to solve ancillary problems such as type setting, or carrying out numerical or symbolic calculation, or enumerating various categories of common natural objects, like polyhedra of various shapes. They're not used to process the actual mathematical knowledge: the theories, the definitions, the theorems, and the proofs of those theorems. My work has been to try to break this barrier and make computers into effective tools. That is mostly about finding ways of expressing mathematical knowledge so that it can be processed by software."

In the same announcement Andrew Blake, director of Microsoft Research Cambridge, summarizes:

> "One may anticipate that this could affect profoundly both computer science and mathematics."

The whole development is presented in detail in Gonthier et al. (2013). As pointed out in the conclusion by Georges Gonthier,

> "The outcome is not only a proof of the Odd Order Theorem, but also, more importantly, a substantial library of mathematical components, and a tried and tested methodology that will support future formalization efforts."

This library was indeed an important outcome of the formalization (as SSReflect was for the Four Color Theorem), and includes in reality many different libraires on a great diversity of topics such as numbers, combinatorics, finite group theory, algebra and avanced group theory. Known as "Mathematical Components" (MICROSOFT RESEARCH-INRIA JOINT CENTRE, 2015a), it contributes largely to the growing interest in using the SSReflect extension in other projetcs as well.

3.3 Kepler Conjecture

The Kepler Conjecture is the oldest problem in discrete geometry. It is a problem about the density of the packing of equally sized spheres, and states that under certain conditions this density can not be greater than a certain amount. The Conjecture was part of the 18th problem in the list produced by David Hilbert in 1900, with the 23 most challenging open mathematical problems of that time.

It was stated in 1611 by the german mathematician Johannes Kepler, and many attempts to prove it occurred during the next centuries. The proof was completed only in 1998 by the american mathematician Thomas Hales, with the assistance of his graduate student Samuel Ferguson, but published only in 2006. The reason why it took so long to be published is that the work also relied on long and complex computer proofs, besides traditional mathematics, and this caused immense difficulties for the reviewers. As a consequence, it took many years to achieve a good degree of confidence in the results, and the work was finally published without ever having a complete certification. For this reason, Hales decided to formalize his work and in January 2003 he created and coordinated a team with the objective of producing a formal proof of the Conjecture, including both the informal and computer parts of the proof. The so-called Flyspeck project announced its conclusion in August 2014, and used the Isabelle and HOL/Light theorem provers. Before that, however, it evolved into a "large international collaboration" and directly involved 5 PhD students (HALES et al., 2015).

The page of the Flyspeck project on the Internet (HALES, 2015) contains links to the code of the formal proof and to the official announcement of the conclusion of the formalization, along with reports on the final and intermediate results. The whole formalization was based in the book published by Hales on his proof (HALES, 2012), and a description of the formalization can be found in Hales et al. (2015).

Without citing numbers, Hales states that Flyspeck has size and complexity similar to the Odd Order (Section 3.2), CompCert (Section 3.5) and seL4 (Section 3.6) projects (HALES et al., 2015). He also states that Flyspeck has probably set a new record in terms of lines of code in a verification project, and makes interesting considerations about the rewiewing process for both the informal and the formal proofs developed by him and his group.

3.4 Homotopy Type Theory and Univalent Foundations of Mathematics

The Univalent Foundations Program is an ambitious research program developed at the Institute for Advanced Study in Princeton by a group led by Vladimir Voevodsky. It aims at building new foundations for mathematics, as an alternative to classical set theory, and is based on a new interpretation of Martin Löf's constructive type theory into the traditional field of homotopy (AWODEY, 2012; PELAYO; WARREN, 2012).

Lots of efforts are being put into this promising project, which now has the collaboration of many researchers from different parts of the world. Large part of the development is done in a modified version of Coq (another proof assistant, Agda, is also used) and, contrary to usual practice, many theorems are first formally proved and then converted to an informal description (VOEVODSKY, 2013). Although considered a work in progress, four important sources compile the main results obtained so far: the official Homotopy Type Theory site (VOEVODSKY, 2015a), the GitHub repository where the development is done (VOEVODSKY, 2015b), the arXiv page (VOEVODSKY, 2014) and the collaborative book "Homotopy Type Theory: Univalent Foundations Program", freely available for download (VOEVODSKY, 2013).

3.5 Compiler Certification

Moving from strictly academic and theoretical projects into more applied and relevant to computer science ones, it is possible to find some important commercial initiatives, such as the CompCert project led by Xavier Leroy (LEROY, 2009; COMPCERT, 2015). CompCert is a shorthand for the words "Compiler" and "Certification", and refers to a compiler that was fully verified by a machine in order to assure its semantic preservation properties: in other words, that the low-level code produced as output has, in every case, the same semantics as the high-level code taken as input.

Developed to be used in the programming of critical software systems, for which exhaustive testing is a very difficult task, CompCert brings new perspectives into the construction of safety-critical software for embedded applications, while generating code that still meets the performance and size criteria required for this type of usage. To be certified, in this case, means that all internal phases of the compiler were proved to implement correct translations, including complex ones such as code optimizations. Thus, assuring that source and object code behave in exactly the same way can be extremely useful when, for example, formally proving that the source code satisfies all the required specifications, because these will also be satisfied by the executable code. Otherwise, a formally certified source program compiled by a non certified compiler could, due to programming errors in the implementation of the compiler, lead to code that did not comply with the specifications.

CompCert translates a large subset of the C programming language into the PowerPC, ia32 and ARM machine code, the choice of both being related to their popularity in real-time applications, specially avionics. CompCert was not only certified in Coq, it was mostly developed (programmed) in the proof assistant as well, thus making the whole project self-contained and easier to reason about. With the help of Coq's code extraction facility, functions and proofs could easily be transformed into functional programs, thus leading to an efficient implementation. The whole project consumed 3 person-years over a five years period, and comprises 42,000 lines of Coq code.

3.6 Microkernel Certification

Generating certified low level code is important, but running this code on a certified environment is of no less importance in order to assure that the expected behaviour is achieved.

Operating systems are critical components in every computer system and must be trusted in all situations, as they provide an execution environment — supposedly reliable — for application programs. Among its building blocks are microkernel programs that execute on the privileged mode of the hardware, meaning that detecting and recovering from software faults in this case is extremely difficult. This, of course, explains the interest in their mechanized certification, which in this case means making sure that the final implementation conforms exactly to the properties of the corresponding high level specifications.

The seL4 project was developed with this objective (KLEIN et al., 2010). Consisting of approximately 10,000 lines of C code, it was formally certified with the Isabelle/HOL proof assistant and, as a result, it claims to exhibit ideal characteristics for this kind of program: no crash and no execution of any unsafe operation (code injection, buffer overflow, general exceptions, memory leaks etc), under any circumstances. The cost of certification was not low, indeed: some 200,000 lines of Isabelle script had to be used or developed in order to complete the proof, and consumed around 11 person-years. The authors believe, however, that with the experience and lessons learned from this project, the size of new and similar projects could be reduced to 8 person-years, which is the double of the estimate for equivalent projects without any kind of certification. This means, in practice, that formal certification is indeed a real possibility, even for a commercial product with thousands of lines of code, considering the benefits that can be obtained. It is also interesting to note, in their work, that they considered not only the cost of a full verification, but also the incremental cost of repeating the verification when some feature is added or changed to the microkernel, which is a realistic attitude for projects that evolve over time.

3.7 Digital Security Certification

Even before the Four Colors Theorem was mechanically proved, an important project of digital security verification was being conducted with the support of the Coq proof assistant.

Java Card^{TM} is one of the most important smart card platforms being used. It allows users to carry with them, in a supposedly safe way, different applications that keep and share personal data, such as banking, credit card, health and many others. Applications run on a Java Card platform, and they can be preloaded or even loaded after the card has been issued to the user.

Although the convenience and relevance of this technology is unquestionable, it raises problems that must be adequately dealt with, specially related to the confidence and integrity of the data stored in the card. In a context of multiple applications distributed by different companies over different periods of time, “confidence” means that sensitive data can not be shared without authorization, and “integrity” means that data cannot be changed without permission. Achieving high levels of security in such an environment is not an easy task, but is certainly a must in order to enable its widespread and safe use, without exposing one’s personal data among other possible harmful consequences.

For this reason, it is quite natural that efforts have been directed at formalizing the behaviour and properties of the Java Card platform, including the virtual machine, the runtime environment, the API and the Java Card programs themselves, in interactive theorem provers such as Coq, where a full machine-checked certification could be pursued.

This happened, indeed, with the work of many and different researchers and private companies in the beginning of the 2000s. Barthe, Dufay, Jakubiec and Sousa at INRIA, for example, published a series of papers that address the problem from different perspectives: from the formalization of the Java Card Virtual Machine (BARTHE et al., 2000) and other components (BARTHE et al., 2001) to the construction of offensive (more efficient) virtual machines from defensive (less efficient) ones (BARTHE et al., 2002).

The industrial segment (banks, insurance and credit card companies) has invested a lot in this technology and also put direct efforts in the verification of its safety properties. An example of this is the paper by Andronick, Chetali and Ly from the Advanced Research Group on Smart Cards at Schlumberger Systems, which used Coq to prove applet isolation properties in the Java Card platform (ANDRONICK; CHETALI; LY, 2003). Applet isolation, in this case, means that neither the platform will become damaged, nor an application will interfere with another in an improper way. Another initiative was published in 2007, when Gemalto, an important developer and supplier of digital security technology for financial institutions, telecom operators and governments all over the world, announced that it had achieved, for the first time in history, the highest possible level of certification for its Java Card implementation, after extensive formalization with the Coq proof assistant (GEMALTO, 2007).

3.8 Summary

The size, complexity and importance of the projects reviewed in this chapter are evidences that mathematical formalization is a mature discipline, being increasingly adopted by researchers and professionals in both academic and industrial institutions.

In computer science applications, the interest in the use of interactive proof assistants derives basically from the fact that it allows the construction of certified software, that is, software that provably conforms to a certain specification. This is the case with compiler certification (Section 3.5), microkernel ceritification (Section 3.6) and digital security cerification (Section 3.7), as well as with most other applications in this domain.

On the more theoretical side, the objective is usually different: some projects aim in formalizing known results, like classical theorems or theories, while others try to develop whole new theories. In the first group we find the proof of the Four Color Theorem (Section 3.1), the Odd Order Theorem (Section 3.2) and the Kepler Conjecture (Section 3.3). Homotopy Type Theory (Section 3.4) is in the second group.

Still, the projects in the first group have completely different natures: the first had already been informally proved before the formalization was done, however with the assistance of computer programs to check the many different cases; the second also had an informal proof prior to the formalization, but no computers were used in it; the special character of the third comes from the fact that, despite some previous efforts by other people, that were never proved to be completely correct, the same researcher conducted the conclusion of the informal proof and, right after, its formalization.

Together, these projects set a clear trend towards the increased practice of mathematical formalization. This, in turn, feeds the continued development of interactive proof assistants and, more important, demands the education of students, professionals and researchers in order to master the theory, evolve the tools and expand the repository of formalized theories and certified software.

The present work follows the trend towards general mathematical formalization of classical theories. Although it is of special relevance to computer science, the extraction of cerfified programs has not been a concern. Nevertheless, it constitutes an important contribution about a theory that is fundamental to both mathematics and computer science.

The projects discussed in this chapter involved groups of researchers and professionals over long periods of time. This, of course, is not a common characteristic to all formalization projects. However, it is quite clear from the experience acquired with the present project, and confirmed by the projects that were reviewed, that the complexity of a formalization project grows very fast with direct impact on the quantity of resources that need to be allocated in order to fulfill the objectives.

4 Related Work

Language and automata theory has been subject of formalization since the mid-1980s, when Kreitz used the Nuprl proof assistant to prove results about deterministic finite automata and the pumping lemma for regular languages (KREITZ, 1986). Since then, the theory of regular languages has been the subject of intense formalization by various researchers using many different proof assistants, as explained in Section 4.1. The formalization of context-free language theory, on the other hand, is more recent and includes fewer accomplishments, mostly concentrated in certified parser generation. These are described in Section 4.2. The conclusions, including the motivation for this work, are presented in Section 4.3.

The search for formalization results in language and automata theory was conducted by the author from the second half of 2013 to the second half of 2015. Different strategies were used, as explained below:

- Search on the Internet using keywords such as *formalization*, *Coq*, *language theory*, *automata theory* and *context-free*;
- Personal feedback from researchers such as Yves Bertot (INRIA, France), Nelma Moreira (Porto University, Portugal), Jean-Christophe Filliâtre (CNRS, France) and Michael Norrish (NICTA, Australia);
- Feedback from the anonymous reviewers of the papers submitted by the author to ITP'14, LSFA'14, LFMTP'14, TYPES'14, LSFA'15, WoLLIC'15, LFCS'16 and LATA'16;
- Background material of published papers by different researchers on related subjects;
- Participation and interaction of the author in workshops and seminars in Marseille and Paris (in 2014) and Frejus (in 2015).

These are the sources of the references that are discussed in the next sections. Except for any missing or very recent work not covered here, this chapter presents a complete and up-to-date review of language theory formalization efforts over the last decades.

4.1 Regular Languages

Among the most representative formalizations of regular languages theory, we can mention the works by Constable (CONSTABLE et al., 1997), Kaloper (KALOPER; RUDNICKI, 1996), Filliâtre (FILLIÂTRE, 1997a), Briais (BRIAIS, 2008), Miyamoto (MIYAMOTO, 2014), Moreira (MOREIRA; PEREIRA; SOUSA, 2009; ALMEIDA; MOREIRA; REIS, 2010; ALMEIDA et al., 2011; MOREIRA; PEREIRA; SOUSA, 2012), Braibant (BRAIBANT; POUS, 2010; BRAIBANT; POUS, 2012), Asperti (ASPERTI, 2012), Coquand (COQUAND; SILES, 2011), Krauss (KRAUSS; NIPKOW, 2012) and Wu (WU; ZHANG; URBAN, 2011). The most recent and complete formalization, however, is the work by Doczkal, Kaiser and Smolka (DOCZKAL; KAISER; SMOLKA, 2013), which used Coq and the SSReflect extension to prove the main results of regular language theory. In the next paragraphs we discuss briefly each of these.

An important initiative is the work of Jean-Christophe Filliâtre and Judicäel Courant (FILLIÂTRE, 1997a; FILLIÂTRE, 1997b), whose libraries developed during the mid-1990s are still available as a user contribution under the label "Computer Science/Formal Languages Theory and Automata" in the Coq official page in the Internet (INRIA, 2015). It includes some results of regular languages theory and on context-free language theory as well.

As the authors explain, it "formalises the beginning of formal language theory" and the main results are restricted to proofs of (i) every regular language being accepted by a finite automata, including the extraction of a certified function that maps regular expressions into finite automata; (ii) a few results of context-free language theory (see Section 4.2). The whole development consists of 29 files with approximately 11,000 lines of Coq scripts. In Filliâtre (1997a), the authors describe the details of their formalization in Coq of Kleene's theorem (which expresses the equivalence of regular expressions and finite automata). Also, they mention in it their intention of formalizing other results of regular language theory in the future as well. However, as confirmed by one of the authors in a recent personal correspondence (2013), no further development has happened since then.

The paper by Almeida, Moreira, Pereira and Sousa (ALMEIDA et al., 2011) gives an account of this work and mentions the work of Sébastien Briais as being, somehow, a further development of it. As they explain:

> "Formalization of finite automata in Coq was first approached by J.-C. Filliâtre... The author's aim was to prove the pumping lemma for regular languages and the extraction of an OCaml program. More recently, S. Briais ... developed a new formalization of formal languages in Coq, which covers Filliâtre's work. This formalization includes Thompson construction of an automaton from a regular expression and a naïve construction of two automata equivalence based in testing if the difference of their languages is the empty language."

An extensive search, however, did not reveal any paper or publication documenting this work, or any consequences of it, except for a Briais' personal web page where the whole development, consisting of some 9,000 lines of Coq script distributed in 5 files, can be downloaded (BRIAIS, 2008).

Moreira and Sousa (MOREIRA; PEREIRA; SOUSA, 2009) have also published on the formalization of regular languages as a Kleene algebra, using the Coq proof assistant, and point out that their work is partially based on the work of Filliâtre (FILLIÂTRE, 1997a).

Also available as a user contribution in the Coq official page, the library of Takashi Miyamoto (MIYAMOTO, 2014) provides support for the use of regular expressions and expresses the most important results and properties related to it. Distributed in 8 files, the project is based on Janusz Brzozowski's work (BRZOZOWSKI, 1964), satisfies the axioms of Kleene algebra (associativity, commutativity, distributivity and identity elements) and comprises some 1,500 lines of Coq code. The work, however, does not map regular expressions to equivalent finite automata. As in the previous case, there is neither knowledge of any paper or report on the project, nor any account of the period when this development occurred.

In the end of the 1990s, Constable, Jackson, Naumov and Uribe, from Cornell University, worked in the area of automata theory formalization using the Nuprl proof assistant (CONSTABLE et al., 1997). They followed closely the structure of the first textbook by Hopcroft and Ullman (HOPCROFT; ULLMAN, 1969) in their development, and as a result they were able to formalize the concepts of symbol, sentence, language and finite automata. The main result presented in the paper is, however, restricted to the formalization of the Myhill-Nerode Theorem and its state minimization corollary. Yet, the authors consider their work to have an exploratory character, as they wanted to investigate ways of formalizing "computational mathematics" as opposed to the formalization of "classical mathematics", which is based on set theory. Essentially, they express that they "... want to show that constructive proofs can be used to synthesize programs" and "... want to examine whether constructive type theory is a natural expression of the basic ideas of computational mathematics in the same sense that set theory is for purely classical mathematics". The paper follows closely the definitions of the book, but points out the need for some modifications in order to "enable a constructive proof". Also, some gaps in it were corrected and filled in as a consequence of the detailed formulation and checking done by the proof assistant. As a result, not only the theorems are successfully formalized and proved, but also a state minimization algorithm is automatically obtained. Finally, the authors make considerations on the work needed to formalize the other chapters of the book, and express their belief that, with the experience acquired, the whole contents of it could be formalized by a four-person team in less than 18 months.

A similar initiative, from the mid-1990s, is the work of Kaloper and Rudnicki from the University of Alberta (KALOPER; RUDNICKI, 1996). They used the Mizar proof assistant to formalize finite automata and the minimization theorem. Also, they formalized Mealy and Moore finite transducers and proved the equivalence of these models. All the work is based on the textbook by Denning, Dennis and Qualitz (DENNING; DENNIS; QUALITZ, 1978) and is publicly available at Kaloper and Rudnicki (1994). As pointed out by the authors, some errors in the book were discovered during the formalization process.

The work by Christoph Kreitz on the formalization of deterministic finite automata using the Nuprl proof assistant dates from a decade earlier (1986) and is probably a pioneer in this area (KREITZ, 1986). Kreitz used his formalization to state and prove the pumping lemma for regular languages and also describe how it could be extended to represent non-deterministic finite automata and automata with empty transitions. Code extracted from the proof allows the user to compute the number n of the lemma and, given a string w, $|w| \geq n$, split it into substrings x, y and z, $w = xyz$, that satisfy the requirements of the lemma. Their work was based on the 1979 version of Hopcroft and Ullman's textbook (HOPCROFT; ULLMAN, 1979). Wu, Zhang and Urban worked on a version of the Myhill-Nerode theorem that uses regular expressions instead of finite automata (WU; ZHANG; URBAN, 2011). Berghofer and Reiter formalized a library for automata that served as support for a decision procedure for Presburger arithmetic (BERGHOFER; REITER, 2009).

Firsov and Uustalu have developed a certified parser generator for regular languages, using the Agda prgramming language, by mapping regular expressions into a representation of non-deterministic finite automata (FIRSOV; UUSTALU, 2013). They have also formalized proofs of correctness and completeness for this transformation.

A recent and important reference is the work of Christian Doczkal, Jan-Oliver Kaiser and Gert Smolka (DOCZKAL; KAISER; SMOLKA, 2013). Following the structure of the book by Kozen (KOZEN, 1997), they did a fairly complete formalization of regular languages theory. First of all, they managed to represent regular expressions, deterministic and non-deterministic finite automata, and then to prove the main results of regular language theory, such as the equivalence of these representations, the Myhill-Nerode theorem, the existence and uniqueness of a minimal finite state automata, plus closure properties and the pumping lemma. All the development was done in Coq, is only 1,400 lines long, and benefited from the use of the SSReflect Coq plug-in, which offers direct support for finite structures such as graphs and finite types.

Nelma Moreira and David Pereira, from Universidade do Porto, among other researchers and universities of Portugal, have done extensive formalization and published many papers on related subjects using the Coq proof assistant. These include topics such as the mechanization of Kleene algebra (MOREIRA; PEREIRA; SOUSA, 2009), verification of the equivalence of regular expressions (ALMEIDA; MOREIRA; REIS, 2010; MOREIRA; PEREIRA; SOUSA, 2012) and the construction of finite automata from regular expressions (ALMEIDA et al., 2011).

The decision of the equivalence of regular expressions, in particular, has deserved a lot of attention recently, and a variety of approaches have been used by different authors, besides Moreira and Pereira. Thomas Braibant and Damien Pous have developed a Coq tactic for establishing the equality of regular expressions and deciding Kleene algebras (BRAIBANT;

POUS, 2010; BRAIBANT; POUS, 2012). Andrea Asperti (ASPERTI, 2012) has developed an efficient algorithm for testing regular expression equivalence and also formalized in Matita the equivalence of regular expressions and deterministic finite automata. Thierry Coquand and Vincent Siles (COQUAND; SILES, 2011) and Alexander Krauss and Tobias Nipkow (KRAUSS; NIPKOW, 2012) have recent papers published in this area, respectively using Coq and Isabelle/HOL.

4.2 Context-Free Languages and Other Language Classes

Context-free language theory formalization is a relatively new area of research, when compared with the formalization of regular languages theory, with some results already obtained with the Coq, HOL4 and Agda proof assistants.

The pioneer work in context-free language theory formalization is probably the work by Filliâtre and Courant (INRIA, 2015), which led to incomplete results (along with some important results in regular language theory, see Section 4.1) and includes closure properties (e.g. union), the partial equivalence between pushdown automata and context-free grammars and parts of a certified parser generator. No paper or documentation on the part of their work dedicated to context-free language theory is available however.

Most of the extensive effort started in 2010 and has been devoted to the certification and validation of parser generators. Examples of this are the works of Koprowski and Binsztok using Coq (KOPROWSKI; BINSZTOK, 2010), Ridge using HOL4 (RIDGE, 2011), Jourdan, Pottier and Leroy using Coq (JOURDAN; POTTIER; LEROY, 2012) and, more recently, Firsov and Uustalu using Agda (FIRSOV; UUSTALU, 2014). These works assure that the recognizer fully matches the language generated by the corresponding context-free grammar, and are important contributions in the construction of certified compilers.

On the more theoretical side, on which the present work should be considered, Norrish and Barthwal published on general context-free language theory formalization using the HOL4 proof assistant (BARTHWAL; NORRISH, 2010a; BARTHWAL; NORRISH, 2010b; BARTHWAL; NORRISH, 2014), including the existence of normal forms for grammars, pushdown automata and closure properties. These results are from the PhD thesis of Barthwal (BARTHWAL, 2010), which includes a proof of the Pumping Lemma for context-free languages. This particular result, however, has not been published in any journal. Recently, Firsov and Uustalu proved the existence of a Chomsky Normal Form grammar for every general context-free grammar, using the Agda proof assistant (FIRSOV; UUSTALU, 2015).

A special case of the Pumping Lemma for context-free languages, namely the Pumping Lemma for regular languages, is included in the comprehensive work of Doczkal, Kaiser and Smolka on the formalization of regular languages (DOCZKAL; KAISER; SMOLKA, 2013) using SSReflect.

When it comes to computability theory and Turing machines related classes of languages, Asperti and Ricciotti (ASPERTI; RICCIOTTI, 2012) have used the Matita theorem prover to prove basic results on Turing machines, including the existence of a certified universal Turing machine. Xu, Zhang and Urban (XU; ZHANG; URBAN, 2013), more recently, have developed a formal model of a Turing Machine and related it to abacus machines and recursive functions. Michael Norrish (NORRISH, 2011) has done research on mechanization of some computability theory using λ-calculus and recursive functions using HOL4

4.3 Conclusions

The formalization of regular languages theory is quite extensive, covers many different aspects and has virtually included all important objects and results of the theory.

When it comes to context-free language theory, however, except for the works of Norrish and Barthwal, the efforts are in much smaller number and mostly related to certified parser generation. Besides that, it can be noted that so far apparently no formalization has been done in Coq for results not related directly to parsing and parser verification (except in HOL4 and Agda).

This constitutes an important motivation for the present work, since not many theoretical results of context-free languages theory have been formalized in Coq yet. This definitely constitutes an important opportunity, due to the increasing usage and importance of Coq in different areas and communities.

Specifically in the theoretical side, the formalization done by Norrish and Barthwal in HOL4 is quite comprehensive and includes the Greibach Normal Form and pushdown automata and its relation to context-free grammars. The formalization by Firsov and Uustalu in Agda comprises basically the existence of a Chomsky Normal Form, and formalizes the elimination of empty and unit rules, but not the elimination of useless and inaccessible symbols. Table 4.1 presents an overview of the main results obtained in each of these formalizations. Both are reviewed in more detail and also compared to our own formalization in Section 6.11.5.

Table 4.1: Context-free language theory formalizations of Norrish & Barthwal and Firsov & Uustalu

	Norrish & Barthwal	**Firsov & Uustalu**
Proof assistant	`HOL4`	`Agda`
Closure	✓	×
Simplification	✓	*only empty and unit rules*
CNF	✓	✓
GNF	✓	×
PDA	✓	×
PL	✓	×

The formalization of language and automata theory is relatively recent, fragmented and largely concentrated on regular language theory, with a few important results in context-free language theory as well. In the works discussed in this chapter, focus has been given to results associated to specific devices (finite automata, regular expressions, context-free grammars, Turing Machines etc) in an unrelated way in most of the cases. The early and few initiatives that were aimed at formalizing complete theories (e.g. Filliâtre's and Constable's) apparently did not develop further but recent works point again in this direction. This is the case of the works of Doczkal, Kaiser and Smolka for regular languages, and of Norrish and Barthwal for context-free languages, which are rather complete formalizations of the respective theories in Coq/SSReflect and HOL4. However, not much has been done so far in order to build a complete framework for language and automata theory, including more than a single language class, in Coq or any other proof assistant.

Table 4.1 includes the main formalizations of context-free language theory published up to date, and shows the absence of results in the Coq proof assistant. Considering the importance of Coq itself, it is clear that a formalization that pursues this objective is welcome, and this was the motivation for the present work.

The result of this work is, therefore, the formalization of a substantial part of context-free language theory in the Coq proof assistant. As it will become clear in Chapter 6, the present formalization is the second most comprehensive up to date (after Norrish and Barthwal's), however the most comprehensive in the Coq proof assistant, thus using a type theory. It includes the second ever formalization of the Pumping Lemma (the first is the one by Barthwal) and the first ever done in the Coq proof assistant. The resulting formalization is discussed in Chapter 5 and Chapter 6.

5 Context-Free Language Theory

Language and automata theory is a long standing and well developed area of knowledge that uses mathematical objects and notations to represent devices that constitute the basis of computation and computer technology, both in software and hardware. It has its roots in the then independent automata theory and language theory, which became a single discipline in the 1950s, as a consequence of the research by Noam Chomsky and others. The classical theory is consolidated since the 1970s, and after that only specific research topics continue to evolve independently.

Except for a few initiatives (specially related to regular languages) that have pursued a formal, mechanically checked approach to language and automata theory (see Chapter 4), most of the published literature in books and articles remain informal, in the sense that no single uniform notation has been used (although a certain standard notation established in the 1970s has been used since then with variations), no mechanized verification of the proofs has been obtained and no certified algorithms have been constructed. This contrasts with the large interest in language and automata theory, which ranges from academy to industry, from students to researchers and practitioners. Thus, it seems natural that an effort of formalizing this theory should be beneficial for a wide public and the countless different applications.

This chapter starts with a discussion of the importance and applications of the context-free language theory in Section 5.1. Then, it presents a characterization of context-free grammars and languages in Section 5.2, as well as references to the classical literature of the area. This last section also defines the scope of the present work, namely the properties and results of context-free language theory that were considered for formalization in the Coq proof assistant (in Chapter 6).

5.1 Importance and Applications

Language and automata theory is the basis of language definition, analysis, and implementation in general, and has a direct impact in areas such as programming language specification and compiler construction. Other applications are in the area of data description, digital circuits, control systems and communication protocols design and natural language processing. Also, it is the basis of the study of fundamental areas of computer science, such as problem decidability and algorithm complexity.

Context-free languages are highly important in computer language processing technology as well as in formal language theory. The theory of context-free languages is very rich and was developed from mid 1950s to late 1970s. Thanks to it, the definition, analysis and use of high-level programming languages, for example, has grown enormously over these years and since then, allowing for a new era of software consumption and development whose main charateristics are higher reliability, readability and maintanability and lower development costs. This has been made possible thanks to the use of context-free grammars in the construction of more precise language definitions and the automatic mapping of grammars into efficient and reliable syntactical analyzers.

Context-free grammars are extensively used in the definition of programming languages, natural language applications (for example, grammar correctors), machine protocols and many others. Initially conceived as models of natural languages, they turned out to find various applications in computer science. Examples of this is the development of tools such as *YACC* (which automatically transforms input context-free grammars into parsers written in a commercial programming language) in general compiler construction and modern languages such as *XML* in electronic commerce applications.

Pushdown automata are models of language implementation in language processors that appear in growing number. Context-free language theory brought into science the definition of artificial languages and contributed directly to the development of efficient implementation techniques, allowing for the widespread use of compilers, interpreters and translators in general.

The importance of context-free language formalization is multiple. First, it represents a guarantee that the fundamental results are correct, despite informal proofs of them being widely accepted as correct for a long time now. Second, it allows for a deeper insight into the nature of the proofs, and its implications. Third, the formalization may eventually lead to the certified implementation of some important algorithms, which can be vital in commercial and specially critical applications. Fourth, it can be the basis for an alternative approach towards teaching formal languages, using a constructive mathematical basis (see Section A.5) and tools that are becoming part of the software engineering regular activity, such as interactive proof assistants (see Chapter 2).

5.2 Characterization

Language and automata theory comprises, initially, the representation and classification of phrase-structured languages, and then the study of its properties. In the Chomsky Hierarchy (CHOMSKY, 1956), the language classes considered are the regular, context-free, context-sensitive and recursively enumerable languages. For each of these classes, it is possible to study its representation alternatives (e.g. regular sets, regular expressions, linear grammars and finite automata for regular languages) and the equivalences between them. Also, it is possible to formulate and prove some properties, such as the existence of minimal finite automata, the decidability of some questions (such as whether a string is generated by a linear grammar) and the validity of some closure properties (e.g. the union of the regular languages is also a regular language). Other language classes not originally part of the Hierarchy, such as the recursive and the deterministic context-free languages, were also identified and can be the subject of the same type of study.

Context-free (or type 2 in the Chomsky Hierarchy) languages is the class (set) of languages that can be defined by context-free grammars or pushdown automata. Indeed, it has been proved that these two mechanisms are equivalent, in the sense that they represent the same language class - the class of the context-free languages. This theory is the subject of many classical books, including Hopcroft and Ullman (1979), Lewis and Papadimitriou (1998), Sipser (2005), Sudkamp (2006).

A *context-free grammar* — originally defined by Chomsky in Chomsky (1956) — is a four-tuple $G = (V, \Sigma, P, S)$, where V is the vocabulary of G (it includes all non-terminal and terminal symbols), Σ is the set of terminal symbols (used in the construction of the sentences of the language generated by the grammar), $N = V \setminus \Sigma$ is the set of non-terminal symbols (representing different sentence abstractions), P is the set of rules and $S \in N$ is the start symbol (also called initial or root symbol). Rules have the form $\alpha \rightarrow \beta$, with $\alpha \in N$ and $\beta \in V^*$. Sets N, Σ and P must be finite.

A *word* is a sequence of symbols from an alphabet.

A *sentential form* is word in V^* (the reflexive and transitive closure of V), thus a string of non-terminal and terminal symbols of a grammar. A *derivation* is the process of substituting the left-hand side of a rule in a sentential form for the corresponding right-hand side of the same rule. We say that the first sentential form *derives* the second. The process continues until a sentential form composed only of terminal symbols is obtained (if this is possible). In this case, the sentential form is called a *sentence* of the language generated by the grammar. A *direct derivation* is a derivation in which a single rule has been used.

For example, if $\alpha \rightarrow \beta$ is a rule of grammar G, and $\mu\alpha\gamma$ is a sentential form of the same grammar, then the process of applying the rule in this case leads to the new sentential form $\mu\beta\gamma$, which is represented as $\mu\alpha\gamma \Rightarrow_G \mu\beta\gamma$.

A *language* is a set of words over Σ.

A *context-free language* is a language for which exists at least one context-free grammar that generates it. In other words, if G is a context-free grammar, then the language generated by G is represented by $L(G)$, and $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$ ($\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$).

The class of the context-free languages properly includes the class of regular (or type 3) languages. However, it is possible to prove the existence of non-context-free languages, and also that the class of context-free languages is properly included in another language class, the one constituted by the context-sensitive languages (except for the languages that include the empty string).

The context-free language class is known for having a popular and very useful property: it is possible to define languages that exhibit a strict balance of terms (strings) to the left and right of some other term. The word "balance", in this case, is used to mean that the number of terms on one side can be somehow related to the number of terms on the other side. This property is useful, for example, in the definition of programming languages that need to balance the use of parentheses, brackets, braces and similar syntatic elements. It is derived from the so-called "self-embedding property", which can be observed in context-free grammars that allow for derivations of the form $X \Rightarrow^* \alpha X \beta$, with $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$.

The theory of context-free languages comprises a small number of definitions (e. g. context-free grammars, context-free languages, derivations, derivation trees, pushdown automata), which in turn lead to a large number of important lemmas and theorems. The main definitions and results of this theory can be organized in the following groups, and is extensively discussed in books such as the ones cited before.

- Language representation notations (grammars, automata, BNF etc);
- Notations equivalence (e.g. grammars and automata);
- Grammar simplification;
- Normal forms (Chomsky and Greibach);
- Derivation trees, parsing and ambiguity;
- Determinism and non-determinism;
- Closure properties;
- Decidable and undecidable problems;
- Relation with other language classes.

The formalization of the whole context-free language theory is a work that exceeds by far the resources available for the present work. Thus, it was necessary to define a scope and decide which notations and results would be the object of this formalization. This has led to the selection of the following topics, all of which are related to context-free grammars and languages. No pushdown automata has been considered in the present formalization.

- Closure properties:
  - Union;

    - Concatenation;
    - Kleene star.
- Grammar simplification:
    - Elimination of empty rules;
    - Elimination of unit;
    - Elimination of useless symbols;
    - Elimination of inaccessible symbols.
- Chomsky Normal Form;
- Pumping Lemma.

All the efforts in this work were directed to the formalization of the above items (as described in Chapter 6), which (except for the closure properties) constitute a coherent, mutually dependent and important core of the context-free language theory. The result is a framework in which the remaining notations and results can be pursued with good support.

6 Formalization of Context-Free Language Theory

In this chapter we present the main results of our work, which is a comprehensive formalization of a substantial part of the context-free language theory in the Coq proof assistant. The concepts and results that have been formalized derive directly from the presentation and discussion of Chapter 5. Previous knowledge of the background material of Appendix A and Appendix B is required.

The present formalization can be organized into the following groups:

- Closure properties of context-free languages and grammars;
- Context-free grammar simplification;
- Chomsky Normal Form (CNF);
- Pumping Lemma for context-free languages.

The proper description of these results starts with an overview of the formalization strategy in Section 6.1.

The definition, in the Coq proof assistant, of the fundamental objects and notions of context-free language theory, such as context-free grammars, derivations and context-free languages, is then discussed in Section 6.2.

Once in possession of these definitions, the library that contains fundamental results on context-free grammars, and supports the whole formalization, is briefly presented in Section 6.3. It includes a number of different and essential lemmas on context-free language theory, most of which are unrelated to the others. This library is extensively used in all four groups of results described above.

Now that we have the basic definitions and the fundamental lemmas of context-free language theory, it is possibile to discuss each of the groups of results considered in this work. Before that, however, we describe the common method adopted in the first three groups in Section 6.4.

With the basic definitions, the supporting libraries and the method discussed in these sections, we are able start the description of the main results included in the first three groups of this formalization.

For the first group – closure properties in Section 6.5 –, we show that context-free languages are closed under the union, concatenation and Kleene star operations.

For the second group — grammar simplification in Section 6.6 –, we show, as a final result, that every context-free language can be generated by a context-free grammar that has no useless symbols, no inaccessible symbols, no unit rules, no empty rules and, further more, no rule with the start symbol in the right-hand side.

For the third group – Chomsky Normal Form in Section 6.7 –, we show that every context-free language can be generated by a context-free grammar that satisfies this form.

The fourth group of results does not follow the method described in Section 6.4, since it is not based on the idea of defining a new grammar from a previous one. It does, however, require important results on binary trees and their relation to CNF grammars, which are included in a single library whose contents are discussed in Section 6.8.

For the fourth group – Pumping Lemma in Section 6.9 –, we prove this important property that is valid for all context-free languages and can be used to show the existence of non-context-free languages.

All four groups include many intermediate results before the final lemmas and theorems are presented and proved. Besides presenting the main lemmas and theorems of each group, the following sections of this chapter also try to give an overview of the proof strategies that were used in each case. While the main results of the formalization are only 12, more than 500 lemmas had to be proved to achieve this objective. Thus, each of these main results is the root of a lemma dependency tree, which in most cases has a considerable depth. For this reason, the explanations provided in the following sections do not describe all details of all the lemmas involved, but still try to present the key issues involved in the proof of each main lemma or theorem.

In Section 6.10 we present some numbers of the formalization (number of lemmas, number of lines etc), together with a complete list of files, the corresponding main lemmas and the instructions on how to compile them. An expanded list of the main lemmas and theorems of the main libraries developed, along with brief descriptions, is included in Appendix C as a companion to the present chapter.

Finally, in Section 6.11 we discuss the process, phases and some design choices of the present formalization. We also include a few lessons that were learned during the formalization process and advices that could prove useful for other users.

All the source files of the formalization, including the examples of Section 6.2.1, Section 6.2.2, Section 6.3 and Section 6.5.1, are available for download from the repository (more information in Section 6.10).

6.1 Overview

To achieve the goals of this formalization, as with any other formalization, the following phases have been accomplished:

1. Selection of an underlying formal logic to express the theory and then a tool that supports it adequately;
2. Representation of the objects of the universe of discourse in this logic;
3. Implementation of a set of basic transformations and mappings over these objects;
4. Statement of the lemmas and theorems that describe the properties and the behaviour of these objects, and establish a consistent and complete theory;
5. Formal derivation of proofs of these lemmas and theorems, leading to proof objects that can confirm their validity.

For phase 1, the Calculus of Inductive Constructions and the Coq proof assistant have been selected. Besides Coq having good support and a large community of users, with many important projects of high complexity already developed in it (see Chapter 3), the calculus is very powerful and the language is very flexible (see Appendix B), as mentioned in most papers reporting formalizations in Coq.

Regarding phase 2, the following fundamental objects and notions have been defined:

- Symbols (including terminal and non-terminal);
- Sentential forms (strings of terminal and non-terminal symbols);
- Sentences (strings of terminal symbols);
- Context-free grammars;
- Derivations;
- Binary trees.

Then, the new definitions used in the formalization of closure properties, grammar simplification and Chomsky Normal Form describe, among others, the transformations that are required in order to represent the desired results (phase 3).

For phases 4 and 5, the achieved results can be placed in the four groups described before (closure properties, grammar simplification, Chomsky Normal Form and Pumping Lemma).

The dependencies among the main results of context-free language theory formalized in these groups can be summarized as follows:

1. Closure properties;
2. Grammar simplification $\rightarrow$ Chomsky Normal Form $\rightarrow$ Pumping Lemma.

In other words: the formalization of closure properties does not depend on previous results nor affects other results. On the other hand, the formalization of grammar simplification is needed in order to formalize the existence of a Chomsky Normal Form, and this in turn is required in order to formalize the Pumping Lemma for context-free languages. This dependency is not specific to this formalization, and is a consequence of the usual strategies of the informal proofs normally adopted in the classical literature of the area — for example, Sudkamp (2006).

To support the formalization of these results of context-free language theory, the following more fundamental and generic results had to be formalized first:

- Basic lemmas on arithmetic, lists and logic;
- Basic lemmas on context-free languages and grammars;
- Basic lemmas on binary trees and their relation to CNF grammars;

They can be interpreted as general libraries (specially the arithmetic and lists libraries) that can serve to different purposes, and in this case were used throughout the formalization. Thus, their description is more or less independent of the rest of the formalization. The other libraries form the core of the formalization and comprise the results obtained for the four groups listed in the beginning of the chapter.

6.2 Basic Definitions

In this section we present how the main concepts and objects of context-free language theory were defined in our formalization in Coq. They are used throughout the work. We discuss the representation of context-free grammars in Section 6.2.1, of derivations in Section 6.2.2 and of context-free languages in Section 6.2.3[1].

6.2.1 Grammars

Context-free grammars were represented in Coq very closely to the usual algebraic definition. The basic definitions in Coq are presented below. Let $G = (V, \Sigma, P, S)$ be a context-free grammar.

The sets $N = V \backslash \Sigma$ and Σ are represented as types (usually denoted by names such as, for example, `non_terminal` and `terminal`), separately from G. The idea is that these sets are represented by inductive type definitions whose constructors are its inhabitants. Thus, the

[1]The results of this section are available in libraries `cfg.v` and `cfl.v`.

number of constructors in an inductive type corresponds exactly to the number of (non-terminal or terminal) symbols in a grammar.

As an example, consider the following types representing, respectively, $\Sigma = \{a, b, c\}$ and $N = \{X, Y, Z\}$:

```
Inductive terminal: Type:=
| a
| b
| c.

Inductive non_terminal: Type:=
| X
| Y
| Z.
```

Once these types have been defined, we can create abbreviations for sentential forms (`sf`), sentences (`sentence`) and lists of non-terminals (`nlist`). The first corresponds to the list of the disjoint union of the types `non-terminal` and `terminal`, while the other two correspond to simple lists of, respectively, `non-terminal` and `terminal` symbols:

```
Notation sf := (list (non_terminal + terminal)).
Notation sentence := (list terminal).
Notation nlist:= (list non_terminal).
```

In the following example, $aXYb, aabc$ and ZZX are, respectively, a sentential form (`sf`), a sentence (`sentence`) and a list of non-terminals (`nlist`). The empty string ε is represented as the empty list `[]` in all cases:

```
Definition sf1: sf:= [inr a; inl X; inl Y; inr b].
Definition sf2: sf:= [].
Definition sentence1: sentence:= [a; a; b; c].
Definition sentence2: sentence:= [].
Definition nlist1: nlist:= [Z; Z; X].
Definition nlist2: nlist:= [].
```

The record representation `cfg` has been used for G. The definition states that `cfg` is a new type and contains three components. The first component is the `start_symbol` of the grammar (a non-terminal symbol) and the second is `rules`, that represents the rules of the grammar. Rules are propositions (represented in Coq by `Prop`) that take as arguments a non-terminal symbol and a (possibly empty) list of non-terminal and terminal symbols (corresponding, respectively, to the left and right-hand side of a rule). Grammars are parametrized by the types `non_terminal` and `terminal`.

```
Record cfg (non_terminal terminal : Type): Type:= {
start_symbol: non_terminal;
rules: non_terminal → sf → Prop;
rules_finite:
   ∃ n: nat,
   ∃ ntl: nlist,
   ∃ tl: tlist,
   rules_finite_def start_symbol rules n ntl tl }.
```

The predicate `rules_finite_def` assures that the set of rules of the grammar is finite by proving that the length of right-hand side of every rule is equal or less than a given value, and also that both left and right-hand side of the rules are built from finite sets of, respectively, non-terminal and terminal symbols (represented here by lists). This represents an overhead in the definition of a grammar, but it is necessary in order to allow for the definition of `non_terminal` and `terminal` as generic types in Coq.

```
Definition rules_finite_def
   (non_terminal terminal : Type)
   (ss: non_terminal)
   (rules: non_terminal → sf → Prop)
   (n: nat)
   (ntl: list non_terminal)
   (tl: list terminal) :=
In ss ntl ∧
(∀ left: non_terminal,
 ∀ right: list (non_terminal + terminal),
 rules left right →
 length right ≤ n ∧
 In left ntl ∧
 (∀ s : non_terminal, In (inl s) right → In s ntl) ∧
 (∀ s : terminal, In (inr s) right → In s tl)).
```

Since generic types might have an infinite number of elements, one must make sure that this is not the case when defining the `non_terminal` and `terminal` sets. Also, even if these types contain a finite number of inhabitants (constructors), it is also necessary to prove that the set of rules is finite. All of these is captured by predicate `rules_finite_def`. Thus, for every `cfg` defined directly of constructed from previous grammars, it will be necessary to prove that the predicate `rules_finite_def` holds.

The example below represents the grammar

$$G = (\{S', A, B, a, b\}, \{a, b\}, \{S' \rightarrow aS', S' \rightarrow b\}, S')$$

that generates language a^*b:

```
Inductive nt: Type:= | S' | A | B.
Inductive t: Type:= | a | b.
Inductive rs: nt → list (nt + t) → Prop:=
  r1: rs S' [inr a; inl S']
| r2: rs S' [inr b].

Notation sf:= (list (nt + t)).
Notation nlist:= (list nt).
Notation tlist:= (list t).

Lemma rs_finite:
∃ n: nat,
∃ ntl: nlist,
∃ tl: tlist,
In S' ntl ∧
∀ left: nt,
∀ right: sf,
rs left right →
(length right ≤ n) ∧
(In left ntl) ∧
(∀ s: nt, In (inl s) right → In s ntl) ∧
(∀ s: t, In (inr s) right → In s tl).
Proof.
∃ 2, [S'], [a; b].
split.
- simpl; left; reflexivity.
- intros left right H.
  inversion H.
  + simpl.
    split.
    * omega.
    * {
      split.
      - left; reflexivity.
      - split.
        + intros s H2.
          destruct H2 as [H2 | H2].
          * inversion H2.
          * {
            destruct H2 as [H2 | H2].
            - left; inversion H2. reflexivity.
            - contradiction.
            }
        + intros s H2.
          destruct H2 as [H2 | H2].
          * inversion H2. left; reflexivity.
          * {
            destruct H2 as [H2 | H2].
            - inversion H2.
            - contradiction.
            }
      }
  + simpl. split.
    * omega.
    * {
      split.
      - left; reflexivity.
      - split.
        + intros s H2. destruct H2 as [H2 | H2].
          * inversion H2.
          * contradiction.
        + intros s H2.
          destruct H2 as [H2 | H2].
          * inversion H2. right; left; reflexivity.
          * contradiction.
      }
Qed.

Definition g: cfg nt t:= {|
start_symbol:= S';
rules:= rs;
rules_finite:= rs_finite |}.
```

In this script, `nt` represents the type of the non-terminal symbols (with three inhabitants), `t` represents the type of the terminal symbols (with two inhabitants), `rs` is the set of rules (named `r1` and `r2`), `rs_finite` is the proof that `nt`, `t` and `rs` are finite, and finally `g` is the grammar definition that corresponds to G.

6.2.2 Derivations

Another fundamental concept used in this formalization is the idea of *derivation*: a grammar `g` *derives* a string `s2` from a string `s1` if there exists a series of rules in `g` that, when applied to `s1`, eventually results in `s2`. A direct derivation (i.e. the application of a single rule) is represented by $s_1 \Rightarrow s_2$, and the reflexive and transitive closure of this relation (i.e. the application of zero or more rules) is represented by $s_1 \Rightarrow^* s_2$. An inductive predicate definition of this concept in Coq (`derives`) uses two constructors:

```
Inductive derives
  (non_terminal terminal : Type)
  (g : cfg non_terminal terminal)
  : sf → sf → Prop :=
  | derives_refl :
     ∀ s : sf,
     derives g s s
  | derives_step :
     ∀ (s1 s2 s3 : sf)
     ∀ (left : non_terminal)
     ∀ (right : sf),
     derives g s1 (s2 ++inl left :: s3) →
     rules g left right → derives g s1 (s2 ++right ++s3)
```

The constructors of this definition (`derives_refl` and `derives_step`) are the axioms of our theory. Constructor `derives_refl` asserts that every sentential form `s` can be derived from `s` itself. Constructor `derives_step` states that if a sentential form that contains the left-hand side of a rule is derived by a grammar, then the grammar derives the sentential form with the left-hand side replaced by the right-hand side of the same rule. This case corresponds to the application of a rule in a direct derivation step.

In order to simplify some proofs, alternative (yet equivalent) definitions of the notion of a derivation (besides `derives`) are also used in the formalization (see Section 6.3).

A grammar `generates` a string if this string can be derived from its start symbol. Finally, a grammar `produces` a sentence if it can be derived from its start symbol.

```
Definition generates (g: cfg) (s: sf): Prop:=
derives g [inl (start_symbol g)] s.

Definition produces (g: cfg) (s: sentence): Prop:=
generates g (map terminal_lift s).
```

The relationship between `derives`, `generates` and `produces` is presented below:

- Predicate `generates`: a derivation that begins with the start symbol of the grammar;
- Predicate `produces`: a derivation that begins with the start symbol of the grammar and ends with a sentence.

$$\underbrace{\underbrace{S \Rightarrow \alpha_1 \Rightarrow \overbrace{\alpha_2 \Rightarrow \ldots \Rightarrow \alpha_{n-1}}^{\texttt{derives}}}_{\texttt{generates}} \Rightarrow \alpha_n \Rightarrow \omega}_{\texttt{produces}}$$

Thus, the following are valid propositions for grammar `g` defined in Section 6.2.1:

- `derives g [inr a; inl S'] [inr a; inr b];`
- `generates g [inr a; inl S']` and
- `produces g [a; b].`

It is now possibile to prove that a certain string is derived (or generated or produced) by a context-free grammar. As an example, the lemma below states that G (from Section 6.2.1) derives the string *aab* (that is, that $aab \in L(G)$), considering the derivation sequence $S \Rightarrow_G aS \Rightarrow_G aaS \Rightarrow_G aab$:

```
Lemma derives_g_aab:
derives g [inl S'] [inr a; inr a; inr b].
Proof.
apply derives_step with (s2:=[inr a; inr a])(left:=S')(right:=[inr b]).
apply derives_step with (s2:=[inr a])(left:=S')(right:=[inr a;inl S']).
apply derives_start with (left:=S')(right:=[inr a;inl S']).
apply r1.
apply r1.
apply r2.
Qed.
```

The proof of this lemma relates directly to the derivations in $S \Rightarrow aS \Rightarrow aaS \Rightarrow aab$, however in reverse order because the construction of the proof starts from the goal.

In some occasions, it is easier to build proofs based on the notion of a sequence of derivations, instead of using the `derives` predicate directly. For this reason, we introduce the new definition `sflist` below:

```
Inductive sflist (g: cfg): list sf → Prop:=
| sflist_empty: sflist g []
| sflist_start: ∀ s: sf,
                sflist g [s]
| sflist_step: ∀ l: list sf,
               ∀ s2 s3: sf,
               ∀ left: non_terminal,
               ∀ right: sf,
               sflist g l → last l [] = (s2 ++inl left :: s3) →
               rules g left right →
               sflist g (l++[s2 ++right ++s3]).
```

An `sflist` is a list of sentential forms, such that the next follows the previous one if there is a rule in the grammar that can be used to transform one in the other. Thus, it represents a sequence of direct derivations (or a single sentential form, or the empty list).

As an example, suppose `derives g s1 s5` is true (that is, $s_1 \Rightarrow_g^* s_5$), and also that four rules of `g` have been used to generate `s5` from `s1`, thereby producing intermediate sentential forms `s2`, `s3` and `s4` (that is, $s_1 \Rightarrow_g s_2 \Rightarrow_g s_3 \Rightarrow_g s_4 \Rightarrow_g s_5$). Then, this list of sentential forms can be represented as `sflist g [s1; s2; s3; s4; s5]`. `sflist g [s2]` (among others) and `sflist g []` are also valid.

The equivalence of `derives` and `sflist` is given by lemma `derives_sflist`, which states that every derivation is in relation to a corresponding list of sentential foms, and vice-versa:

```
Lemma derives_sflist:
∀ g: cfg _ _,
∀ s1 s2: sf,
derives g s1 s2 ↔
∃ l: list sf,
sflist g l ∧
hd [] l = s1 ∧
last l [] = s2.
```

In the above statement, `hd` and `last` are functions from the standard Coq library on lists (`List`) that return, respectively, the first and last elements of the argument list. The proof of this lemma is by induction in the hypothesis, in both directions of the implication (on `derives` and `sflist`).

The predicates `produces_empty` and `produces_non_empty` determine, respectively, whether a grammar produces the empty string and whether a grammar produces a non-empty string:

```
Definition produces_empty
(g: cfg non_terminal terminal): Prop:=
produces g [].

Definition produces_non_empty
(g: cfg non_terminal terminal): Prop:=
∃ s: sentence, produces g s ∧ s ≠ [].
```

Predicate `appears` is used to check whether a symbol (non-terminal or terminal) appears in either the left- or right-hand side of some rule of the grammar:

```
Definition appears (g: cfg) (s: non_terminal + terminal): Prop:=
match s with
| inl n ⇒ ∃ left: non_terminal,
          ∃ right: sf,
          rules g left right ∧ ((n=left) ∨ (In (inl n) right))
| inr t ⇒ ∃ left: non_terminal,
          ∃ right: sf,
        rules g left right ∧ In (inr t) right
end.
```

Function `terminal_lift` converts a terminal symbol into an ordered pair of type `(non_terminal + terminal)`.

```
Definition terminal_lift (t: terminal): non_terminal + terminal:=
inr t.
```

With these definitions, it was possible to prove various lemmas about grammars and derivations, and also operations on grammars, all of which were useful when proving the main theorems of this work.

Two grammars g_1 (with start symbol S_1) and g_2 (with start symbol S_2) are *equivalent* (denoted $g_1 \equiv g_2$) if they generate the same language, that is, $\forall s, (S_1 \Rightarrow^*_{g_1} s) \leftrightarrow (S_2 \Rightarrow^*_{g_2} s)$. This is represented in our formalization in Coq by the predicate `g_equiv`:

```
Definition g_equiv
(non_terminal1 non_terminal2 terminal : Type)
(g1: cfg non_terminal1 terminal)
(g2: cfg non_terminal2 terminal): Prop:=
∀ s: sentence,
produces g1 s ↔ produces g2 s.
```

6.2.3 Languages

A language is a set of strings over a given alphabet.

It is useful to define the language generated by a grammar as the set of terminal strings generated by the grammar through derivations:

$$L(G) = \{w \,|\, S \Rightarrow^*_g w\}$$

From the formalization point of view, we have defined a language as a function that is parametrized over a certain type (representing the set of terminal symbols), takes a string built from elements of this type and returns a proposition asserting that the string belongs to the language:

```
Definition lang (terminal: Type):= sentence → Prop.
```

The language generated by a grammar is then a function whose return value is the predicate `produces` presented earlier:

```
Definition lang_of_g (g: cfg): lang :=
fun w: sentence ⇒ produces g w.
```

We are now able to define the equality of two languages, by assuring that they contain exactly the same strings:

```
Definition lang_eq (l k: lang) :=
∀ w, l w ↔ k w.
```

The symbol == is a notation for `lang_eq`[2].

```
Infix "==" := lang_eq (at level 80).
```

Finally, a language is a context-free language if there exists a context-free grammar that generates this language:

```
Definition cfl (terminal: Type) (l: lang terminal): Prop:=
∃ non_terminal: Type,
∃ g: cfg non_terminal terminal,
l == lang_of_g g.
```

Predicates `contains_empty` and `contains_non_empty` are the language counterparts of the previously presented grammar predicates (respectively `produces_empty` and `produces_non_empty`) and state, respectively, whether a language contains the empty string and whether a languages contains a non-empty string:

```
Definition contains_empty (l: lang): Prop:=
l [].

Definition contains_non_empty (l: lang): Prop:=
∃ w: sentence,
l w ∧ w ≠ [].
```

6.3 Generic CFG Library

The definitions presented in the previous section allow the construction of a generic library of fundamental lemmas on context-free grammars. This library (named `cfg.v`) contains 4,393 lines of Coq script (~18.3% of the total), 105 lemmas theorems (~19.7% of the total) and supports the whole formalization, including the specific results discussed next. The following are some examples of the statements contained and proved in it ($s, s', s_1, s'_1, s_2, s'_2, s_3, s_4$ and *right* are sentential forms, n is a non-terminal symbol, w and w' are sentences and g_1, g_2 and g_3 are context-free grammars):

- Derivation transitivity:
  $\forall g, s_1, s_2, s_3,\ (s_1 \Rightarrow_g^* s_2) \rightarrow (s_2 \Rightarrow_g^* s_3) \rightarrow (s_1 \Rightarrow_g^* s_3)$
- Closure under contexts:
  $\forall g, s_1, s_2, s, s',\ (s_1 \Rightarrow_g^* s_2) \rightarrow (s \cdot s_1 \cdot s' \Rightarrow_g^* s \cdot s_2 \cdot s')$

[2]The priority (or precedence) of the == operator, in relation to previously defined operators, is given by the number that follows the words `at level`, in this case `80`. The lowest precedence corresponds to level 100, and higher precedences should be assigned lower levels.

- Derivation independence:
  $\forall g, s_1, s_2, s_3,$
  $(s_1 \cdot s_2 \Rightarrow_g^* s_3) \rightarrow \exists s_1', s_2' \,|\, (s_3 = s_1' \cdot s_2') \wedge (s_1 \Rightarrow_g^* s_1') \wedge (s_2 \Rightarrow_g^* s_2')$

- Derivation of a string of terminals from a non-terminal symbol:
  $\forall g, s_1, s_2, n, w, \; (s_1 \cdot n \cdot s_2 \Rightarrow_g^* w) \rightarrow \exists\, w' \,|\, (n \Rightarrow_g^* w')$

- Direct or indirect derivation:
  $\forall g, n, w, \; (n \Rightarrow_g^* w) \rightarrow (n \rightarrow_g w) \vee (\exists\, right \,|\, n \rightarrow_g right \wedge right \Rightarrow_g^* w)$

- Grammar equivalence transitivity (two grammars are equivalent if they generate the same language):
  $\forall g_1, g_2, g_3, \; (g_1 \equiv g_2) \wedge (g_2 \equiv g_3) \rightarrow (g_1 \equiv g_3)$

Besides that, additional notions of derivations were defined in order to simplify some proofs. This is the case, for example, where a reduction (the opposite of a derivation), a direct derivation or an induction on the number of derivation steps are required. For these cases, we have defined the predicates `derives2` (which reduces a sentential form according to a rule), `derives3` (which represents the derivation of a sentence directly from a non-terminal) and `derives6` (which controls the number of rules applied in the derivation):

```
Inductive derives2
  (non_terminal terminal : Type)
  (g : cfg non_terminal terminal)
  : sf → sf → Prop :=
  | derives2_refl :
     ∀ s : sf,
     derives2 g s s
  | derives2_step :
     ∀ (s1 s2 s3 : sf)
     ∀ (left : non_terminal)
     ∀ (right : sf),
     derives2 g (s1 ++right ++s2) s3 →
     rules g left right →
     derives2 g (s1 ++inl left :: s2) s3

Inductive derives3
  (g: cfg): non_terminal → sentence → Prop :=
  | derives3_rule:
     ∀ (n: non_terminal) (lt: sentence),
     rules g n (map inr lt) → derives3 g n lt
  | derives3_step:
     ∀ (n: non_terminal) (ltnt: sf) (lt: list terminal),
     rules g n ltnt → derives3_aux g ltnt lt → derives3 g n lt
with derives3_aux (g: cfg): sf → sentence → Prop :=
  | derives3_aux_empty:
     derives3_aux g [] []
  | derives3_aux_t:
     ∀ (t: terminal) (ltnt: sf) (lt: sentence),
     derives3_aux g ltnt lt → derives3_aux g (inr t :: ltnt) (t :: lt)
  | derives3_aux_nt:
     ∀ (n: non_terminal) (lt lt': sentence) (ltnt: sf),
     derives3_aux g ltnt lt → derives3 g n lt' →
    derives3_aux g (inl n :: ltnt) (lt' ++lt).

Inductive derives6
  (non_terminal terminal : Type)
  (g : cfg non_terminal terminal)
  : nat → sf → sf → Prop :=
  | derives6_0 :
     ∀ s : sf,
     derives6 g 0 s s
  | derives6_sum :
     ∀ (left : non_terminal)
     ∀ (right : sf)
     ∀ (i : nat)
     ∀ (s1 s2 s3 : sf),
     rules g left right →
     derives6 g i (s1 ++right ++s2) s3 →
     derives6 g (S i) (s1 ++[inl left] ++s2) s3
```

The equivalence of definitions `derives`, `derives2`, `derives3` and `derives6` has been proved. For the first case, we proved that `derives g s1 s2` ↔ `derives2 g s1 s2`. For the second and third cases we proved, respectively, that `derives g n (map inr s)` ↔ `derives3 g n s` and that `derives g s1 s2` ↔ ∃ *n*, `derives6 g n s1 s2`. The corresponding lemmas are:

- `derives_equiv_derives2`;
- `derives_equiv_derives3` and
- `derives_equiv_derives6`.

As an example, consider grammar *G* of Section 6.2.1. We present below proofs of the fact that the string *aab* can also be directly derived in *G* according to the new definitions, however using different strategies.

```
Lemma derives2_g_aab:
derives2 g [inl S'] [inr a; inr a; inr b].
Proof.
assert (derives2 g [inr a; inr a; inr b] [inr a; inr a; inr b]).
  {
  apply derives2_refl.
  }
change [inr a; inr a; inr b] with ([inr a; inr a] ++[inr nt b]) in H.
apply derives2_step
with (s1:= [inr a; inr a]) (right:= [inr b]) (s2:= [])  (left:= S') in H.
- change ([inr a; inr a] ++[inl S']) with ([inr a] ++[inr a; inl S']) in H.
  apply derives2_step
  with (s1:= [inr a]) (right:= [inr a; inl S']) (s2:= [])  (left:= S') in H.
  + change ([inr a] ++[inl S']) with ([inr a; inl S']) in H.
    apply derives2_step
   with (s1:= [])  (right:= [inr a; inl S']) (s2:= [])  (left:= S') in H.
    * simpl in H; exact H.
    * simpl; apply r1.
  + simpl; apply r1.
- simpl; apply r2.
Qed.

Lemma derives3_g_aab:
derives3 g S' [a; a; b].
Proof.
apply derives3_step with (ltnt:= [inr a; inl S']).
- simpl; apply r1.
- apply derives3_aux_t.
  rewrite ← app_nil_r.
  apply derives3_aux_nt.
  + apply derives3_aux_empty.
  + apply derives3_step with (ltnt:= [inr a; inl S']).
    * simpl; apply r1.
    * apply derives3_aux_t.
      rewrite ← app_nil_r.
      {
      apply derives3_aux_nt.
      - apply derives3_aux_empty.
      - apply derives3_rule; simpl; apply r2.
      }
Qed.
```

```
Lemma derives6_g_aab:
derives6 g 3 [inl S'] [inr a; inr a; inr b].
Proof.
assert (derives6 g 0 [inr a; inr a; inr b] [inr a; inr a; inr b]).
  {
  apply derives6_0.
  }
change [inr a; inr a; inr b] with ([inr a; inr a] ++[inr nt b]) in H.
apply derives6_sum
with (s1:= [inr a; inr a]) (right:= [inr b]) (s2:= []) (left:= S') in H.
- change ([inr a; inr a] ++[inl S'] ++[]) with ([inr a] ++[inr a; inl S']) in H.
  apply derives6_sum
  with (s1:= [inr a]) (right:= [inr a; inl S']) (s2:= []) (left:= S') in H.
  + change ([inr a] ++[inl S'] ++[]) with ([inr a; inl S']) in H.
    apply derives6_sum
   with (s1:= []) (right:= [inr a; inl S']) (s2:= []) (left:= S') in H.
    * simpl in H; exact H.
    * simpl; apply r1.
  + simpl; apply r1.
- simpl; apply r2.
Qed.
```

Despite the diversity of definitions for a derivation (`derives` through `derives7`), not all of them have been used in the formalization and some remain only as exercises. Still, the predicate `derives` is used as much as possible in the statements in order to improve readability. Then, when necessary, it is substituted for the alternative predicate that suits better the proof that follows.

6.4 Method

Except for the Pumping Lemma, this formalization is essentially about context-free grammar manipulation. That is, about the definition of a new grammar from a previous one (or two), such that it satisfies some very specific properties. This is exactly the case when we define new grammars that generate the union, concatenation, closure (Kleene star) of given input grammar(s). Also, when we create new grammars that exclude empty rules, unit rules, useless symbols and inaccessible symbols from the original ones. Finally, it is also the case when we construct a new grammar, based on some other input grammar, such that the new grammar generates the same language and observes the Chomsky Normal Form.

Schematically, in the general case the mapping of grammar $g_1 = (V_1, \Sigma, P_1, S_1)$ into grammar $g_2 = (V_2, \Sigma, P_2, S_2)$ requires the definition of a new set of non-terminal symbols N_2, a new set of rules P_2 and a new start symbol S_2:

$$\begin{array}{rcl} g_1 & \rightsquigarrow & g_2 \\ V_1 & \rightsquigarrow & V_2 \\ P_1 & \rightsquigarrow & P_2 \\ S_1 & \rightsquigarrow & S_2 \end{array}$$

Similarly, the mapping of grammar $g_1 = (V_1, \Sigma, P_1, S_1)$ and grammar $g_2 = (V_2, \Sigma, P_2, S_2)$ into grammar $g_3 = (V_3, \Sigma, P_3, S_3)$ requires the definition of a new set of non-terminal symbols N_3, a new set of rules P_3 and a new start symbol S_3:

$$\begin{array}{rcl} g_1, g_2 & \rightsquigarrow & g_3 \\ V_1, V_2 & \rightsquigarrow & V_3 \\ P_1, P_2 & \rightsquigarrow & P_3 \\ S_1, S_2 & \rightsquigarrow & S_3 \end{array}$$

For all cases of grammar manipulation, we consider that the original and final sets of terminal symbols are the same. Also, we have devised the following common approach to constructing the desired grammars:

1. Depending on the case, inductively define the type of the new non-terminal symbols; this will be important, for example, when we want to guarantee that the start symbol of the grammar does not appear in the right-hand side of any rule or when we have to construct new non-terminals from the existing ones; the new type may use some (or all) symbols of the previous type (via mapping), or also add new symbols;

2. Inductively define the rules of the new grammar, in a way that it allows the construction of the proofs that the resulting grammar has the required properties; these new rules will likely make use of the new non-terminal symbols described above; the new definition may exclude some of the original rules, keep others (via mapping) and still add new ones;

3. Define the new grammar by using the new type of non-terminal symbols and the new rules; define the new start symbol (which might be a new symbol or an existing one) and build a proof of the finiteness of the set of rules for this new grammar;

4. State and prove all the lemmas and theorems that will assert that the newly defined grammar has the desired properties;

5. Consolidate the results within the same scope and finally with the previously obtained results.

In the following sections, this approach will be explored with further detail for each main result listed in the beginning of the section.

6.5 Closure Properties

After context-free grammars and derivations were defined, and the generic CFG library was built, the basic operations of union, concatenation and closure for context-free grammars were described in a rather straightforward way. These operations provide, as their name suggests, new context-free grammars that generate, respectively, the union (Section 6.5.1), concatenation (Section 6.5.2) and the Kleene star closure (Section 6.5.3) of the language(s) generated by the input grammar(s)[3].

6.5.1 Union

Given two arbitrary context-free grammars g_1 and g_2, we want to construct g_3 such that $L(g_3) = L(g_1) \cup L(g_2)$ (that is, the language generated by g_3 is the union of the languages generated by g_1 and g_2).

The classical informal proof constructs $g_3 = (V_3, \Sigma, P_3, S_3)$ from g_1 and g_2 such that $N_3 = N_1 \cup N_2 \cup \{S_3\}$ and $P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1, S_3 \rightarrow S_2\}$. Thus:

$$\begin{array}{rcl} g_1,\ g_2 & \rightsquigarrow & g_3 \\ P_1,\ P_2 & \underbrace{\rightsquigarrow}_{+2} & P_3 \\ \Sigma,\ \Sigma & \underbrace{\rightsquigarrow}_{same} & \Sigma \\ V_1,\ V_2 & \underbrace{\rightsquigarrow}_{+1} & V_3 \\ S_1,\ S_2 & \underbrace{\rightsquigarrow}_{new} & S_3 \end{array}$$

The first definition below (`g_uni_nt`) represents the type of the non-terminal symbols of the union grammar, created from the non-terminal symbols of the source grammars (respectively, `non_terminal1` and `non_terminal2`). Initially, the non-terminals of the source grammars are mapped to non-terminals of the union grammar. Second, there is the need to add a new and unique non-terminal symbol (`Start_uni`), which will be the start symbol of the union grammar.

```
Inductive g_uni_nt (non_terminal_1 non_terminal_2 : Type): Type:=
| Start_uni
| Transf1_uni_nt: non_terminal_1 → g_uni_nt
| Transf2_uni_nt: non_terminal_2 → g_uni_nt.
```

[3]The results of this section are available in libraries `union.v`, `concatenation.v` and `closure.v`.

The functions g_uni_sf_lift1 and g_uni_sf_lift2 simply map sentential forms from, respectively, the first or the second grammar, and produce sentential forms of the union grammar. This will be useful when defining the rules of the union grammar.

```
Notation sf1:= (list (non_terminal_1 + terminal)).
Notation sf2:= (list (non_terminal_2 + terminal)).
Notation sfu:= (list (g_uni_nt + terminal)).

Definition g_uni_sf_lift1 (c: non_terminal_1 + terminal)
: g_uni_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf1_uni_nt nt)
  | inr t  ⇒ inr t
  end.

Definition g_uni_sf_lift2 (c: non_terminal_2 + terminal)
: g_uni_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf2_uni_nt nt)
  | inr t  ⇒ inr t
  end.
```

The rules of the union grammar are represented by the g_uni_rules inductive definition. Constructors Start1_uni and Start2_uni state that two new rules are added to the union grammar: respectively the rule that maps the new start symbol to the start symbol of the first grammar, and the rule that does the same for the second grammar. Then, constructors Lift1_uni and Lift2_uni simply map rules of first (resp. second) grammar into rules of the union grammar.

```
Inductive g_uni_rules
(non_terminal_1 non_terminal_2 terminal : Type)
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal)
: g_uni_nt → sfu → Prop :=
| Start1_uni:
   g_uni_rules g1 g2 Start_uni [inl (Transf1_uni_nt (start_symbol g1))]
| Start2_uni:
   g_uni_rules g1 g2 Start_uni [inl (Transf2_uni_nt (start_symbol g2))]
| Lift1_uni:
   ∀ nt: non_terminal_1,
   ∀ s: sf1,
   rules g1 nt s →
   g_uni_rules g1 g2 (Transf1_uni_nt nt) (map g_uni_sf_lift1 s)
```

```
| Lift2_uni:
   ∀ nt: non_terminal_2,
   ∀ s: sf2,
   rules g2 nt s →
   g_uni_rules g1 g2 (Transf2_uni_nt nt) (map g_uni_sf_lift2 s).
```

Finally, g_uni describes how to create a union grammar from two arbitrary source grammars. It uses the previous definitions to give values to each of the components of a new grammar definition.

```
Definition g_uni
(non_terminal_1 non_terminal_2 terminal : Type)
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal)
: (cfg g_uni_nt terminal):=
   {| start_symbol:= Start_uni;
      rules:= g_uni_rules g1 g2;
      rules_finite:= g_uni_finite g1 g2 |}.
```

To summarize, we relate below the transformations that create the new union grammar to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals (inductive definition `g_uni_nt`):
    - All the non-terminals of g_1 (constructor `Transf1_uni_nt`);
    - All the non-terminals of g_2 (constructor `Transf2_uni_nt`);
    - A fresh new non-terminal symbol S_3 (constructor `Start_uni`).
- For the new set of rules (inductive definition `g_uni_rules`):
    - All the rules of g_1 (constructor `Lift1_uni`);
    - All the rules of g_2 (constructor `Lift2_uni`);
    - Two new rules: $S_3 \rightarrow S_1$ and $S_3 \rightarrow S_2$ (respectively constructors `Start1_uni` and `Start2_uni`).
- For the new grammar (definition `g_uni`):
    - The new set of non-terminals (implicit);
    - The new set of rules (`g_uni_rules g1 g2`);
    - The new non-terminal as the start symbol (`Start_uni` as S_3).

As an example, consider grammars G_1 and G_2:

- $G_1 = (\{S_1, X_1, a, b\}, \{a, b\}, \{S_1 \rightarrow aX_1, X_1 \rightarrow aX_1 \mid b\}, S_1)$;

- $G_2 = (\{S_2, X_2, a, c\}, \{a, c\}, \{S_2 \to aX_2, X_2 \to aX_2 \mid c\}, S_2)$.

Then, the new grammar G_3 that generates $L(G_1) \cup L(G_2)$ can be expressed as:

$$G_3 = (\{S_3, S_1, S_2, X_1, X_2, a, b, c\}, \{a, b, c\}, P_3, S_3)$$

with P_3 containing the following rules:

$$\begin{array}{lcl} S_3 & \to & S_1 \mid S_2 \\ S_1 & \to & aX_1 \\ X_1 & \to & aX_1 \mid b \\ S_2 & \to & aX_2 \\ X_2 & \to & aX_2 \mid c \end{array}$$

Observe, in this example, that S_3 (the start symbol of G_3) is the newly introduced non-terminal symbol and corresponds to constructor `Start_uni` in the definition of `g_uni_nt`. Also, that since `g_uni` is parametrized on types `non_terminal_1` and `non_terminal_2`, this means that the original names of the non-terminals are preserved, respectively, under constructors `Transf1_uni_nt` and `Transf2_uni_nt` (in the example represented by numeric subscripts).

The Coq scripts for `g1`, `g2` and `g3` can be defined as (the proofs of finiteness of the grammars have been omitted):

```
Inductive non_terminal1: Type:=
| S1
| X1.

Inductive non_terminal2: Type:=
| S2
| X2.

Inductive terminal: Type:=
| a
| b
| c.

Inductive rs1: non_terminal1 → list (non_terminal1 + terminal) → Prop:=
| r11: rs1 S1 [inr a; inl X1]
| r12: rs1 X1 [inr a; inl X1]
| r13: rs1 X1 [inr b].

Definition g1: cfg non_terminal1 terminal := {|
start_symbol:= S1;
rules:= rs1;
rules_finite:= rs1_finite |}.

Inductive rs2: non_terminal2 → list (non_terminal2 + terminal) → Prop:=
| r21: rs2 S2 [inr a; inl X2]
| r22: rs2 X2 [inr a; inl X2]
| r23: rs2 X2 [inr c].

Definition g2: cfg non_terminal2 terminal := {|
start_symbol:= S2;
rules:= rs2;
rules_finite:= rs2_finite |}.

Definition g3:= g_uni g1 g2.
```

Note that `g1` and `g2` use different sets of non-terminal symbols (in this case, respectively `non_terminal1` and `non_terminal2`) but share the same set of terminal symbols (`terminal`), as required by the previous definitions. The union grammar `g3` is then defined simply by `g_uni g1 g2`. Observe, also, that since the non-terminal symbols correspond to the constructors of an inductive definition, it is not possible to use the same non-terminal symbol for two different grammars, as this would violate the rule of Coq that requires these names to be different.

Similar definitions were created to represent the concatenation of any two grammars and the closure of a grammar, as explained n the folowing sections.

6.5.2 Concatenation

Given two arbitrary context-free grammars g_1 and g_2, we want to construct g_3 such that $L(g_3) = L(g_1) \cdot L(g_2)$ (that is, the language generated by g_3 is the concatenation of the languages generated by g_1 and g_2).

The classical informal proof constructs $g_3 = (V_3, \Sigma, P_3, S_3)$ from g_1 and g_2 such that $N_3 = N_1 \cup N_2 \cup \{S_3\}$ and $P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 S_2\}$. Thus:

$$\begin{array}{rcl}
g_1,\ g_2 & \rightsquigarrow & g_3 \\
P_1,\ P_2 & \underbrace{\rightsquigarrow}_{+1} & P_3 \\
\Sigma,\ \Sigma & \underbrace{\rightsquigarrow}_{\mathit{same}} & \Sigma \\
V_1,\ V_2 & \underbrace{\rightsquigarrow}_{+1} & V_3 \\
S_1,\ S_2 & \underbrace{\rightsquigarrow}_{\mathit{new}} & S_3
\end{array}$$

The following definitions are used in our formalization:

```
Inductive g_cat_nt (non_terminal_1 non_terminal_2 terminal : Type): Type:=
| Start_cat
| Transf1_cat_nt: non_terminal_1 → g_cat_nt
| Transf2_cat_nt: non_terminal_2 → g_cat_nt.

Notation sf1:= (list (non_terminal_1 + terminal)).
Notation sf2:= (list (non_terminal_2 + terminal)).
Notation sfc:= (list (g_cat_nt + terminal)).

Definition g_cat_sf_lift1 (c: non_terminal_1 + terminal):
g_cat_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf1_cat_nt nt)
  | inr t  ⇒ inr t
  end.

Definition g_cat_sf_lift2 (c: non_terminal_2 + terminal):
g_cat_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf2_cat_nt nt)
  | inr t  ⇒ inr t
  end.

Inductive g_cat_rules
(non_terminal_1 non_terminal_2 terminal : Type)
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal)
: g_cat_nt → sfc → Prop :=
| New_cat:
   g_cat_rules g1 g2 Start_cat
   ([ inl (Transf1_cat_nt (start_symbol g1))]++
    [inl (Transf2_cat_nt (start_symbol g2))])
| Lift1_cat:
   ∀ nt s,
   rules g1 nt s →
   g_cat_rules g1 g2 (Transf1_cat_nt nt) (map g_cat_sf_lift1 s)
| Lift2_cat:
   ∀ nt s,
   rules g2 nt s →
   g_cat_rules g1 g2 (Transf2_cat_nt nt) (map g_cat_sf_lift2 s).
```

```
Definition g_cat
(non_terminal_1 non_terminal_2 terminal : Type)
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal)
: (cfg g_cat_nt terminal):=
  {| start_symbol:= Start_cat;
     rules:= g_cat_rules g1 g2;
     rules_finite:= g_cat_finite g1 g2 |}.
```

In this case, the new grammar (`g_cat g1 g2`) is built in such a way that it has all the rules of `g1` and `g2`, plus a new rule that maps the new start symbol (`Start_cat`) to the concatenation of the start symbols of the argument grammars (respectively `start_symbol g1` and `start_symbol g2`).

To summarize, we relate below the transformations that create the new concatenation grammar to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals (inductive definition `g_cat_nt`):
    - All the non-terminals of g_1 (constructor `Transf1_cat_nt`);
    - All the non-terminals of g_2 (constructor `Transf2_cat_nt`);
    - A fresh new non-terminal symbol S_3 (constructor `Start_cat`).
- For the new set of rules (inductive definition `g_cat_rules`):
    - All the rules of g_1 (constructor `Lift1_cat`);
    - All the rules of g_2 (constructor `Lift2_cat`);
    - One new rule: $S_3 \rightarrow S_1 S_2$ (constructor `New_cat`).
- For the new grammar (definition `g_cat`):
    - The new set of non-terminals (implicit);
    - The new set of rules (`g_cat_rules g1 g2`);
    - The new non-terminal as the start symbol (`Start_cat` as S_3).

As an example, consider again grammars G_1 and G_2 of Section 6.5.1:

- $G_1 = (\{S_1, X_1, a, b\}, \{a, b\}, \{S_1 \rightarrow aX_1, X_1 \rightarrow aX_1 \mid b\}, S_1)$;
- $G_2 = (\{S_2, X_2, a, c\}, \{a, c\}, \{S_2 \rightarrow aX_2, X_2 \rightarrow aX_2 \mid c\}, S_2)$.

Then, the new grammar G_3 that generates $L(G_1) \cdot L(G_2)$ can be expressed as:

$$G_3 = (\{S_3, S_1, S_2, X_1, X_2, a, b, c\}, \{a, b, c\}, P_3, S_3)$$

with P_3 containing the following rules:

$$\begin{array}{lcl} S_3 & \rightarrow & S_1 S_2 \\ S_1 & \rightarrow & aX_1 \\ X_1 & \rightarrow & aX_1 \mid b \\ S_2 & \rightarrow & aX_2 \\ X_2 & \rightarrow & aX_2 \mid c \end{array}$$

Observe, in this example, that S_3 (the start symbol of G_3) is the newly introduced non-terminal symbol and corresponds to constructor `Start_cat` in the definition of `g_cat_nt`. Also, that since `g_cat` is parametrized on types `non_terminal_1` and `non_terminal_2`, this means that the original names of the non-terminals are preserved, respectively, under constructors `Transf1_cat_nt` and `Transf2_cat_nt` (in the example represented by numeric subscripts).

The corresponding concatenation grammar `g3`, in this case, can be obtained directly from the statement:

```
Definition g3:= g_cat g1 g2.
```

supposing that `g1` and `g2` were defined as in Section 6.5.1.

6.5.3 Kleene Star

Given an arbitrary context-free grammar g_1, we want to construct g_2 such that $L(g_2) = (L(g_1))^*$ (that is, the language generated by g_2 is the reflexive and transitive concatenation (Kleene star) of the language generated by g_1).

The classical informal proof constructs $g_2 = (V_2, \Sigma, P_2, S_2)$ from g_1 such that $N_2 = N_1 \cup N_2 \cup \{S_2\}$ and $P_2 = P_1 \cup P_2 \cup \{S_2 \rightarrow S_2 S_1, S_2 \rightarrow S_1\}$. Thus:

$$\begin{array}{ccc} g_1 & \rightsquigarrow & g_2 \\ P_1 & \underbrace{\rightsquigarrow}_{+2} & P_2 \\ \Sigma & \underbrace{\rightsquigarrow}_{same} & \Sigma \\ V_1 & \underbrace{\rightsquigarrow}_{+1} & V_2 \\ S_1 & \underbrace{\rightsquigarrow}_{new} & S_2 \end{array}$$

The following definitions are used in our formalization:

```
Notation sfc:= (list (g_clo_nt + terminal)).

Inductive g_clo_nt (non_terminal : Type): Type :=
| Start_clo : g_clo_nt
| Transf_clo_nt : non_terminal → g_clo_nt.

Definition g_clo_sf_lift (c: non_terminal + terminal):
g_clo_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf_clo_nt nt)
  | inr t  ⇒ inr t
  end.

Inductive g_clo_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: g_clo_nt → sfc → Prop :=
| New1_clo:
   g_clo_rules g Start_clo ([inl Start_clo] ++
   [inl (Transf_clo_nt (start_symbol g))])
| New2_clo:
   g_clo_rules g Start_clo []
| Lift_clo:
   ∀ nt: non_terminal,
   ∀ s: sf,
   rules g nt s →
   g_clo_rules g (Transf_clo_nt nt) (map g_clo_sf_lift s).

Definition g_clo (g: cfg non_terminal terminal):
(non_terminal terminal : Type)
(g: cfg g_clo_nt terminal):=
 {| start_symbol:= Start_clo;
    rules:= g_clo_rules g;
    rules_finite:= g_clo_finite g |}.
```

In this case, the new grammar (`g_clo g`) is built in such a way that it has all the rules of `g`, plus two new rules: one that generates the empty string directly from the start symbol of `g_clo g`, and another one that allows for the arbitrary concatenation of strings generated by `g`.

To summarize, we relate below the transformations that create the new Kleene star grammar to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals (inductive definition `g_clo_nt`):
  - All the non-terminals of g_1 (constructor `Transf_clo_nt`);
  - A fresh new non-terminal symbol S_2 (constructor `Start_clo`).
- For the new set of rules (inductive definition `g_clo_rules`):
  - All the rules of g_1 (constructor `Lift1_clo`);
  - Two new rules: $S_2 \to S_2S_1$ and $S_2 \to \varepsilon$ (respectively constructors `New1_clo` and `New2_clo`).
- For the new grammar (definition `g_clo`):
  - The new set of non-terminals (implicit);
  - The new set of rules (`g_clo_rules g`);
  - The new non-terminal as the start symbol (`Start_clo` as S_2).

As an example, consider once more grammar G_1 of Section 6.5.1:

$$G_1 = (\{S_1, X_1, a, b\}, \{a, b\}, \{S_1 \to aX_1, X_1 \to aX_1 \mid b\}, S_1)$$

Then, the new grammar G_2 that generates $L(G_1)^*$ can be expressed as:

$$G_2 = (\{S_2, S_1, X_1, a, b\}, \{a, b\}, P_2, S_2)$$

with P_2 containing the following rules:

$$\begin{array}{lcl} S_2 & \to & \varepsilon \\ S_2 & \to & S_2S_1 \\ S_1 & \to & aX_1 \\ X_1 & \to & aX_1 \mid b \end{array}$$

Observe, in this example, that S_2 (the start symbol of G_2) is the newly introduced non-terminal symbol and corresponds to constructor `Start_clo` in the definition of `g_clo_nt`. Also, that since `g_clo` is parametrized on type `non_terminal`, this means that the original names of the non-terminals are preserved under constructor `Transf_clo_nt`.

The corresponding Kleene star closure grammar `g3`, in this case, can be obtained directly from the statement:

```
Definition g2:= g_clo g1.
```

supposing that `g1` was defined as in Section 6.5.1.

6.5.4 Correctness and Completeness

Although simple in their structure, it must be proved that the definitions g_uni, g_cat and g_clo always produce the correct result. In other words, these definitions must be "certified", which is one of the main goals of formalization. In order to accomplish this, we must first state the theorems that capture the expected semantics of these definitions. Finally, we have to derive proofs of the correctness of these theorems.

This can be done with a pair of theorems for each grammar definition: the first relates the output to the inputs, and the other one does the converse, providing assumptions about the inputs once an output is generated. This is necessary in order to guarantee that the definitions do only what one would expect, and no more.

In what follows, an informal statement is presented right before the corresponding Coq theorem. This is intended to abstract over the necessary mappings that occur in the Coq terms, due to the different types of non-terminals, sentential forms etc involved.

For union, we need to prove (considering that g_3 is the union of g_1 and g_2 and S_3, S_1 and S_2 are, respectively, the start symbols of g_3, g_1 and g_2):

$$\forall g_1, g_2, s_1, s_2, (S_1 \Rightarrow^*_{g_1} s_1 \rightarrow S_3 \Rightarrow^*_{g_3} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2 \rightarrow S_3 \Rightarrow^*_{g_3} s_2)$$

which translates in Coq into:

```
Theorem g_uni_correct:
∀ g1: cfg non_terminal_1 terminal,
∀ g2: cfg non_terminal_2 terminal,
∀ s1: sf1,
∀ s2: sf2,
(generates g1 s1 → generates (g_uni g1 g2) (map g_uni_sf_lift1 s1))
∧
(generates g2 s2 → generates (g_uni g1 g2) (map g_uni_sf_lift2 s2)).
```

The strategy used in the proof of the correctness of the union is as follows:

- Split the statement $(S_1 \Rightarrow^*_{g_1} s_1 \rightarrow S_3 \Rightarrow^*_{g_3} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2 \rightarrow S_3 \Rightarrow^*_{g_3} s_2)$ in two parts:
    - First part: $S_1 \Rightarrow^*_{g_1} s_1 \rightarrow S_3 \Rightarrow^*_{g_3} s_1$;
        - Split the goal $S_3 \Rightarrow^*_{g_3} s_1$ into subgoals $S_3 \Rightarrow_{g_3} S_1$ and $S_1 \Rightarrow^*_{g_3} s_1$; (this is done by lemma derives_trans)
        - Prove that $S_3 \Rightarrow_{g_3} S_1$; (this is a direct consequence of the definition of g_uni_rules, constructor Start1_uni)
        - Prove that $S_1 \Rightarrow^*_{g_1} s_1 \rightarrow S_1 \Rightarrow^*_{g_3} s_1$; (this is done by lemma derives_add_uni_left, which is proved by induction on predicate derives in $S_1 \Rightarrow^*_{g_1} s_1$)

- Second part: $S_2 \Rightarrow^*_{g_2} s_2 \rightarrow S_3 \Rightarrow^*_{g_3} s_2$;
  - Split the goal $S_3 \Rightarrow^*_{g_3} s_2$ into subgoals $S_3 \Rightarrow_{g_3} S_2$ and $S_2 \Rightarrow^*_{g_3} s_2$; (this is done by lemma `derives_trans`)
  - Prove that $S_3 \Rightarrow_{g_3} S_2$; (this is a direct consequence of the definition of `g_uni_rules`, constructor `Start2_uni`)
  - Prove that $S_2 \Rightarrow^*_{g_2} s_2 \rightarrow S_2 \Rightarrow^*_{g_3} s_2$; (this is done by lemma `derives_add_uni_right`, which is proved by induction on predicate `derives` in $S_2 \Rightarrow^*_{g_2} s_2$)

For the converse of union we have:

$$\forall s_3, (S_3 \Rightarrow^*_{g_3} s_3) \rightarrow (S_1 \Rightarrow^*_{g_1} s_3) \vee (S_2 \Rightarrow^*_{g_2} s_3)$$

```
Theorem g_uni_correct_inv:
∀ g1: cfg non_terminal_1 terminal,
∀ g2: cfg non_terminal_2 terminal,
∀ s: sfu,
generates (g_uni g1 g2) s →
(s=[inl (start_symbol (g_uni g1 g2))]) ∨
(∃ s1: sf1, (s=(map g_uni_sf_lift1 s1) ∧ generates g1 s1)) ∨
(∃ s2: sf2, (s=(map g_uni_sf_lift2 s2) ∧ generates g2 s2)).
```

The idea here is to express that, if a string is generated by `g_uni`, then it must only belong to the union of strings generated by the grammars combined by the definition. The statement in Coq, however, is actually more general: it uses the `generates` predicate instead of the `produces` predicate, which allows one to consider sentential forms instead of sentences, and for this reason includes the possibility of S_3 as a sentential form generated by g_3 (in this case with no rule application).

The strategy used in the proof of the completeness of the union uses, first, induction on predicate `derives` (in $S_3 \Rightarrow^*_{g_3} s_3$). The base case is trivial and for the induction case a series of case analysis over lists, disjunctions and the definition of `g_uni_rules` completes the proof.

Together, the two theorems represent the semantics of the context-free grammar union operation. The same ideas have been applied to the statement and proof of the following theorems, relative to the concatenation and closure operations.

For concatenation, the following Coq statement expresses the result we want to prove (considering that g_3 is the concatenation of g_1 and g_2 and S_3, S_1 and S_2 are, respectively, the start symbols of g_3, g_1 and g_2):

$$\forall g_1, g_2, s_1, s_2, (S_1 \Rightarrow^*_{g_1} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2) \rightarrow (S_3 \Rightarrow^*_{g_3} s_1 \cdot s_2)$$

```
Theorem g_cat_correct:
∀ g1: cfg non_terminal_1 terminal,
∀ g2: cfg non_terminal_2 terminal,
∀ s1: sf1,
∀ s2: sf2,
generates g1 s1 ∧ generates g2 s2 →
generates (g_cat g1 g2) ((map g_cat_sf_lift1 s1)++(map g_cat_sf_lift2 s2)).
```

The above theorem states that if context-free grammars `g1` and `g2` generate, respectively, strings `s1` and `s2`, then the concatenation of these two grammars, according to the proposed mapping, generates the concatenation of string `s1` with string `s2`.

The strategy used in the proof of the correctness of the concatenation is as follows:

- Split the goal $S_3 \Rightarrow^*_{g_3} s_1 \cdot s_2$ into subgoals $S_3 \Rightarrow_{g_3} S_1 \cdot S_2$ and $S_1 \cdot S_2 \Rightarrow^*_{g_3} s_1 \cdot s_2$; (this is done by lemma `derives_trans`)
- Prove that $S_3 \Rightarrow_{g_3} S_1 \cdot S_2$; (this is a direct consequence of the definition of `g_cat_rules`, constructor `New_cat`)
- Prove that $S_1 \Rightarrow^*_{g_1} s_1 \wedge S_2 \Rightarrow^*_{g_2} \rightarrow S_1 \cdot S_2 \Rightarrow^*_{g_3} s_1 \cdot s_2$; (this is done by lemma `g_cat_correct_aux`, which is proved by simultaneous induction on predicate `derives` in $S_1 \Rightarrow^*_{g_1} s_1$ and $S_2 \Rightarrow^*_{g_2} s_2$)

As mentioned before, the above theorem alone does not guarantee that `g_cat` will not produce outputs other than the concatenation of its input strings. This idea is captured by the following complementary theorem:

$$\forall s_3, (S_3 \Rightarrow^*_{g_3} s_3) \rightarrow \exists s_1, s_2 \,|\, (s_3 = s_1 \cdot s_2) \wedge (S_1 \Rightarrow^*_{g_1} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2)$$

For the converse of concatenation, the following Coq statement expresses the result we want to prove:

```
Theorem g_cat_correct_inv:
∀ g1: cfg non_terminal_1 terminal,
∀ g2: cfg non_terminal_2 terminal,
∀ s: sfc,
generates (g_cat g1 g2) s →
s = [inl (start_symbol (g_cat g1 g2))] ∨
∃ s1: sf1,
∃ s2: sf2,
s =(map g_cat_sf_lift1 s1)++(map g_cat_sf_lift2 s2) ∧
generates g1 s1 ∧ generates g2 s2.
```

Similarly to the stategy used in the proof of the completeness of the union, the strategy used in the proof of the completeness of the concatenation uses, first, induction on predicate `derives` (in $S_3 \Rightarrow^*_{g_3} s_3$). The base case is trivial and for the induction case a series of case analysis over lists, disjunctions and the definition of `g_cat_rules` completes the proof.

For closure, we have (considering that g_2 is the Kleene star of g_1 and S_2 and S_1 are, respectively, the start symbols of g_2 and g_1):

$$\forall g_1, s_1, s_2, (S_2 \Rightarrow^*_{g_2} \varepsilon) \land ((S_2 \Rightarrow^*_{g_2} s_2) \land (S_1 \Rightarrow^*_{g_1} s_1) \to S_2 \Rightarrow^*_{g_2} s_2 \cdot s_1)$$

which translates in Coq into:

```
Theorem g_clo_correct:
∀ g: cfg non_terminal terminal,
∀ s: sf,
∀ s': sfc,
generates (g_clo g) nil ∧ (generates (g_clo g) s' ∧ generates g s →
generates (g_clo g) (s'++ map g_clo_sf_lift s)).
```

The strategy used in the proof of the correctness of the Kleene star is as follows:

- Split the statement $(S_2 \Rightarrow^*_{g_2} \varepsilon) \land ((S_2 \Rightarrow^*_{g_2} s_2) \land (S_1 \Rightarrow^*_{g_1} s_1) \to S_2 \Rightarrow^*_{g_2} s_2 \cdot s_1)$ in two parts:
    - First part:
        - Prove that $S_2 \Rightarrow^*_{g_2} \varepsilon$;
          (this is a direct consequence of the definition of `g_clo_rules`, constructor `New2_clo`)
    - Second part: $(S_2 \Rightarrow^*_{g_2} s_2) \land (S_1 \Rightarrow^*_{g_1} s_1) \to S_2 \Rightarrow^*_{g_2} s_2 \cdot s_1$;
        - Split the goal $S_2 \Rightarrow^*_{g_2} s_2 \cdot s_1$ into subgoals $S_2 \Rightarrow_{g_2} S_2 \cdot S_1$ and $S_2 \cdot S_1 \Rightarrow^*_{g_2} s_2 \cdot s_1$;
          (this is done by lemma `derives_trans`)
        - Prove that $S_2 \Rightarrow_{g_2} S_2 \cdot S_1$;
          (this is a direct consequence of the definition of `g_clo_rules`, constructor `New1_clo`)
        - Prove that $S_1 \Rightarrow^*_{g_1} s_1 \to S_1 \Rightarrow^*_{g_2} s_1$;
          (this is done by lemma `derives_add_clo`, which is proved by induction on predicate `derives` in $S_1 \Rightarrow^*_{g_1} s_1$)
        - Prove that $S_1 \Rightarrow^*_{g_2} s_1$;
          (use the hypothesis and the previous result)
        - Split the goal $S_2 \cdot S_1 \Rightarrow^*_{g_2} s_2 \cdot s_1$ into subgoals $S_2 \Rightarrow^*_{g_2} s_2$ and $S_1 \Rightarrow^*_{g_2} s_1$;
          (this is done by lemma `derives_combine`)

- Use the hypothesis to prove $S_2 \Rightarrow^*_{g_2} s_2$;
- Use the previous result to prove $S_1 \Rightarrow^*_{g_2} s_1$.

Finally:

$$\forall s_2, (S_2 \Rightarrow^*_{g_2} s_2) \rightarrow (s_2 = \varepsilon) \vee (\exists\, s_1, s_2' \,|\, (s_2 = s_2' \cdot s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2') \wedge (S_1 \Rightarrow^*_{g_1} s_1))$$

```
Theorem g_clo_correct_inv:
∀ g: cfg non_terminal terminal,
∀ s: sfc,
generates (g_clo g) s →
(s=[]) ∨
(s=[inl (start_symbol (g_clo g))]) ∨
(∃ s': sfc,
 ∃ s'': sf,
 generates (g_clo g) s' ∧ generates g s'' ∧ s=s' ++map g_clo_sf_lift s'').
```

Similarly to the stategy used in the proof of the completeness of the union, the strategy used in the proof of the completeness of the Kleene star uses, first, induction on predicate `derives` (in $S_2 \Rightarrow^*_{g_2} s_2$). The base case is trivial and for the induction case a series of case analysis over lists, disjunctions and the definition of `g_clo_rules` completes the proof.

The proofs of all the six main theorems have been completed:

- `g_uni_correct` and `g_uni_correct_inv` for union;
- `g_cat_correct` and `g_cat_correct_inv` for concatenation, and
- `g_clo_correct` and `g_clo_correct_inv` for closure.

In all three cases, the correctness proofs are straightforward and follow closely the informal proofs available in most textbooks. The formalization consists of a set of short and readable lemmas, except for the details related to mappings involving sentential forms. Since every grammar is defined with a different set of non-terminal symbols (i.e. uses a different type for these symbols), sentential forms from one grammar have to "mapped" to sentential forms of another grammar in order to be usable and not break the typing rules of Coq. This required a lot of effort in order to provide and use the correct mapping functions, and also to cope with it during proof construction. This is something that we don´t see in informal proofs, and is definitely a burden when doing the formalization.

The completeness proofs, on the other hand, resulted in single lemmas with reasonably long scripts (~280 lines) in each case. Intermediate lemmas were not easily identifiable as in the correctness cases and, besides the initial induction of predicate `derives`, the long list of various types of case analysis increased the complexity of the scripts, which are thus more difficult to read.

It should be added that the closure operations considered here can be explained in a very intuitive way (either with grammars or automata), and for this reason many textbooks don´t even bother going into the details with mathematical reasoning. Because of this, our formalization was a nice exercise in revealing how simple and intuitive proofs can grow in complexity with many details not even considered before.

6.5.5 Closure over Languages

The results of the previous section were all formulated over grammars, and it is desirable to obtain equivalent versions using languages instead. Thus, we have defined the union, concatenation and closure of arbitrary languages as follows:

```
Inductive l_uni (terminal : Type) (l1 l2: lang terminal): lang terminal:=
| l_uni_l1: ∀ s: sentence, l1 s → l_uni l1 l2 s
| l_uni_l2: ∀ s: sentence, l2 s → l_uni l1 l2 s.

Inductive l_cat (terminal : Type) (l1 l2: lang terminal): lang terminal:=
| l_cat_app: ∀ s1 s2: sentence, l1 s1 → l2 s2 → l_cat l1 l2 (s1 ++s2).

Inductive l_clo (terminal : Type) (l: lang terminal): lang terminal:=
| l_clo_nil: l_clo l []
| l_clo_app: ∀ s1 s2: sentence, (l_clo l) s1 → l s2 → l_clo l (s1 ++s2).
```

With these definitions, it is immediate to prove that the operations of union, concatenation and closure are correct, including the converse versions. However, it remains to be proved that the newly generated languages are also context-free, which leads to the following theorems:

```
Theorem l_uni_is_cfl:
∀ l1 l2: lang terminal,
cfl l1 → cfl l2 → cfl (l_uni l1 l2).

Theorem l_cat_is_cfl:
∀ l1 l2: lang terminal,
cfl l1 → cfl l2 → cfl (l_cat l1 l2).

Theorem l_clo_is_cfl:
∀ l: lang terminal,
cfl l → cfl (l_clo l).
```

In all cases, the proofs obtained rely on (i) the existence of context-free grammars that generated the original languages, a direct consequence of the definition of the predicate `cfl` and (ii) the results that were previously proved for context-free grammars in Section 6.5.4.

6.6 Simplification

The definition of a context-free grammar, and also the operations specified in the previous section, allow for the inclusion of symbols and rules that may not contribute to the language being generated. Besides that, context-free grammars may also contain rules that can be substituted by equivalent smaller and simpler ones. Unit rules, for example, do not expand sentential forms (instead, they just rename the symbols in them) and empty rules can cause them to contract. Although the appropriate use of these features can be important for human communication in some situations, this is not the general case, since it leads to grammars that have more symbols and rules than necessary, making difficult its comprehension and manipulation. Thus, simplification is an important operation on context-free grammars.

Let G be a context-free grammar, $L(G)$ the language generated by this grammar and ε the empty string. Different authors use different terminology when presenting simplification results for context-free grammars. In what follows, we adopt the terminology and definitions of Sudkamp (2006).

Context-free grammar simplification comprises the manipulation of rules and symbols, as described below:

1. An *empty rule* $r \in P$ is a rule whose right-hand side β is empty (e.g. $X \rightarrow \varepsilon$). We formalize, in Section 6.6.1, that for all G there exists G' such that $L(G) = L(G')$ and G' has no empty rules, except for a single rule $S \rightarrow \varepsilon$ if $\varepsilon \in L(G)$; in this case, S (the initial symbol of G') does not appear on the right-hand side of any rule of G';

2. A *unit rule* $r \in P$ is a rule whose right-hand side β contains a single non-terminal symbol (e.g. $X \rightarrow Y$). We formalize, in Section 6.6.2, that for all G there exists G' such that $L(G) = L(G')$ and G' has no unit rules;

3. A symbol $s \in V$ is *useful* (SUDKAMP, 2006, p. 116) if it is possible to derive a string of terminal symbols from it using the rules of the grammar. Otherwise, s is called an *useless symbol*. A useful symbol s is one such that $s \Rightarrow^* \omega$, with $\omega \in \Sigma^*$. Naturally, this definition concerns mainly non-terminals, as terminals are trivially useful. We formalize, in Section 6.6.3, that for all G such that $L(G) \neq \emptyset$, there exists G' such that $L(G) = L(G')$ and G' has no useless symbols;

4. A symbol $s \in V$ is *accessible* (SUDKAMP, 2006, p. 119) if it is part of at least one string generated from the root symbol of the grammar. Otherwise, it is called an *inaccessible symbol*. An accessible symbol s is one such that $S \Rightarrow^* \alpha s \beta$, with $\alpha, \beta \in V^*$. We formalize, in Section 6.6.4, that for all G there exists G' such that $L(G) = L(G')$ and G' has no inaccessible symbols.

Finally, we formalize, in Section 6.6.5, a unification result: that for all G, if G is non-empty, then there exists G' such that $L(G) = L(G')$ and G' has no empty rules (except for one, if

G generates the empty string), no unit rules, no useless symbols, no inaccessible symbols and the start symbol of G' does not appear on the right-hand side of any other rule of G'[4].

In all these four cases and the five grammars that are discussed next (namely `g_emp`, `g_emp'`, `g_unit`, `g_use` and `g_acc`), the proof of `rules_finite` is based on the proof of the corresponding predicate for the argument grammar. Thus, all new grammars satisfy the `cfg` specification and are finite as well.

6.6.1 Empty rules

Result (1) of Section 6.6 is achieved in two steps. In the first step, we map grammar g_1 into an equivalent grammar g_2 (except for the empty string), which is free of empty rules and whose start symbol does not appear on the right-hand side of any rule. Next, we use g_2 to map g_1 into g_3 which is fully equivalent to g_1 (including the empty string if this is the case). Thus, from g_1 we:

1. Construct g_2 such that $L(g_2) = L(g_1) - \varepsilon$;

2. Construct g_3 (using g_2) such that:

    - $L(g_3) = L(g_1) \cup \{\varepsilon\}$ if $\varepsilon \in L(g_1)$ or
    - $L(g_3) = L(g_1)$ if $\varepsilon \notin L(g_1)$.

Schematically, for step 1 we construct g_2 such that some rules are excluded from the original grammar (the empty rules), new rules are added (one directly, the others indirectly) and a new start symbol is introduced:

$$\begin{array}{ccc}
g_1 & \rightsquigarrow & g_2 \\
P_1 & \underbrace{\rightsquigarrow}_{\leq,\geq,+1} & P_2 \\
\Sigma & \underbrace{\rightsquigarrow}_{\textit{same}} & \Sigma \\
V_1 & \underbrace{\rightsquigarrow}_{+1} & V_2 \\
S_1 & \underbrace{\rightsquigarrow}_{\textit{new}} & S_2
\end{array}$$

The idea of a *nullable* (SUDKAMP, 2006, p. 107) symbol is represented by the definition `empty`:

```
Definition empty
(g: cfg terminal _) (s: non_terminal + terminal): Prop:=
derives g [s] [].
```

[4]The results of this section are available in libraries `emptyrules.v`, `unitrules.v`, `useless.v`, `inaccessible.v` and `simplification.v`.

Notation `sf'` represents a sentential form that is constructed with elements of types `non_terminal'` and `terminal`.

Function `symbol_lift` maps a pair of type `(non_terminal + terminal)` into a pair of the disjoint union `(non_terminal' + terminal)` by replacing each inhabitant of type `non_terminal` with the corresponding inhabitant of type `non_terminal'`:

```
Inductive non_terminal': Type:=
| Lift_nt: non_terminal → non_terminal'
| New_ss.

Notation sf' := (list (non_terminal' + terminal)).

Definition symbol_lift
(s: non_terminal + terminal): non_terminal' + terminal:=
match s with
| inr t ⇒ inr t
| inl n ⇒ inl (Lift_nt n)
end.
```

With these definitions, a new grammar `g_emp g` has been defined, such that the language generated by it matches the language generated by the original grammar (`g`), except for the empty string. Predicate `g_emp_rules` states that every non-empty rule of `g` is also a rule of `g_emp g`, and also adds new rules to `g_emp g` where every possible combination of nullable non-terminal symbols that appears on the right-hand side of a rule of `g` is removed, as long as the resulting right-hand side is not empty. Finally, it adds a rule that maps a new symbol, the start symbol of the new grammar (`New_ss`), to the start symbol of the original grammar. For this reason, the new type `non_terminal'` has been defined. The motivation for introducing a new start symbol at this point is to be able to prove that the start symbol does not appear in the right-hand side of any rule of the new grammar, a result that will be important in future developments.

```
Inductive g_emp_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal' → sf' → Prop :=
| Lift_direct :
    ∀ left: non_terminal,
    ∀ right: sf,
    right ≠ [] → rules g left right →
    g_emp_rules g (Lift_nt left) (map symbol_lift right)
| Lift_indirect:
    ∀ left: non_terminal,
    ∀ right: sf,
    g_emp_rules g (Lift_nt left) (map symbol_lift right)→
    ∀ s1 s2: sf,
    ∀ s: non_terminal,
    right = s1 ++(inl s) :: s2 →
    empty g (inl s) →
    s1 ++s2 ≠ [] →
    g_emp_rules g (Lift_nt left) (map symbol_lift (s1 ++s2))
| Lift_start_emp:
    g_emp_rules g New_ss [inl (Lift_nt (start_symbol g))].

Definition g_emp
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal' terminal :=
  {| start_symbol:= New_ss;
     rules:= g_emp_rules g;
     rules_finite:= g_emp_finite g |}.
```

To summarize, we relate below the transformations that create g_2 (or `g_emp`) from g_1 to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals (inductive definition `non_terminal'`):
  - All the non-terminals of g_1 (constructor `Lift_nt`);
  - A fresh new non-terminal symbol S_2 (constructor `New_ss`).
- For the new set of rules (inductive definition `g_emp_rules`):
  - All non-empty rules of g_1 (constructor `Lift_direct`);
  - All rules of g_1 with every combination on nullable symbols in the right-hand side removed, except if empty (constructor `Lift_indirect`);
  - One new rule: $S_2 \rightarrow S_1$ (constructor `Lift_start_emp`).
- For the new grammar (definition `g_emp`):
  - The new set of non-terminals (implicit);
  - The new set of rules (`g_emp_rules g`);
  - The new non-terminal as the start symbol (`New_ss` as S_2).

Suppose, for example, that X,A,B,C are non-terminals, of which A,B and C are nullable, a,b and c are terminals and $X \rightarrow aAbBcC$ is a rule of `g`. Then, the above definitions assert that $X \rightarrow aAbBcC$ is a rule of `g_emp g`, and also:

- $X \rightarrow aAbBc$;
- $X \rightarrow abBcC$;
- $X \rightarrow aAbcC$;
- $X \rightarrow aAbc$;
- $X \rightarrow abBc$;
- $X \rightarrow abcC$;
- $X \rightarrow abc$.

Observe that resulting grammar (`g_emp g` or g_2) does not generate the empty string, even if `g` (or g_1) does so. The second step, thus, consists of constructing g_3 such that it generates all the strings of g_2 plus the empty string if g_1 does so. This is done by conditionally adding a rule that maps the start symbol to the empty string. Schematically:

$$\begin{array}{ccc}
g_1 & \rightsquigarrow & g_3 \\
P_1 & \underbrace{\rightsquigarrow}_{same\ of\ g_2\ or\ +1} & P_3 \\
\Sigma & \underbrace{\rightsquigarrow}_{same\ of\ g_2} & \Sigma \\
V_1 & \underbrace{\rightsquigarrow}_{same\ of\ g_2} & V_3 \\
S_1 & \underbrace{\rightsquigarrow}_{same\ of\ g_2} & S_3
\end{array}$$

We define `g_emp' g` (or g_3) such that `g_emp' g` generates the empty string if `g` generates the empty string. This was done by stating that every rule from `g_emp g` is also a rule of `g_emp' g` and also by adding a new rule that allow `g_emp' g` to generate the empty string directly if necessary.

```
Inductive g_emp'_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal' non_terminal → sf' → Prop :=
| Lift_all:
   ∀ left: non_terminal' _,
   ∀ right: sf',
   rules (g_emp g) left right → g_emp'_rules g left right
| Lift_empty:
   empty g (inl (start_symbol g)) → g_emp'_rules g (start_symbol (g_emp g)) [].
```

```
Definition g_emp'
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: cfg (non_terminal' _) terminal :=
  {| start_symbol:= New_ss _;
     rules:= g_emp'_rules g;
     rules_finite:= g_emp'_finite g |}.
```

Note that the generation of the empty string by `g_emp' g` depends on `g` generating the empty string.

To summarize, we relate below the transformations that create g_3 (or `g_emp' g`) from g_1 (or `g`) to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals:
    - All the non-terminals of step 1.
- For the new set of rules (inductive definition `g_emp'_rules`):
    - All the rules of step 1 (constructor `Lift_all`);
    - One new rule: $S_2 \rightarrow \varepsilon$ if $\varepsilon \in L(g_1)$ (constructor `Lift_empty`).
- For the new grammar (definition `g_emp'`):
    - The same set of non-terminals;
    - The new set of rules (`g_emp'_rules g`);
    - The same start symbol (`New_ss` as S_2).

The proof of the correctness of the previous definitions is achieved through the following theorem:

```
Theorem g_emp'_correct:
∀ g: cfg non_terminal terminal,
g_equiv (g_emp' g) g ∧
(produces_empty g → has_one_empty_rule (g_emp' g)) ∧
(∼ produces_empty g → has_no_empty_rules (g_emp' g)) ∧
start_symbol_not_in_rhs (g_emp' g).
```

Three new predicates are used in this statement: `has_one_empty_rule`, to describe a grammar that has a single empty rule among its set of rules (one whose left-hand side is the initial symbol), `has_no_empty_rules` for a grammar that has no empty rules at all and `start_symbol_not_in_rhs` to state that the start symbol does not appear in the right-hand side of any rule of the argument grammar:

```
Definition has_one_empty_rule (g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right →
(( left = start_symbol g) ∧ (right = []) ∨ right ≠ []).

Definition has_no_empty_rules (g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right → right ≠ [].

Definition start_symbol_not_in_rhs (g: cfg non_terminal terminal):=
∀ left: non_terminal,
∀ right: sf,
rules g left right → ~ In (inl (start_symbol g)) right.
```

The definition of `g_equiv`, when applied to this theorem, yields:

```
∀ s: sentence,
produces (g_emp' g) s ↔ produces g s.
```

For the `produces (g_emp' g) s` → `produces g s` part of the statement, the strategy used is to prove that, for every rule $left \rightarrow_{g\_emp'g} right$ of `g_emp' g`, either $left \rightarrow_g right$ is a rule of `g` or $left \Rightarrow_g^* right$. Thus, any derivation in `g_emp' g` can be translated into a derivation in `g`. The proof is as follows:

- Case analysis shows $right = \varepsilon$ (s=ε):
  - We know that `produces (g_emp' g)` ε;
    (by simple substitution)
  - We know that the empty rule of `g_emp'` must have been used in this derivation;
    (constructor `Lift_empty`)
  - We know that `produces g` ε.
    (case analysis on `g_emp'_rules`)
- Case analysis shows $right \neq \varepsilon$ (s$\neq \varepsilon$):
  - We can substitute `g_emp'` for `g_emp` in the hypothesis;
    (using lemma `g_emp_equiv_g_emp'`, since $right \neq \varepsilon$)
  - We know that if $left \rightarrow_{g\_emp} right$, either $left \rightarrow_g right$ is a rule of `g` or $left \Rightarrow_g^* right$; (from lemma `derives_g_emp_g`, which is proved by induction on predicate `derives`)

  - In either situation, the proof if straightforward.

For the `produces (g_emp' g) s` $\leftarrow$ `produces g s` part of the statement, the strategy is similar to the previous one, and relies on proving that any non empty string that can be derived in `g` can also be derived in `g_emp g`:

- Case analysis shows *right* $= \varepsilon$ (`s=`ε):
  - We know that `produces g` ε;
    (by simple substitution)
  - We know that `g_emp'` has an empty rule;
    (constructor `Lift_empty`)
  - We know that `produces (g_emp' g)` ε.
    (application of `Lift_empty`)
- Case analysis shows *right* $\neq \varepsilon$ (`s`$\neq \varepsilon$):
  - Substitute `g` for `g_emp` in the hypothesis, which then becomes $S_1 \Rightarrow^*_{g_2} s_1$;
    (using lemma `derives_g_g_emp`, which states that every sentence generated by `g` is also generated by `g_emp`, with the exception ε, and since *right* $\neq \varepsilon$. This lemma is proved by induction on the number of derivations in `g`, after substituting predicate `derives` by `derives6`, see Section 6.3)
  - Substitute `g_emp'` by `g_emp` in the goal, which then becomes $S_2 \Rightarrow^*_{g_2} s_2$;
    (using lemma `g_emp_equiv_g_emp'`, since *right* $\neq \varepsilon$)
  - The new goal now is $S_1 \Rightarrow^*_{g_2} s_1 \rightarrow S_2 \Rightarrow^*_{g_2} s_1$;
  - Substitute $S_2 \Rightarrow^*_{g_2} s_1$ by subgoals $S_2 \Rightarrow^*_{g_2} S_1$ and $S_1 \Rightarrow^*_{g_2} s_1$;
    (using lemma `derives_trans`)
  - The second subgoal is trivial;
    (use the hypothesis)
  - For the first subgoal, use the definition of `g_emp_rules`;
    (constructor `Lift_start_emp`)

For the `produces_empty g` $\rightarrow$ `has_one_empty_rule (g_emp' g)` part, the proof follows from the definition of `g_emp'_rules`: constructor `Lift_empty` adds an empty rule to the grammar whenever `g` produces the empty string.

For the `~produces_empty g` $\rightarrow$ `has_no_empty_rules (g_emp' g)` part, it is enough to observe that `g_emp'` can not have an empty rule when `g` does not generate the empty string, since the rules of `g_emp` (constructor `Lift_all`) are all non-empty by construction.

Finally, the `start_symbol_not_in_rhs (g_emp' g)` part of the statement is a direct consequence of the fact that (i) a new start symbol (`New_ss`) has been added to `g_emp`; (ii) this symbol is also the start symbol of `g_emp'` and (iii) no rules have been added to `g_emp` or `g_emp'` that use this symbol in the right-hand side of any rule.

As it has been shown, the proof of `g_emp'_correct` was reduced to the proof of the equivalence of grammars `g` and `g_emp g`. The most difficult part of this formalization, by far, was to prove this equivalence, as expressed by lemmas `derives_g_g_emp` and `derives_g_emp_g`. While the second case was relatively straightforward, the first proved much more difficult. This happens because the application of a rule of `g` can cause a non-terminal symbol to be eliminated from the sentential form (if it is an empty rule), and for this reason we have to introduce a new structure and do many case analysis in the sentential form of `g` in order to determine the corresponding new rule of `g_emp g` that has to be used in the derivation. We believe that the root of this difficulty was the desire to follow strictly the informal proof of Sudkamp (2006, Theorem 5.1.5) which depends on an intuitive lemma (lemma 3.1.5), however not easily formalizable. Probably for this reason, the solution constructed in our formalization is definitely not easy or readable, and this motivates the continued search for a more simple and elegant one.

6.6.2 Unit rules

Result (2) of Section 6.6 is achieved in only one step. We first define the relation `unit` such that, for any two non-terminal symbols X and Y, `unit X Y` is true when $X \Rightarrow^{+} Y$ (SUDKAMP, 2006, p. 114). This means that Y can be derived from X by the exclusive use of one or more unit rules.

The mapping of grammar g_1 into an equivalent grammar g_2 such that g_2 is free of unit rules consists basically of keeping all non-unit rules of g_1 and creating new rules that reproduce the effect of the unit rules that were left behind. No new non-terminal symbols are necessary. Schematically:

$$\begin{array}{ccc}
g_1 & \rightsquigarrow & g_2 \\
P_1 & \underbrace{\rightsquigarrow}_{\leq,\geq} & P_2 \\
\Sigma & \underbrace{\rightsquigarrow}_{\mathit{same}} & \Sigma \\
V_1 & \underbrace{\rightsquigarrow}_{\mathit{same}} & V_2 \\
S_1 & \underbrace{\rightsquigarrow}_{\mathit{same}} & S_2
\end{array}$$

Definition `unit` is formalized as:

```
Inductive unit
(terminal non_terminal : Type)
(g: cfg terminal non_terminal)
(a: non_terminal)
: non_terminal → Prop:=
| unit_rule:
   ∀ (b: non_terminal),
   rules g a [inl b] → unit g a b
| unit_trans:
   ∀ b c: non_terminal,
   unit g a b → unit g b c → unit g a c.
```

Grammar `g_unit g` (presented next) represents the grammar that is equivalent to `g`, except that the unit rules of the latter have been substituted by others, non-unit rules, that produce the same results in terms of the generated language. The idea is that `g_unit g` has all non-unit rules of `g`, plus new rules that are created by anticipating the possible application of unit rules in `g`, as informed by `g_unit`.

```
Inductive g_unit_rules
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal → sf → Prop :=
| Lift_direct' :
   ∀ left: non_terminal,
   ∀ right: sf,
   (∀ r: non_terminal, right ≠ [inl r]) →
   rules g left right →
   g_unit_rules g left right
| Lift_indirect':
   ∀ a b: non_terminal,
   unit g a b →
   ∀ right: sf,
   rules g b right →
   (∀ c: non_terminal, right ≠ [inl c]) →
   g_unit_rules g a right.

Definition g_unit
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal terminal :=
  {| start_symbol:= start_symbol g;
     rules:= g_unit_rules g;
     rules_finite:= g_unit_finite g |}.
```

To summarize, we relate below the transformations that create g_2 (or `g_unit g`) from g_1 (or `g`) to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals:
  - All the non-terminals of g_1.
- For the new set of rules (inductive definition `g_unit_rules`):
  - All non-unit rules of g_1 (constructor `Lift_direct'`);
  - New rules: one for each `a`, `b`, *right* such that (i) `unit a b`, (ii) $b \rightarrow right$, (iii) *right* is not a single non-terminal; the new rule becomes $a \rightarrow right$ (constructor `Lift_indirect'`).
- For the new grammar (`g_unit`):
  - The same set of non-terminals;
  - The new set of rules (`g_unit_rules g`);
  - The same start symbol (S_1).

As an example, consider the grammar $G = (\{S,X,Y,Z,a,b,c\},\{a,b,c\},P,S)$, with P containing the following rules:

$$\begin{array}{rcl} S & \rightarrow & X \mid ab \\ X & \rightarrow & Y \mid bc \\ Y & \rightarrow & Z \mid ac \\ Z & \rightarrow & abc \end{array}$$

The above definitions assert that the new grammar G' (the grammar that is equivalent to G and is free of unit rules) has the following rules:

$$\begin{array}{rcl} S & \rightarrow & abc \mid ac \mid bc \mid ab \\ X & \rightarrow & abc \mid ac \mid bc \\ Y & \rightarrow & abc \mid ac \\ Z & \rightarrow & abc \end{array}$$

Finally, the correctness of `g_unit` comes from the following theorem:

```
Theorem g_unit_correct:
∀ g: cfg non_terminal terminal,
g_equiv (g_unit g) g ∧ has_no_unit_rules (g_unit g).
```

The predicate `has_no_unit_rules` states that the argument grammar has no unit rules at all:

```
Definition has_no_unit_rules (g: cfg non_terminal terminal): Prop:=
∀ left n: non_terminal,
∀ right: sf,
rules g left right → right ≠ [inl n].
```

Similar to the proof of `g_emp'_correct`, for the `produces (g_unit g) s → produces g s` part of the statement (obtained from `g_equiv (g_unit g) g`), the strategy used here is to prove that, for every rule $left \rightarrow_{g\_unit\ g} right$ of `g_unit g`, either $left \rightarrow_g right$ is a rule of `g` or $left \Rightarrow_g^* right$. Thus, any derivation in `g_unit g` can be translated into a derivation in `g`. The proof is as follows, and corresponds to lemma `generates_g_unit_g`:

- Induction on predicate `derives` in the hypothesis:
    - Base case: trivial
      (use constructor `derives_refl`)
    - Induction case:
        - Substitute $left \Rightarrow_{g\_unit g}^* right$ in the hypothesis by $left \rightarrow_g right \vee left \Rightarrow_g^* right$;
          (use lemma `rules_g_unit_g`, which is proved by case analysis on `g_unit_rules`)
        - Case analysis on the disjunction:
            - $left \rightarrow_g right$: trivial;
              (use hypotheses and constructor `derives_step` to obtain the derivation)
            - $left \Rightarrow_g^* right$: trivial;
              (use hypotheses and lemma `derives_subs` to obtain the derivation)

For the `produces (g_unit g) s ← produces g s` part of the statement, the strategy involves induction over predicate `derives3` that is equivalent to `derives` but generates the sentence directly from a non-terminal symbol (see Section 6.3). Here is a sketch of the proof, that corresponds to lemmas `derives_g_g_unit` and `derives3_g_g_unit`:

- Substitute predicate `derives` by predicate `derives3` both in the hypothesis and the goal;
  (use lemma `derives_equiv_derives3`)

- Apply a previously created specialized induction principle to the hypothesis; this generates 6 subgoals;
  (`derives3_ind_2`, since `derives3` is a mutual inductive definition)

- Most subgoals are trivial and can be proved directly or with the help of the constructors of `derives3` and `derives3_aux`;

- One of the subgoals, however, is longer and more complex and its proof demands the additional use of the constructors of `unit` and the following lemmas:

  - `rules_g_g_unit`, to prove that every non-unit rule of `g` is also a rule of `g_unit g`;
  - `exists_rule_derives3_aux`, to prove that the derivation of a sentence is obtained exclusively by a single rule or the combination of a rule and a derivation;
  - `rules_g_unit_g'`, to prove that a rule in `g_unit g` with only terminal symbols in the right-hand side implies the existence of a single rule in `g` or the combination of a `unit` relation and a derivation in `g`;
  - `rules_g_unit_not_unit`, to prove that no rule of `g_unit g` can not have a single non-terminal symbol in the right-hand side.

The proof of the `has_no_unit_rules (g_unit g)` part of `g_unit_correct` is simple and can be obtained by case analysis on the right-hand side of the rules of `g_unit g` and also on the definition of `g_unit_rules` (lemma `g_unit_has_no_unit_rules`).

We find important similarities in the proofs of grammar equivalence for the elimination of empty rules (lemma `g_emp'_correct`) and the elimination of unit rules (lemma `g_unit_correct`). In both cases, going backwards (from the new to the original grammar) was relatively straightforward and required no special machinery. On the other hand, going forward (from the original to the new grammar) proved much more difficult and required new definitions, functions and lemmas in order to complete the corresponding proofs.

In the case of the elimination of unit rules, for example, the proof that every sentence generated by the original grammar is also generated by the transformed grammar required the introduction of the `derives3` predicate specially for this purpose. Because this definition represents the derivation of sentences directly from a non-terminal symbol, it was possible to abstract over the use of unit rules. Since `derives3` is a mutual inductive definition, we had to create a specialized induction principle (`derives3_ind_2`) and use it explicitly, which resulted in the increased complexity of the proofs.

6.6.3 Useless symbols

Result (3) of Section 6.6 is obtained in a single and simple step, which consists of inspecting all rules of grammar g_1 and eliminating the ones that contain useless symbols in either the left or right-hand side. The other rules are kept in the new grammar g_2. Thus, $P_2 \subseteq P_1$. No new non-terminals are required. Schematically:

$$\begin{array}{ccc} g_1 & \rightsquigarrow & g_2 \\ P_1 & \underbrace{\rightsquigarrow}_{\leq} & P_2 \\ \Sigma & \underbrace{\rightsquigarrow}_{\textit{same}} & \Sigma \\ V_1 & \underbrace{\rightsquigarrow}_{\textit{same}} & V_2 \\ S_1 & \underbrace{\rightsquigarrow}_{\textit{same}} & S_2 \end{array}$$

The idea of a useful symbol is captured by the definition `useful`:

```
Definition useful
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
(s: non_terminal + terminal): Prop:=
match s with
| inr t ⇒ True
| inl n ⇒ ∃ s: sentence, derives g [inl n] (map term_lift s)
end.
```

The removal of useless symbols comprises, first, the identification of useless symbols in the grammar and, second, the elimination of the rules that use them. Definition `g_use_rules` selects, from the original grammar, only the rules that do not contain useless symbols. The new grammar, without useless symbols, can then be defined as in `g_use`:

```
Inductive g_use_rules
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal → sf → Prop :=
| Lift_use :
   ∀ left: non_terminal,
   ∀ right: sf,
   rules g left right →
   useful g (inl left) →
   (∀ s: non_terminal + terminal, In s right → useful g s) →
   g_use_rules g left right.
```

```
Definition g_use
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal terminal:=
  {| start_symbol:= start_symbol g;
     rules:= g_use_rules g;
     rules_finite:= g_use_finite g |}.
```

To summarize, we relate below the transformations that create g_2 (or `g_use g`) from g_1 (or `g`) to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals:
  - All the non-terminals of g_1.
- For the new set of rules (inductive definition `g_use_rules`):
  - All rules of g_1, except those that have useless symbols (constructor `Lift_use`).
- For the new grammar (definition `g_use`):
  - The same set of non-terminals;
  - The new set of rules (`g_use_rules g`);
  - The same start symbol (S_1, which must be useful).

As an example, consider grammar $G = (\{X,Y,Z,a,b,c\},\{a,b,c\},P,S)$, with P containing the following rules:

$$\begin{array}{rcl} S & \rightarrow & Xa \mid Ya \mid Za \\ X & \rightarrow & aX \mid bY \\ Y & \rightarrow & aY \mid bX \\ Z & \rightarrow & bZ \mid c \end{array}$$

Clearly, symbols X and Y are useless symbols and can thus be removed from G, resulting in G' with the following set of rules:

$$\begin{array}{rcl} S & \rightarrow & Za \\ Z & \rightarrow & bZ \mid c \end{array}$$

The `g_use` definition, of course, can only be used if the language generated by the original grammar is not empty, that is, if the start symbol of the original grammar is useful. If it were useless then it would be impossible to assign a root to the grammar and the language would be empty. The correctness of the useless symbol elimination operation can be certified by proving theorem `g_use_correct`, which states that every context-free grammar whose start symbol is useful generates a language that can also be generated by an equivalent context-free grammar whose symbols are all useful.

```
Theorem g_use_correct:
∀ g: cfg non_terminal terminal,
non_empty g → g_equiv (g_use g) g ∧ has_no_useless_symbols (g_use g).
```

The predicates `non_empty`, and `has_no_useless_symbols` used above assert, respectively, that grammar `g` generates a language that contains at least one string (which in turn may or may not be empty) and the grammar has no useless symbols at all:

```
Definition non_empty (g: cfg non_terminal terminal): Prop:=
useful g (inl (start_symbol g)).

Definition has_no_useless_symbols (g: cfg non_terminal terminal): Prop:=
∀ n: non_terminal, appears g (inl n) → useful g (inl n).
```

Hypothesis `non_empty g` on lemma `g_use_correct` is necessary in order to assure that the new grammar will have a start symbol (the start symbol should be a useful symbol, otherwise it would not be possible to obtain a new grammar free of useless symbols).

For the proof of the `g_equiv` part of the statement of `g_use_correct`, we first split it in two parts:

- `produces (g_use g) s` → `produces g s`, and
- `produces g s` → `produces (g_use g) s`.

In what follows, the predicate `sflist` (see Section 6.2.2) is used in order to support the construction of these proofs.

The proof of the first part is relatively straightforward, since every rule of `g_use` is also a rule of `g`. It corresponds to the statement of lemma `produces_g_use_g`:

- Substitute predicate `derives` by `sflist` both in the hypothesis and the conclusion;
  (use lemma `derives_sflist`, which states that every derivation (`derives`) is in relation to a corresponding `sflist`)
- Use, for the goal, the same list of derivations of the hypothesis; we now have to prove that the list of derivations in `g_use g` is also a valid list of derivations in `g`;
- Substitute the goal `sflist g l` by `sflist (g_use g) l`;
  (use lemma `sflist_g_use_g`, which is proved by simple induction on `sflist`)

- The goal matches the hypothesis.

For the converse, it is necessary to show that every symbol used in a derivation in `g` is useful, and thus the rules used in this derivation also appear in `g_use g`. It corresponds to the statement of lemma `produces_g_g_use`:

- Substitute predicate `derives` by `sflist` both in the hypothesis and the conclusion;
  (use lemma `derives_sflist`)

- Use, for the goal, the same list of derivations of the hypothesis; we now have to prove that the list of derivations in `g` is also a valid list of derivations in `g_use g`;

- Substitute the goal `sflist (g_use g) l` by `sflist g l`;
  (use lemma `sflist_g_g_use`, which is proved first by case analysis on the length of the list and then by showing that every rule of `g` used in this list is made of useful symbols (since the last sentential form in the list is a sentence), and thus the same rule is also part of `g_use g`)

- The goal matches the hypothesis.

For the proof of the `has_no_useless_symbols (g_use g)` part of the statement of `g_use_correct`, it is enough to do case analysis in `g_use_rules` to conclude that all symbols of the rules of `g_use g` are useful, and thus have no useless symbols (by means of lemma `useful_g_g_use`, which in turn is proved with the aid of predicate `sflist`).

The formalization of the results of this section is not long and is quite natural when compared to the arguments of the informal proof. The only exception are lemmas `sflist_g_g_use` and `useful_g_g_use` (below), whose statements are simple and intuitive but whose proofs are relatively long (~150 lines each) and filled with small details. Specifically for the proofs of this and the next section, it was necessary to define the predicate `sflist` and to use it in place of `derives`.

```
Lemma sflist_g_g_use:
∀ g: cfg _ _,
∀ l: list sf,
∀ s: sentence,
sflist g l ∧ last l [] = map term_lift s → sflist (g_use g) l.

Lemma useful_g_g_use:
∀ g: cfg _ _,
∀ n: non_terminal,
useful g (inl n) → useful (g_use g) (inl n).
```

6.6.4 Inaccessible symbols

Result (4) of Section 6.6 is similar to the previous case: the rules of the original grammar g_1 are kept in the new grammar g_2 as long as their left-hand consist of accessible non-terminal symbols (by definition, if the left-hand side is accessible then all the symbols in the right-hand side of the same rule are also accessible). If this is not the case, then the rules are left behind. Thus, $P_2 \subseteq P_1$. Schematically:

$$\begin{array}{ccc}
g_1 & \rightsquigarrow & g_2 \\
P_1 & \underbrace{\rightsquigarrow}_{\leq} & P_2 \\
\Sigma & \underbrace{\rightsquigarrow}_{same} & \Sigma \\
V_1 & \underbrace{\rightsquigarrow}_{same} & V_2 \\
S_1 & \underbrace{\rightsquigarrow}_{same} & S_2
\end{array}$$

Definition `accessible` is used to represent accessible symbols in context-free grammars.

```
Definition accessible
(terminal non_terminal : Type)
(g : cfg non_terminal terminal)
(s: non_terminal + terminal): Prop:=
∃ s1 s2: sf, derives g [inl (start_symbol g)] (s1++s::s2).
```

Definition `g_acc_rules` selects, from the original grammar, only the rules that do not contain inaccessible symbols. Definition `g_acc` represents a grammar whose inaccessible symbols have been removed:

```
Inductive g_acc_rules
(terminal non_terminal : Type)
(g : cfg non_terminal terminal)
: non_terminal → sf → Prop :=
| Lift_acc : ∀ left: non_terminal,
   ∀ right: sf,
   rules g left right → accessible g (inl left) → g_acc_rules g left right.

Definition g_acc
(terminal non_terminal : Type)
(g : cfg non_terminal terminal)
: cfg non_terminal terminal :=
  {| start_symbol:= start_symbol g;
     rules:= g_acc_rules g;
rules_finite:= g_acc_finite g |}.
```

To summarize, we relate below the transformations that create g_2 (or `g_acc g`) from g_1 (or `g`) to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals:
    - All the non-terminals of g_1.
- For the new set of rules (inductive definition `g_acc_rules`):
    - All rules of g_1, except those that have inaccessible symbols (constructor `Lift_acc`).
- For the new grammar (definition `g_acc`):
    - The same set of non-terminals;
    - The new set of rules (`g_ac_rules g`);
    - The same start symbol (S_1).

As an example, consider grammar $G = (\{X,Y,Z,a,b,c\},\{a,b,c\},P,S)$, with P containing the following rules:

$$\begin{array}{rcl} S & \rightarrow & aX \mid bX \\ X & \rightarrow & aX \mid bX \mid a \mid b \\ Y & \rightarrow & cZ \mid a \\ Z & \rightarrow & cZ \mid b \end{array}$$

Clearly, symbols Y, Z and c are inaccessible symbols and can thus be removed from G, resulting in G' with the following set of rules:

$$\begin{array}{rcl} S & \rightarrow & aX \mid bX \\ X & \rightarrow & aX \mid bX \mid a \mid b \end{array}$$

The correctness of the inaccessible symbol elimination operation can be certified by proving theorem `g_acc_correct`, which states that every context-free grammar generates a language that can also be generated by an equivalent context-free grammar whose symbols are all accessible.

```
Theorem g_acc_correct:
∀ g: cfg non_terminal terminal,
g_equiv (g_acc g) g ∧ has_no_inaccessible_symbols (g_acc g).
```

In a way similar to `has_no_useless_symbols`, the absence of inaccessible symbols in a grammar is expressed by predicate `has_no_inaccessible_symbols` used above and defined as:

```
Definition has_no_inaccessible_symbols (g: cfg non_terminal terminal):
Prop:=
∀ s: (non_terminal + terminal), appears g s → accessible g s.
```

The proof of the first part of the `g_equiv (g_acc g) g` statement (`produces (g_acc g) s → produces g s`) is straightforward, since every rule of `g_acc` is also a rule of `g`. It corresponds to the statement of lemma `produces_g_acc_g`:

- Substitute predicate `derives` by `sflist` both in the hypothesis and the conclusion;
  (use lemma `derives_sflist`)
- Use, for the goal, the same list of derivations of the hypothesis; we now have to prove that the list of derivations in `g_acc g` is also a valid list of derivations in `g`;
- Substitute the goal `sflist g l` by `sflist (g_acc g) l`;
  (use lemma `sflist_g_acc_g`, which is proved by simple induction on `sflist`)
- The goal matches the hypothesis.

For the converse (`produces g s → produces (g_acc g) s`), it is necessary to show that every symbol used in a derivation in `g` is accessible, and thus the rules used in this derivation also appear in `g_acc g`. It corresponds to lemma `produces_g_g_acc`:

- Substitute predicate `derives` by `sflist` both in the hypothesis and the conclusion;
  (use lemma `derives_sflist`)
- Use, for the goal, the same list of derivations of the hypothesis; we now have to prove that the list of derivations in `g` is also a valid list of derivations in `g_acc g`;
- Substitute the goal `sflist (g_acc g) l` by `sflist g l`;
  (use lemma `sflist_g_g_acc`, which is proved first by case analysis on the length of the list and then by showing that every rule of `g` used in this list is made of accessible symbols, and thus the same rule is also part of `g_acc g`)
- The goal matches the hypothesis.

For the proof of the `has_no_inaccessible_symbols (g_acc g)` part of the statement of `g_acc_correct`, it is necessary to do case analysis in the symbol `s` used in the definition. For the non-terminal case, it is enough to see that it may occur either in the left or the right-hand side of the rules of `g_acc g`. If it occurs in the left-hand side, a case analysis on `g_acc_rules` reveals (i) the original rule of `g` and (ii) that the non-terminal symbol is accessible in `g`. Finally, lemma `accessible_g_g_acc` proves that the non-terminal is accessible in `g_acc g` as well. If it occurs in the right-hand side, we use an identical procedure and conclude (i) that the non-terminal in the left-hand side of the rule where it appears is accessible in `g`; and also (ii) that our non-terminal appears in the right-hand side of the same rule. To prove that the non-terminal in the right-hand side is also accessible in `g_acc g`, we simply use lemma `acc_step` (this lemma states that if the left-hand side is accessible, so are all the symbols in the right-hand side of the same rule, and is a direct consequence of the definition of accessibility). For the terminal case, the symbol must necessarily appear in the right-hand side of some rule, and thus the previous steps apply here as well.

Lemma `accessible_g_g_acc` (below) is used to assert that every accessible symbol of `g` is also accessible in `g_acc g`. Its prove uses the predicate `sflist` in the place of predicate `derives`, as explained in other situations.

```
Lemma accessible_g_g_acc:
∀ g: cfg _ _,
∀ s: non_terminal + terminal,
accessible g s → accessible (g_acc g) s.
```

The formalization of the results of this section is also natural when compared to the arguments of the informal proof. It has only 384 lines on Coq script and, despite the similarities between the formalization of this section and those of the previous section (elimination of useless symbols), it is still ~40% shorter than that. This is partially due to a difference in the definitions of `g_use_rules` and `g_acc_rules`: in the first case, in order to be ellegible as a rule of `g_use`, a rule of `g` must provably consist only of useful symbols in both the left and right-hand sides; in the second, it is enough to prove that only the left-hand side is accessible (the rest is consequence of the definition). Since we have a few uses of the constructors of these definitions, the more simple definition of `g_acc_rules` resulted in simpler and shorter proofs. As a matter of fact, it should be possible to do something similar to the definition of `g_use_rules`, since the left-hand side of a rule is automatically useful once all the symbols in the right-hand side are proved useful (also a consequence of the definition). This will be considered in a future review of the formalization.

6.6.5 Unification

So far we have only considered each simplification strategy independently of the others. If one wants to obtain a new grammar that is simultaneously free of empty and unit rules, and of useless and inaccessible symbols, it is not enough to consider the previous independent results: it is necessary to establish a suitable order to apply these simplifications, in order to guarantee that the final result satisfies all desired conditions. Then, it is necessary to prove that the claims do hold.

For the order, we should start with (i) the elimination of empty rules, followed by (ii) the elimination of unit rules. The reason for this is that (i) might introduce new unit rules in the grammar, and (ii) will surely not introduce empty rules, as long as the original grammar is free of them (except for $S \rightarrow \varepsilon$, in which case S, the initial symbol of the grammar, must not appear on the right-hand side of any rule). Then, elimination of useless and inaccessible symbols (in either order) is the right thing to do, since they only remove rules from the original grammar (which is specially important because they do not introduce new empty or unit rules).

The formalization of this result is captured in the following theorem, which represents the main result of this section:

```
Theorem g_simpl_ex_v1:
∀ g: cfg non_terminal terminal,
 non_empty g →
 ∃ g': cfg (non_terminal' non_terminal) terminal,
 g_equiv g' g ∧
 has_no_inaccessible_symbols g' ∧
 has_no_useless_symbols g' ∧
(produces_empty g → has_one_empty_rule g') ∧
(~ produces_empty g → has_no_empty_rules g') ∧
 has_no_unit_rules g' ∧
 start_symbol_not_in_rhs g'.
```

The proof of `g_simpl_ex_v1` demands auxiliary lemmas to prove that the characteristics of the initial transformations are preserved by the following ones. For example, that all of the unit rules elimination, useless symbol elimination and inaccessible symbol elimination operations preserve the characteristics of the empty rules elimination operation. Figure 6.1 summarizes the situation.

6.7 Chomsky Normal Form

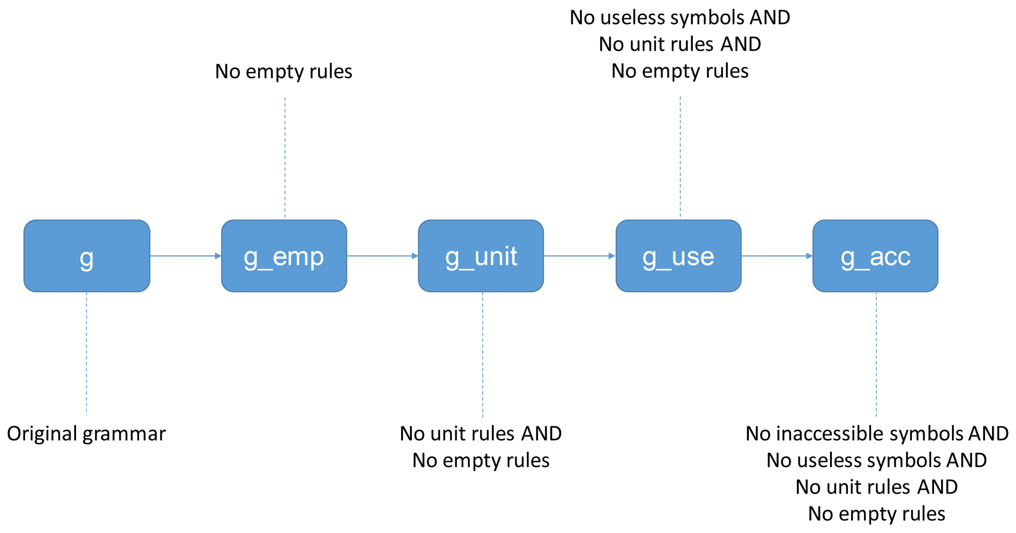
The Chomsky Normal Form (CNF) theorem, proposed and proved by Chomsky in Chomsky (1959), asserts:

$$\forall\, G = (V, \Sigma, P, S),\ \exists\, G' = (V', \Sigma, P', S') \mid$$

$$L(G) = L(G') \wedge \forall\, (\alpha \rightarrow_{G'} \beta) \in P', (\beta \in \Sigma) \vee (\beta \in N \cdot N)$$

That is, every context-free grammar can be converted to an equivalent one whose rules have only one terminal symbol or two non-terminal symbols in the right-hand side. Naturally, this is valid only if G does not generate the empty string. If this is the case, then the grammar that has this format, plus a single rule $S' \rightarrow_G \varepsilon$, is also considered to be in the Chomsky Normal

Figure 6.1: Simplification sequence



Source: the author

Form, and generates the original language, including the empty string. It can also be assured that in either case the start symbol of G' does not appear on the right-hand side of any rule of G'.

The existence of a CNF can be used for a variety of purposes, including to prove that there is an algorithm to decide whether an arbitrary context-free language accepts an arbitrary string, and to test if a language is not context-free (using the Pumping Lemma for context-free languages, which can be proved with the help of CNF grammars).

The idea of mapping G into G' consists of creating a finite number of new non-terminal symbols and new rules, in the following way:

1. For every terminal symbol σ that appears in the right-hand side of a rule $r = \alpha \rightarrow_G \beta_1 \cdot \sigma \cdot \beta_2$ of G, create a new non-terminal symbol $[\sigma]$, a new rule $[\sigma] \rightarrow_{G'} \sigma$ and substitute σ for $[\sigma]$ in r;

2. For every rule $r = \alpha \rightarrow_G N_1 N_2 \cdots N_k$ of G, where N_i are all non-terminals, create a new set of non-terminals and a new set of rules such that:

$$\begin{array}{rcl}
\alpha & \rightarrow_{G'} & N_1[N_2 \cdots N_k], \\
{[N_2 \cdots N_k]} & \rightarrow_{G'} & N_2[N_3 \cdots N_k], \\
 & \cdots & \\
{[N_{k-2} N_{k-1} N_k]} & \rightarrow_{G'} & N_{k-2}[N_{k-1} N_k], \\
{[N_{k-1} N_k]} & \rightarrow_{G'} & N_{k-1} N_k
\end{array}$$

Case (1) substitutes all terminal symbols of the grammar for newly created non-terminal symbols. Case (2) splits rules that have three or more non-terminal symbols on the right-hand side by a set of rules that have only two non-terminal symbols in the right-and side. Both changes preserve the language of the original grammar.

As an example, consider $G = (\{S', X, Y, Z, a, b, c\}, \{a, b, c\}, P, S')$ with P equal to:

$$\begin{aligned} \{S' &\rightarrow XYZd, \\ X &\rightarrow a, \\ Y &\rightarrow b, \\ Z &\rightarrow c, \} \end{aligned}$$

The CNF grammar G', equivalent to G, would then be the one with the following set of rules:

$$\begin{aligned} \{S' &\rightarrow X[YZd], \\ [YZd] &\rightarrow Y[Zd], \\ [Zd] &\rightarrow Z[d], \\ [d] &\rightarrow d, \\ X &\rightarrow a, \\ Y &\rightarrow b, \\ Z &\rightarrow c, \} \end{aligned}$$

It is clear from above that the original grammar must be free of empty and unit rules in order to be converted to a CNF equivalent. Also, it is desirable that the original grammar contains no useless and no inaccessible symbols, besides assuring that the start symbol does not appear on the right-hand side of any rule. Thus, it will be required that the original grammar be first simplified according to the results of Section 6.6.

Given the original grammar g_1, we will construct two new grammars g_2 and g_3. This first generates the same set of sentences of g_1, except for the empty string, and the second includes the empty string:

1. Construct g_2 such that $L(g_2) = L(g_1) - \varepsilon$;

2. Construct g_3 (using g_2) such that $L(g_3) = L(g_2) \cup \{\varepsilon\}$.

Then, either g_2 or g_3 will be used to prove the existence of a CNF grammar equivalent to g_1.

For step 1, the construction of g_2 is more complex, as we need to substitute terminals for new non-terminals, introduce new rules for these non-terminals and also split the rules with three or more smbols on the right-hand side. Schematically:

$$\begin{array}{ccc} g_1 & \rightsquigarrow & g_2 \\ P_1 & \underbrace{\rightsquigarrow}_{various} & P_2 \\ \Sigma & \underbrace{\rightsquigarrow}_{same} & \Sigma \\ V_1 & \underbrace{\rightsquigarrow}_{various} & V_2 \\ S_1 & \underbrace{\rightsquigarrow}_{new} & S_2 \end{array}$$

These ideas are captured by the following definitions[5]. The non-terminals of the new grammar `g_cnf g` are represented by the type `non_terminal'`. Its elements are associated with sentential forms of `g` via the constructor `Lift_r`:

```
Inductive non_terminal' (non_terminal terminal : Type): Type:=
| Lift_r: sf → non_terminal'.


Notation sf':= (list (non_terminal' + terminal)).
Notation term_lift:= ((terminal_lift non_terminal) terminal).
```

The function `symbol_lift`, presented below, maps sentential forms of `g` into sentential forms of `g_cnf g`:

```
Definition symbol_lift (s: non_terminal + terminal)
: non_terminal' + terminal:=
match s with
| inr t ⇒ inr t
| inl n ⇒ inl (Lift_r [inl n])
end.
```

The rules of `g_cnf g` and `g_cnf g` itself are defined as:

```
Inductive g_cnf_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal' → sf' → Prop:=
| Lift_cnf_t:
   ∀ t: terminal,
   ∀ left: non_terminal,
   ∀ s1 s2: sf,
   rules g left (s1++[inr t]++s2) →
   g_cnf_rules g (Lift_r [inr t]) [inr t]
| Lift_cnf_1:
```

[5]The results of this section are available in library `chomsky.v`.

```
  ∀ left: non_terminal,
  ∀ t: terminal,
  rules g left [inr t] →
  g_cnf_rules g (Lift_r [inl left]) [inr t]
| Lift_cnf_2:
  ∀ left: non_terminal,
  ∀ s1 s2: symbol,
  ∀ beta: sf,
  rules g left (s1 :: s2 :: beta) →
  g_cnf_rules g (Lift_r [inl left])
  [inl (Lift_r [s1]); inl (Lift_r (s2 :: beta))]
| Lift_cnf_3:
  ∀ left: sf,
  ∀ s1 s2 s3: symbol,
  ∀ beta: sf,
  g_cnf_rules g (Lift_r left)
  [inl (Lift_r [s1]); inl (Lift_r (s2 :: s3 :: beta))] →
  g_cnf_rules g (Lift_r (s2 :: s3 :: beta))
  [inl (Lift_r [s2]); inl (Lift_r (s3 :: beta))].

Definition g_cnf
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal' terminal :=
  {| start_symbol:= Lift_r [inl (start_symbol g)];
     rules:= g_cnf_rules g;
     rules_finite:= g_cnf_finite g |}.
```

To summarize, we relate below the transformations that create g_2 (or `g_emp g`) from g_1 (or `g`) to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals (inductive definition `non_terminal'`):
    - One for every possible sequence of terminal and non-terminal symbols of g_1 (constructor `Lift_r`).
- For the new set of rules (inductive definition `g_cnf_rules`):
    - One for every rule $left \rightarrow \alpha t \beta$ of g_1 (constructor `Lift_cnf_t`);
    - One for every rule $left \rightarrow t$ of g_1 (constructor `Lift_cnf_1`);
    - One for every rule $left \rightarrow s_1 s_2 \beta$ of g_1 (constructor `Lift_cnf_2`);
    - One for every rule $[left] \rightarrow [s_1][s_2 s_3 \beta]$ of g_2 (constructor `Lift_cnf_3`).

- For the new grammar (definition `g_cnf g`):
  - The new set of non-terminals (`non_terminal'`);
  - The new set of rules (`g_cnf_rules g`);
  - The mapped start symbol `Lift_r [inl (start_symbol g)]`.

Note that s_1, s_2 and s_3 represent single arbitrary symbols, t represents a terminal symbol and α and β represent arbitrary strings, and also that:

- Constructor `Lift_cnf_t` creates for every rule $left \rightarrow \alpha t \beta$ of g_1 a new rule $[t] \rightarrow t$;
- Constructor `Lift_cnf_1` creates for every rule $X \rightarrow t$ of g_1 a new rule $[X] \rightarrow t$);
- Constructors `Lift_cnf_2` and `Lift_cnf_3` analyze all the rules such that the right-hand side has three or more symbols, and create, accordingly, new rules with only two symbols in the right-hand side that produce the same effect (as explained in the beginning of the section).

Next, we prove that `g_cnf g` (or g_2) is equivalent to `g` (or g_1). It should be noted, however, that the set of rules defined above do not generate the empty string. If this is the case, then we construct g_3 with a new empty rule, as follows:

$$\begin{array}{ccc} g_1 & \rightsquigarrow & g_3 \\ P_1 & \underbrace{\rightsquigarrow}_{same\ of\ g_2\ +1} & P_3 \\ \Sigma & \underbrace{\rightsquigarrow}_{same\ of\ g_2} & \Sigma \\ V_1 & \underbrace{\rightsquigarrow}_{same\ of\ g_2} & V_3 \\ S_1 & \underbrace{\rightsquigarrow}_{same\ of\ g_2} & S_3 \end{array}$$

The definitions below represent a new grammar `g_cnf'` (or g_3) with a new rule that adds the empty string to the language generated by `g_cnf` (or g_2):

```
Inductive g_cnf'_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal' → sf' → Prop:=
| Lift_cnf'_all:
  ∀ left: non_terminal',
  ∀ right: sf',
  g_cnf_rules g left right →
  g_cnf'_rules g left right
```

```
| Lift_cnf'_new:
   g_cnf'_rules g (start_symbol (g_cnf g)) [].

Definition g_cnf'
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal' terminal:=
  {| start_symbol:= start_symbol (g_cnf g);
     rules:= g_cnf'_rules g;
     rules_finite:= g_cnf'_finite g |}.
```

To summarize, we relate below the transformations that create g_3 (or `g_emp' g`) from g_1 (or `g`) to the new inductive definitions and corresponding constructors:

- For the new set of non-terminals:
  - The same of g_2.
- For the new set of rules (inductive definition `g_cnf'_rules`):
  - The same of g_2 (constructor `Lift_cnf'_all`);
  - One extra rule $[S_2] \rightarrow \varepsilon$ (constructor `Lift_cnf'_new`).
- For the new grammar (definition `g_cnf'`):
  - The same set of non-terminals of g_2;
  - The new set of rules (`g_cnf'_rules g`);
  - The same start symbol of g_2.

The statement of the CNF theorem can then be presented as:

```
Theorem g_cnf_ex:
∀ g: cfg non_terminal terminal,
(produces_empty g ∨ ~produces_empty g) ∧
(produces_non_empty g ∨ ~produces_non_empty g) →
∃ g': cfg (non_terminal' (emptyrules.non_terminal' non_terminal) terminal)
terminal,
g_equiv g' g ∧
(is_cnf g' ∨ is_cnf_with_empty_rule g') ∧
start_symbol_not_in_rhs g'.
```

It should be observed that the statement of `g_cnf_ex` is not entirely constructive, as we require, for any context-free grammar `g`, a proof that either `g` produces the empty string or `g` does not produce the empty string, and also that `g` produces a non-empty string or `g` does not produce a non-empty string. Since we have not yet included a proof of the decidability of these predicates in our formalization (something that can be done in the future), the statement of the lemma has to require such proofs explicitly. They are demanded, respectively, by the elimination of empty rules and elimination of useless symbols phases of grammar simplification (see Section 6.6.1 and Section 6.6.3).

The new predicates used above assert, respectively, that the argument grammar (i) is in the Chomsky Normal Form (`is_cnf`) and (ii) is in the Chomsky Normal Form and has a single empty rule with the start symbol in the left-hand side (`is_cnf_with_empty_rule`):

```
Definition is_cnf_rule (left: non_terminal) (right: sf): Prop:=
(∃ s1 s2: non_terminal, right = [inl s1; inl s2]) ∨
(∃ t: terminal, right = [inr t]).

Definition is_cnf (g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right → is_cnf_rule left right.

Definition is_cnf_with_empty_rule (g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right →
(left = (start_symbol g) ∧ right = []) ∨
is_cnf_rule left right.
```

The proof of this theorem starts with a case analysis on the two hypotheses:

1. `produces_empty g` and `produces_non_empty g`:
   (that is, $\varepsilon \in L(g)$ and $L(g) - \{\varepsilon\} \neq \emptyset$)

2. `~produces_empty g` and `produces_non_empty g`:
   (that is, $\varepsilon \notin L(g)$ and $L(g) \neq \emptyset$)

3. `produces_empty g` and `~produces_non_empty g`:
   (that is, $L(g) = \{\varepsilon\}$)

4. `~produces_empty g` and `~produces_non_empty g`:
   (that is, $L(g) = \emptyset$)

For case (3), we prove that grammar $g' = (\{S'\}, \{\}, \{S' \to \varepsilon\}, S')$ is in CNF (that is, it satisfies `is_cnf_with_empty_rule`), is equivalent to g (that is, it satisfies `g_equiv g' g`) and satisfies also `start_symbol_not_in_rhs g'`. This result is proved by lemma `g_cnf_ex_v3`.

For case (4), we prove that grammar $g' = (\{S'\}, \{\}, \{\}, S')$ is in CNF (that is, it satisfies `is_cnf`), is equivalent to g (that is, it satisfies `g_equiv g' g`) and satisfies also `start_symbol_not_in_rhs g'`. This result is proved by lemma `g_cnf_ex_v4`.

The proofs of cases (1) and (2) are longer, more complex and similar, except for the fact that for (1) we prove `is_cnf_with_empty_rule` and for (2) we prove `is_cnf`.

For case (1), we first simplify g according to lemma `g_simpl_ex_v2` (which gives us an equivalent grammar g', except for the empty string). Then, we use this g' to prove the existential quantifier of the goal. For the `g_equiv g' g` part of the goal, we first substitute it (using lemma `g_equiv_split`) by:

```
(produces (g_cnf' g') [] ↔ produces g []) ∧
g_equiv_without_empty (g_cnf' g') g
```

since $\varepsilon \in L(g)$ and $\varepsilon \notin L(g')$. The `g_equiv_without_empty` predicate asserts that the argument grammars are equivalent without considering the empty string:

```
Definition g_equiv_without_empty
(g1: cfg non_terminal terminal) (g2: cfg non_terminal' terminal): Prop:=
∀ s: sentence,
s ≠ [] →
(produces g1 s ↔ produces g2 s).
```

The first part of the conjunction is straightforward to prove from the hypotheses and the constructor `Lift_cnf'_new` of `g_cnf'_rules`. For the second part of the conjunction (`g_equiv_without_empty (g_cnf' g') g`), we substitute it (using lemma `g_equiv_without_empty_trans`) by:

```
g_equiv_without_empty (g_cnf' g') g' ∧
g_equiv_without_empty g' g
```

The first part of this conjunction is proved by lemma `g_cnf'_correct` (correctness of `g_cnf'`) and the second part comes directly from the hypothesis. The proof of `g_cnf'_correct` is finally reduced to the proof of `g_cnf_correct_v1` (correctness of `g_cnf`), discussed next.

Case (2) is very similar to case (1), except that we use `g_simpl_ex_v1` to obtain the simplified grammar g' which, in this case, generates the empty string. Then, we use `g_cnf g'` for the existental quantifier of the goal. Next, we substitute `g_equiv (g_cnf g') g` by:

```
g_equiv (g_cnf g') g' ∧ g_equiv g' g
```

We now substitute the first part of the conjunction by `g_equiv_without_empty (g_cnf g') g'`, since $\varepsilon \notin L(g')$ (using lemma `with_without_empty`) and then apply lemma `g_cnf_corrrect_v1` (correctness of `g_cnf`). For the second part of this conjunction we can use the hypotheses directly.

The proof of `g_cnf_corrrect_v1` is based in the proof of two other lemmas, namely `derives_g_cnf_g` and `derives_g_g_cnf_v1`, one for each direction of the implication $S \Rightarrow_g^* w \leftrightarrow S' \Rightarrow_{g\_cnfg}^* w$.

For the proof of `derives_g_g_cnf_v1` (that is, $S \Rightarrow_g^* w \rightarrow S' \Rightarrow_{g\_cnfg}^* w$), the strategy adopted is to prove that for every rule $left \rightarrow_g right$, either $left \rightarrow_{g\_cnfg} right$ or $left \Rightarrow_{g\_cnfg}^* right$.

For the proof of `derives_g_cnf_g`, that is, ($S \Rightarrow_g^* w \leftarrow S' \Rightarrow_{g\_cnfg}^* w$), it is enough to note that the sentential forms of `g` are embedded in the sentential forms of `g_cnf g`, specifically in the arguments of the constructor `Lift_r` of `non_terminal'`. Because of this, a simple extraction mechanism, resulting from the combination of `derives_g_cnf_g_aux` and fixpoint `flat_sf` (below) allows the implication to be proved.

```
Lemma derives_g_cnf_g_aux:
∀ g: cfg _ _,
∀ l r: sf',
derives (g_cnf g) l r →
derives g (flat_sf l) (flat_sf r).

Fixpoint flat_sf (l: sf'): sf :=
match l with
| nil ⇒ nil
| inl (Lift_r x) :: xs ⇒ x ++flat_sf xs
| inr x :: xs ⇒ inr x :: flat_sf xs
end.
```

The `flat_sf` operation defined above simply maps a sentential form that uses the `Lift_r` constructor and the `symbol_lift` map operator into the corresponding sentential which is free of them and is obtained from their arguments. Thus, the statement of lemma `derives_g_cnf_g_aux` can be used to ensure that every derivation in `g_cnf` has a corresponding derivation in `g`. The proof is obtained by induction on the structure of the sentential form `l` (or s_1 in the previous paragraph) with some auxiliary lemmas.

Using the previous example, suppose we have: $X[YZd] \Rightarrow_{g\_cnfg}^* abcd$, which would be represented in our formalization as:

```
derives (g_cnf g) [inl X] ++[inl (Lift_r ([inl Y; inl Z; inr d]))]
(map (·symbol_lift _ _) (map term_lift [inr a; inr b; inr c; inr d]))
```

The extraction mechanism, applied to this case, would yield:

```
derives g [inl X; inl Y; inl Z; inr d] (map term_lift [inr a; inr b; inr c; inr d])
```

which is exactly the expected result ($XYZd \Rightarrow_g^* abcd$). Observe that `[inl X; inl Y; inl Z; inr d]` is obtained from `[inl X] ++ [inl (Lift_r ([inl Y; inl Z; inr d]))]` by eliminating the `Lift_t` constructor, and that `(map term_lift [inr` a; inr b; inr c; inr d]) is obtained from (map (@symbol_lift _ _) (map term_lift [inr a; inr b; inr c; inr d])) by eliminating the symbol_lift map operator.

Regarding the proof of predicate start_symbol_not_in_rhs g' in theorem g_cnf_ex, it comes directly from the fact that this property is satisfied by the simplified grammar (indeed, by the first phase of grammar simplification, elimination of empty rules), prior to normalization, and the fact the normalization itself does change it.

The formalization of this section required a lot of insights not directly available from the informal proofs, the most important being the definition of g_cnf_rules. In a first attempt, this inductive definition resulted with 14 constructors. Although correct, it was refined many times until it was simplified to only 4 after the definition of non_terminal' was adjusted properly, with a single constructor. This effort resulted in elegant definitions which allowed the simplification of the corresponding proofs, thus leading to a natural and readable formalization. In particular, the strategy used in the proof of lemma derives_g_cnf_g is a very simple and elegant one, which uses the information already available in the definition of g_cnf_rules.

6.8 Generic Binary Trees Library

In order to support the formalization of the Pumping Lemma, an extensive library of definitions and lemmas on binary trees and their relation to CNF grammars has been developed. This library (named trees.v) has 4,539 lines of Coq script (~18.9% of the total), 84 lemmas (~15.7% of the total)) and is based in the definition of a binary tree (btree) whose internal nodes are non-terminal symbols and leaves are terminal symbols. The type btree is defined with the objective of representing derivation trees for strings generated by context-free grammars in the Chomsky Normal Form:

```
Inductive btree (non_terminal terminal: Type): Type:=
| bnode_1: non_terminal → terminal → btree
| bnode_2: non_terminal → btree → btree → btree.
```

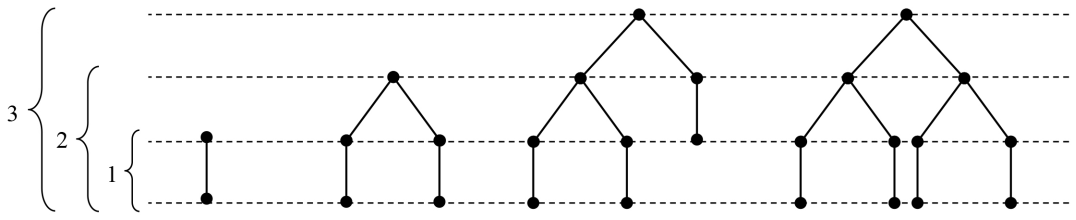
Figure 6.2 is a graphical representation of sample binary trees based on the definition of btree, with different heights.

The constructors of btree relate to the two possible forms that the rules of a CNF grammar can assume (namely with one terminal symbol or two non-terminal symbols in the right-hand side). Naturally, the inhabitants of the type btree can only represent the derivation of non-empty strings.

The root, the frontier and the height of a btree are defined in the usual manner:

```
Definition broot (t: btree): non_terminal:=
match t with
| bnode_1 n t ⇒ n
```

Figure 6.2: Binary trees and their heights



Source: the author

```
| bnode_2 n t1 t2 ⇒ n
end.

Fixpoint bfrontier (t: btree): sentence:=
match t with
| bnode_1 n t ⇒ [t]
| bnode_2 n t1 t2 ⇒ bfrontier t1 ++bfrontier t2
end.

Fixpoint bheight (t: btree): nat:=
match t with
| bnode_1 n t ⇒ 1
| bnode_2 n t1 t2 ⇒ S (max (bheight t1) (bheight t2))
end.
```

Among other useful lemmas, the following one is of fundamental importance in the proof of the Pumping Lemma, as it relates the length of the frontier of a binary tree to its height:

```
Lemma length_bfrontier_ge:
∀ t: btree,
∀ i: nat,
length (bfrontier t) ≥ 2 ^ (i − 1) →
bheight t ≥ i.
```

The notion of subtree is also important, and is defined inductively as follows (note that a tree is not, in this definition, a subtree of itself):

```
Inductive subtree (t: btree): btree → Prop:=
| sub_br: ∀ tl tr: btree,
          ∀ n: non_terminal,
          t = bnode_2 n tl tr →
          subtree t tr
| sub_bl: ∀ tl tr: btree,
          ∀ n: non_terminal,
        t = bnode_2 n tl tr →
        subtree t tl
| sub_ir: ∀ tl tr t': btree,
        ∀ n: non_terminal,
        subtree tr t' →
        t = bnode_2 n tl tr →
        subtree t t'
| sub_il: ∀ tl tr t': btree,
        ∀ n: non_terminal,
        subtree tl t' →
        t = bnode_2 n tl tr →
        subtree t t'.
```

The following lemmas, related to subtrees, are also fundamental in the proof of the Pumping Lemma:

```
Lemma subtree_trans:
∀ t1 t2 t3: btree,
subtree t1 t2 →
subtree t2 t3 →
subtree t1 t3.

Lemma subtree_includes:
∀ t1 t2: btree,
subtree t1 t2 →
∃ l r : sentence,
bfrontier t1 = l ++bfrontier t2 ++r ∧ (l ≠ [] ∨ r ≠ []).
```

A path in a binary tree (`bpath`) is a sequence of non-terminal symbols that starts at its root and ends at a terminal symbol that corresponds to a leaf of the tree:

```
Inductive bpath (bt: btree): sf → Prop:=
| bp_1: ∀ n: non_terminal,
       ∀ t: terminal,
       bt = (bnode_1 n t) → bpath bt [inl n; inr t]
| bp_l: ∀ n: non_terminal,
       ∀ bt1 bt2: btree,
       ∀ p1: sf,
       bt = bnode_2 n bt1 bt2 → bpath bt1 p1 → bpath bt ((inl n) :: p1)
| bp_r: ∀ n: non_terminal,
       ∀ bt1 bt2: btree,
       ∀ p2: sf,
       bt = bnode_2 n bt1 bt2 → bpath bt2 p2 → bpath bt ((inl n) :: p2).
```

A bpath is a string that starts at the root of a btree and ends at a leaf of the same btree. Thus, it contains a sequence of non-terminal symbols, ends with a terminal symbol and its length is limited by the height of the btree. If a bpath is maximal, then its length corresponds to the height of the btree plus one. The following lemma asserts that every btree has a maximal bpath:

```
Lemma btree_ex_bpath:
∀ bt: btree,
∀ ntl: list non_terminal,
bheight bt ≥ length ntl + 1 →
bnts bt ntl →
∃ z: sf,
bpath bt z ∧
length z = bheight bt + 1 ∧
∃ u r: sf,
∃ t: terminal,
z = u ++r ++[inr t] ∧
length u ≥ 0 ∧
length r = length ntl + 1 ∧
(∀ s: symbol, In s (u ++r) → In s (map inl ntl)).
```

Predicate bnts is used to express the fact that all non-terminal symbols of a btree come from the same list:

```
Inductive bnts (bt: btree) (ntl: list non_terminal): Prop:=
| bn_1: ∀ n: non_terminal,
        ∀ t: terminal,
        bt = (bnode_1 n t) → In n ntl → bnts bt ntl
| bn_2: ∀ n: non_terminal,
        ∀ bt1 bt2: btree,
        bt = bnode_2 n bt1 bt2 →
        In n ntl →
        bnts bt1 ntl →
        bnts bt2 ntl →
        bnts bt ntl.
```

Since a path is not unique in a tree, it is necessary to use another representation that can describe this path uniquely, which is done by the predicate bcode. It uses a sequence of boolean values (false, true) to respectively select the left or right subtrees in a tree, and thus uniquely define a path in it:

```
Inductive bcode (bt: btree): list bool → Prop:=
| bcode_0: ∀ n: non_terminal,
            ∀ t: terminal,
           bt = (bnode_1 n t) → bcode bt []
| bcode_1: ∀ n: non_terminal,
           ∀ bt1 bt2: btree,
           ∀ c1: list bool,
           bt = bnode_2 n bt1 bt2 → bcode bt1 c1 → bcode bt (false :: c1)
| bcode_2: ∀ n: non_terminal,
           ∀ bt1 bt2: btree,
           ∀ c2: list bool,
           bt = bnode_2 n bt1 bt2 → bcode bt2 c2 → bcode bt (true :: c2).
```

Lemma bpath_ex_bcode asserts that every bpath can be assigned a bcode, and is proved by induction on t:

```
Lemma bpath_ex_bcode:
∀ t: btree,
∀ p: sf,
bpath t p →
∃ c: list bool,
bcode t c ∧
bpath_bcode t p c.
```

The predicate bpath_bcode ensures that bcode c is valid for bpath p in tree t:

```
Inductive bpath_bcode (bt: btree): sf → (list bool) → Prop:=
| bb_0: ∀ n: non_terminal,
        ∀ t: terminal,
        bt = (bnode_1 n t) → bpath_bcode bt [inl n; inr t] []
| bb_1: ∀ n: non_terminal,
        ∀ bt1 bt2: btree,
        ∀ c1: list bool,
        ∀ p1: sf,
        bt = (bnode_2 n bt1 bt2) →
        bpath bt1 p1 →
        bpath_bcode bt1 p1 c1 →
        bpath_bcode bt ((inl n) :: p1) (false :: c1)
| bb_2: ∀ n: non_terminal,
        ∀ bt1 bt2: btree,
        ∀ c2: list bool,
        ∀ p2: sf,
        bt = (bnode_2 n bt1 bt2) →
        bpath bt2 p2 →
        bpath_bcode bt2 p2 c2 →
        bpath_bcode bt ((inl n) :: p2) (true :: c2).
```

The following lemma, which is key in the formalization of the Pumping Lemma, has a statement with a number of hypotheses and conclusions which provide useful information on the newly identified subtree. Among them, its height and its frontier (this one embedded in the definition btree_decompose). It is proved by induction on t:

```
Lemma bcode_split:
∀ t: btree,
∀ p1 p2: sf,
∀ c: list bool,
bpath_bcode t (p1 ++p2) c →
length p1 > 0 →
length p2 > 1 →
bheight t = length p1 + length p2 − 1 →
∃ c1 c2: list bool,
c = c1 ++c2 ∧
length c1 = length p1 ∧
∃ t2: btree,
∃ x y: sentence,
bpath_bcode t2 p2 c2 ∧
btree_decompose t c1 = Some (x, t2, y) ∧
bheight t2 = length p2 − 1.
```

Function btree_decompose takes as arguments a tree and a sequence of boolean values, and returns a triple consisting of the subtree located in this position and the two sentences to the left and right of it. It is used to enable reasoning on the frontiers of the subtrees obtained before:

```
Fixpoint btree_decompose (bt: btree) (c: list bool):
option (sentence ∗ btree ∗ sentence):=
match bt, c with
| bnode_1 n t, [] ⇒
          Some ([], bt, [])
| bnode_1 n t, _ ⇒
          None
| bnode_2 n bt1 bt2, [] ⇒
          Some ([], bt, [])
| bnode_2 n bt1 bt2, false :: c ⇒
          match btree_decompose bt1 c with
          | None ⇒ None
          | Some (l, bt, r) ⇒ Some (l, bt, r ++bfrontier bt2)
          end
| bnode_2 n bt1 bt2, true :: c ⇒
          match btree_decompose bt2 c with
          | None ⇒ None
          | Some (l, bt, r) ⇒ Some (bfrontier bt1 ++l, bt, r)
          end
end.
```

Next, the substitution of a subree by another at a certain point in a tree is implemented by the following function, for which a number of lemmas have been proved:

```
Fixpoint btree_subst (t1 t2: btree) (c: list bool): option btree:=
match t1, c with
| bnode_1 n t, [] ⇒ Some t2
| bnode_1 n t, _ ⇒ None
| bnode_2 n1 bt1 bt2, [] ⇒ Some t2
| bnode_2 n1 bt1 bt2, false :: y ⇒
                                 match (btree_subst bt1 t2 y) with
                                 | None ⇒ None
                                 | Some bt1 ⇒ Some (bnode_2 n1 bt1 bt2)
                                 end
| bnode_2 n1 bt1 bt2, true :: y ⇒
                                 match (btree_subst bt2 t2 y) with
                                 | None ⇒ None
                                 | Some bt2 ⇒ Some (bnode_2 n1 bt1 bt2)
                                 end
end.
```

Finally, we have relate binary trees to CNF grammars. This is done with the predicate `btree_cnf`, used to assert that a binary tree `bt` represents a derivation in CNF grammar `g`:

```
Inductive btree_cnf (g: cfg non_terminal' terminal)
(bt: btree non_terminal' terminal): Prop:=
| bt_c1: ∀ n: non_terminal',
        ∀ t: terminal,
        rules g n [inr t] →
        bt = (bnode_1 n t) →
        btree_cnf g bt
| bt_c2: ∀ n n1 n2: non_terminal',
        ∀ bt1 bt2: btree _ _,
        rules g n [inl n1; inl n2] →
        btree_cnf g bt1 →
        broot bt1 = n1 →
        btree_cnf g bt2 →
        broot bt2 = n2 →
        bt = (bnode_2 n bt1 bt2) →
        btree_cnf g bt.
```

Now we can show that binary trees and derivations in CNF grammars are equivalent. This is accomplished by two lemmas, one for each direction of the equivalence.

Lemma `derives_g_cnf_equiv_btree` asserts that for every derivation in a CNF grammar exists a binary tree that represents this derivation. It is general enough in order to accept that the input grammar might either be a CNF grammar, or a CNF grammar with an empty rule. If this is the case, then we have to ensure that the derived sentence is not empty>

```
Lemma derives_g_cnf_equiv_btree:
∀ g: cfg non_terminal' terminal,
∀ n: non_terminal',
∀ s: sentence,
s ≠ [] →
(is_cnf g ∨ is_cnf_with_empty_rule g) →
start_symbol_not_in_rhs g →
derives g [inl n] (map term_lift' s) →
∃ t: btree non_terminal' terminal,
btree_cnf g t ∧
broot t = n ∧
bfrontier t = s.
```

The proof of `derives_g_cnf_equiv_btree` is reasonably long and uses induction on the number of derivation steps in G' in order to generate α.

Lemma `btree_equiv_derives_g_cnf` proves that every binary tree that satisfies `btree_cnf` corresponds to a derivation in the the same (CNF) grammar:

```
Lemma btree_equiv_derives_g_cnf:
∀ g: cfg _ _,
∀ t: btree _ _,
btree_cnf g t →
derives g [inl (broot t)] (map inr (bfrontier t)).
```

Its proof is done by induction on the structure of the tree. This strategy, together with case analysis on the structure of the tree and induction and case analysis in the other definitions of the library, is used in the proofs of most of the lemmas of this section.

6.9 Pumping Lemma

The statement and applications of the Pumping Lemma for CFLs (or Pumping Lemma for short) are presented in Section 6.9.1. A typical informal proof, which served as the basis for the present formalization, is described in Section 6.9.2. The formalization is then described in Section 6.9.3, where the definitions and auxiliary lemmas used are discussed in some detail, as well as the Pumping Lemma itself[6].

[6]The results of this section are available in libraries `pigeon.v` and `pumping.v`.

6.9.1 Statement and Application

The Pumping Lemma is a property that is verified for all CFLs and was stated and proved for the first time by Bar-Hillel, Perles and Shamir in 1961 (BAR-HILLEL; PERLES; SHAMIR, 1961), and further published in Bar-Hillel (1964).

The Pumping Lemma does not characterize the CFLs, however, since it is also verified by some non CFLs. It states that, for every context-free language and for every sentence of such a language that has a certain minimum length, it is possible to obtain an infinite number of new sentences that must also belong to the language. This minimum length depends only on the language defined. In other words (let $\mathscr{L}$ be defined over alphabet Σ):

$$\begin{gathered}\forall\, \mathscr{L}, (\text{cfl}\ \mathscr{L}) \rightarrow \exists\, n \,| \\ \forall\, \alpha, (\alpha \in \mathscr{L}) \wedge (|\alpha| \geq n) \rightarrow \\ \exists\, u, v, w, x, y \in \Sigma^* \,|\, (\alpha = uvwxy) \wedge (|vx| \geq 1) \wedge (|vwx| \leq n) \wedge \\ \forall\, i, uv^iwx^iy \in \mathscr{L}\end{gathered}$$

A typical use of the Pumping Lemma is to show that a certain language is not context-free by using the contrapositive of the statement of the lemma. The proof proceeds by contraposition: the language is assumed to be context-free, and this leads to a contradiction from which one concludes that the language in question can not be context-free.

As an example, consider the language $\mathscr{L} = \{a^ib^ic^i \,|\, i \geq 1\}$. This language is defined over the alphabet $\{a, b, c\}$ and includes sentences such as $abc, aabbcc, aaabbbccc$.

Should $\mathscr{L}$ be context-free, then the Pumping Lemma should hold for it. Consider n to be the constant of the Pumping Lemma and let's choose the sentence $\alpha = a^nb^nc^n$. Clearly, $\alpha \in \mathscr{L}$ and $|\alpha| = 3 * n \geq n$. Thus, $\alpha = uvwxy$ such that $|vx| \geq 1$, $|vwx| \leq n$ and $uv^iwx^iy \in \mathscr{L}, i \geq 0$.

However, it is easy to observe that, due to its length limitation, the sentence vwx should contain only one or two different symbols (namely, vwx should belong to either a^*, b^*, c^*, a^*b^* or b^*c^*). This implies that the repetition of v and x in uv^iwx^iy should increase (or decrease) the number of at most two different symbols while keeping the number of the third symbol unchanged. As a result, the new sentence can not belong to the language and this proves that the initial hypothesis can not be true. Thus, $\mathscr{L}$ is not a context-free language.

6.9.2 Informal Proof

In short, the Pumping Lemma derives from the fact that the number of non-terminal symbols in any context-free grammar G that generates $\mathscr{L}$ is finite. There are different strategies that can be used to prove that the lemma can be derived from this fact. We searched through 13 proofs published in different textbooks and articles by different authors, and concluded that in 6 cases (BAR-HILLEL, 1964; HOPCROFT; ULLMAN, 1969; DAVIS; SIGAL; WEYUKER, 1994; KOZEN, 1997; HOPCROFT et al., 2000; SUDKAMP, 2006) the strategy uses CNF grammars and binary trees for representing derivations. The other 5 cases (GINSBURG, 1966; DENNING; DENNIS; QUALITZ, 1978; BROOKSHEAR, 1989; LEWIS; PAPADIMITRIOU, 1998; SIPSER, 2005) present tree-based proofs that however do not require the grammar to be in CNF. Finally, Harrison (HARRISON, 1978) proves the Pumping Lemma as a corolary to the more general Ogden's Lemma and Amarilli and Jeanmougin (AMARILLI; JEANMOUGIN, 2012) use a strategy with pushdown automata instead of context-free grammars.

The difference between the proofs that use CNF grammars and those that don't is that the former uses $n = 2^k$ (where k is the number of non-terminal symbols the grammar) and the latter uses $n = m^k$ (where m is the length of the longest right-hand side among all rules of the grammar and k is the number of non-terminal symbols in the grammar). In both cases, the idea is the same: to show that sufficiently long sentences have parse trees for which a maximal path contains at least two instances of the same non-terminal symbol.

Since 11 out of 13 proofs considered use grammars and generic trees and, of these, 6 use CNF grammars and binary trees (including the authors of the original proof), this strategy was initially considered as the choice for the present work. Besides that, binary trees can be easily represented in Coq as simple inductive types, where generic trees require mutually inductive types, which increases the complexity of related proofs. Thus, for all these reasons we have adopted the proof strategy that uses CNF grammars and binary trees in what follows.

The classical proof considers that G is in the Chomsky Normal Form (a form in which the rules of the grammar have at most two symbols in the right-hand side), which means that derivation trees have the simpler form of binary trees. Then, if the sentence has a certain minimum length, the frontier of the derivation tree should have two or more instances of the same non-terminal symbol in some path that starts in the root of this tree. Finally, the context-free character of G guarantees that the subtrees related to these duplicated non-terminal symbols can be cut and pasted in such a way that an infinite number of new derivation trees are obtained, each of which is related to a new sentence of the language.

The proof comprises the following steps and is based in Ramos, Neto and Vega (2009):

1. Since $\mathscr{L}$ is declared to be a context-free language (predicate `cfl`), then there exists a context-free grammar G such that $L(G) = \mathscr{L}$;

2. Obtain G' such that G' is in Chomsky Normal Form and $L(G') = L(G)$;

3. Take n as 2^k, where k is the number of non-terminal symbols in G';

4. Consider an arbitrary sentence α such that $\alpha \in \mathscr{L}$ and $|\alpha| \geq n$;

5. Obtain a derivation tree t that represents the derivation of α in G';

6. Take a path that starts in the root of t and whose length is the height of t plus 1 (maximum length);

7. Then, the height of t should be greater or equal than $k+1$;
8. This means that the selected path has at least $k+2$ symbols, being at least $k+1$ non-terminals and one (the last) a terminal symbol;
9. Since G' has only k non-terminal symbols, this means that this path has at least one non-terminal symbol that appears at least twice in it;
10. Name the duplicated symbols n_1 and n_2 ($n_1 = n_2$) and the corresponding subtrees t_1 and t_2 (note that t_2 is a subtree of t_1 and t_1 is a subtree of t);
11. It is then possible to prove that the height of t_1 is greater than or equal to 2, and less than or equal to 2^k;
12. Also, that the height of t_2 is greater than or equal to 1 and less than or equal to 2^{k-1};
13. This implies that the frontier of t can be split into five parts: u,v,w,x,y, where w is the frontier of t_2 and vwx is the frontier of t_1;
14. As a consequence of the heights of the corresponding subtrees, it can be shown that $|vx| \geq 1$ and $|vwx| \leq n$;
15. If t_1 is removed from t, and t_2 is inserted in its place, then we have a new tree t^0 that represents the derivation of string $uv^0wx^0y = uwy$;
16. If, instead, t_1 is inserted in the place where t_2 lies originally, then we have a tree t^2 that represents the derivation of string uv^2wx^2y;
17. Repetition of the previous step generates all trees t^i that represent the derivation of the string uv^iwx^iy, $\forall i \geq 2$.

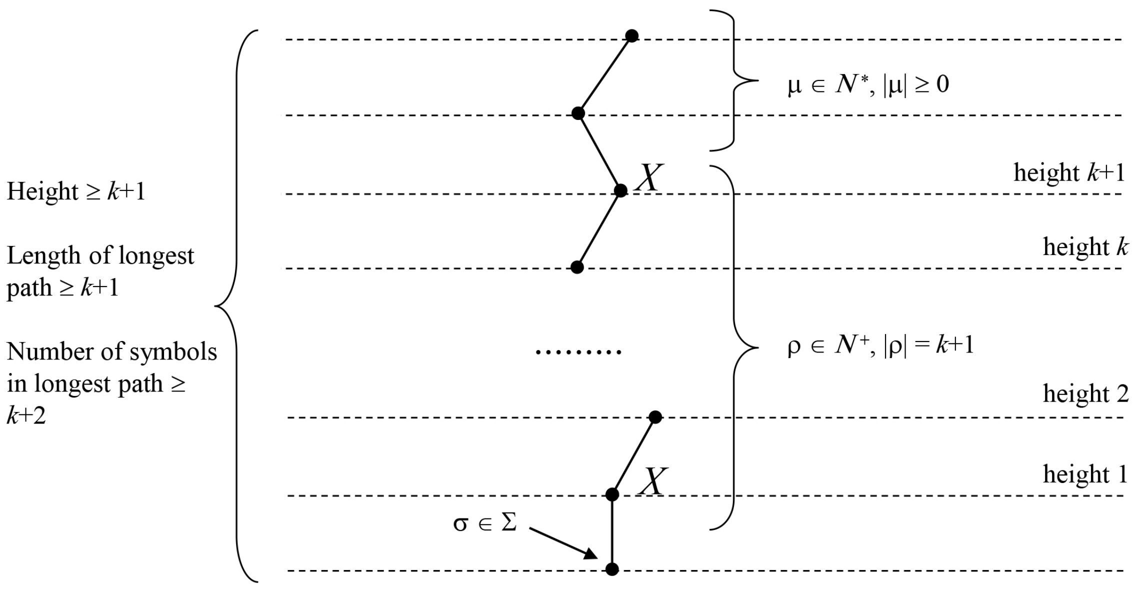
Figure 6.3 illustrates the analysis of the longest path of the tree: in the bottom we have the terminal symbol, above it we have a string of exactly $k+1$ non-terminal symbols and finally a (possibily empty) string of non-terminal symbols. It corresponds to steps 1 to 12 of the informal proof, and allows one to confirm the results presented up to this point.

Figure 6.4 shows that the tree has at least two subtrees with the same non-terminal symbol as a root (steps 13 and 14), and finally Figure 6.5 presents some possible substitutions of these trees in order to generate different strings (steps 15 to 17).

The Pumping Lemma is valid also for finite languages, and not only for infinite languages. An informal proof is:

- Suppose L is finite;
- Let G in CNF such that $L = L(G)$;

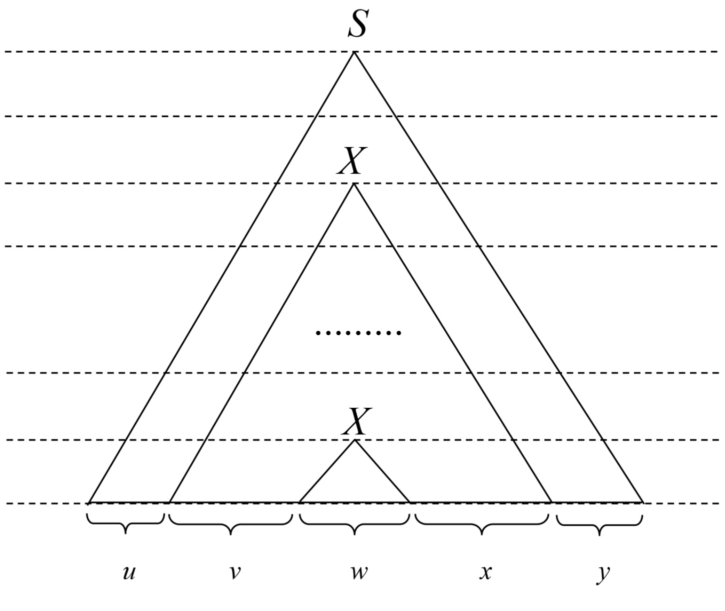
Figure 6.3: Analysis of the longest path



$\mu \in N^*, |\mu| \geq 0$
X
height k+1
Height $\geq k$+1
Length of longest path $\geq k$+1
height k
Number of symbols in longest path $\geq k$+2
....
$\rho \in N^+, |\rho| = k$+1
height 2
X
height 1
$\sigma \in \Sigma$

Source: the author

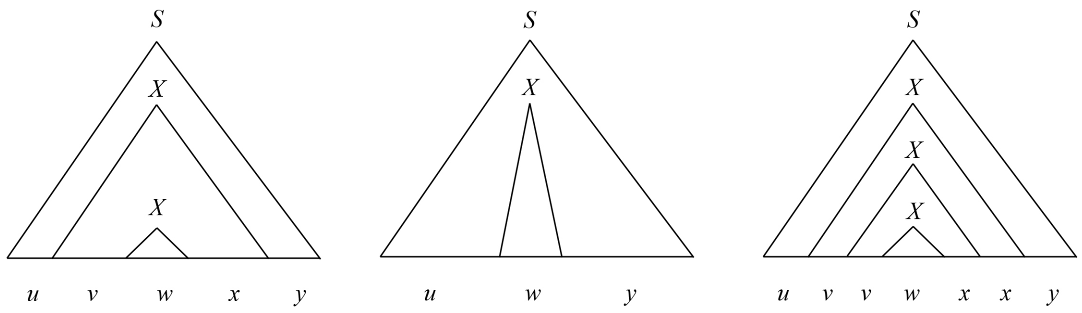
Figure 6.4: Subtrees with the same root



Source: the author

- Let k be the number of non-terminals of G;
- We claim there is no $w \in L$ such that $|w| \geq 2^k$:
    - If there were, then the PL asserts that L would be infinite, which contradicts the hypothesis.

Figure 6.5: Substitution of subtrees



Source: the author

- Since there is no $w \in L$ such that $|w| \geq 2^k$, then the PL is trivially true.

6.9.3 Formalization

The formalization follows closely the steps described in Section 6.9.2. The Pumping Lemma has been stated as follows[7]:

```
Lemma pumping_lemma:
∀ l: lang terminal,
(contains_empty l ∨ ~contains_empty l) ∧
(contains_non_empty l ∨ ~contains_non_empty l) →
cfl l →
∃ n: nat,
∀ s: sentence,
l s →
length s ≥ n →
∃ u v w x y: sentence,
s = u ++v ++w ++x ++y ∧
length (v ++x) ≥ 1 ∧
length (u ++y) ≥ 1 ∧
length (v ++w ++x) ≤ n ∧
∀ i: nat, l (u ++(iter v i) ++w ++(iter x i) ++y).
```

Application `iter l i` on a list `l` and a natural `i` yields list `l`$^{\texttt{i}}$.

The proof of the Pumping Lemma starts by finding a grammar G that generates the input language L (this is a direct consequence of the predicate `is_cfl` and corresponds to step 1 of Section 6.9.2). Next, we obtain a CNF grammar G' that is equivalent to G (step 2), using previous results. Then, G is substituted for G' and the value for n is defined as 2^k (step 3) where k is the length of the list of non-terminals of G' (which in turn is obtained from the predicate

[7]Differently from the statement of Section 6.9.1, the statement in Coq of this section contains the extra clause `length (u ++ y) >= 1`, corresponding to $|uy| \geq 1$, which is normally not mentioned in textbooks. The subject is further discussed in Section 6.11.3.

rules_finite). An arbitrary sentence α of $L(G')$ that satisfies the required minimum length n is considered (step 4).

Lemma derives_g_cnf_equiv_btree is then applied in order to obtain a btree t that represents the derivation of α in G' (step 5). Naturally we have to ensure that $\alpha \neq \varepsilon$, which is true since by assumption $|\alpha| \geq 2^k$.

The next step is to obtain a path (a sequence of non-terminal symbols ended by a terminal symbol) that has maximum length, that is, whose length is equal to the height of t plus 1 (steps 6 and 7). This is accomplished by means of the definition bpath and the lemma btree_ex_bpath.

The length of this path (which is $\geq k+2$) allows one to infer that it must contain at least one non-terminal symbol that appears at least twice in it (steps 8, 9 and 10). This result comes from the application of the lemma pigeon which represents a list version of the well-known pigeonhole principle:

```
Lemma pigeon:
∀ A: Type,
∀ x y: list A,
(∀ e: A, In e x → In e y) →
length x = length y + 1→
∃ d: A,
∃ x1 x2 x3: list A,
x = x1 ++[d] ++x2 ++[d] ++x3.
```

This lemma (and other auxiliary lemmas) is included in library pigeon.v, and its proof requires the use of classical reasoning (and thus library Classical_Prop of the Coq Standard Library). It is necessary in order to have a decidable equality on the type of the non-terminals of the grammar, and this is the only place in the whole formalization where this is required. Nevertheless, we plan to pursue in the future a constructive version of this proof.

Since a path is not unique in a tree, it is necessary to use some other representation that can describe this path uniquely, which is done by the predicate bcode and the lemma bpath_ex_bcode.

Once the path has been identified with a repeated non-terminal symbol, and a corresponding bcode has been assigned to it, lemma bcode_split is applied twice in order to obtain the two subtrees t_1 and t_2 that are associated respectively to the first and second repeated non-terminals of t.

From this information it is then possible to extract most of the results needed to prove the goal (steps 11, 12, 13 and 14), except for the pumping condition. This is obtained by an auxiliary lemma pumping_aux, which takes as hypothesis the fact that a tree t_1 (with frontier vwx) has a subtree t_2 (with frontier w), both with the same roots, and asserts the existence of an infinite number of new trees obtained by repeated substitution of t_2 by t_1 or simply t_1 by t_2, with respectively frontiers $v^iwx^i, i \geq 1$ and w, or simply $v^iwx^i, i \geq 0$. The lemma is proved by induction on `i`:

```
Lemma pumping_aux:
∀ g: cfg _ _,
∀ t1 t2: btree (non_terminal' non_terminal terminal) _,
∀ n: _,
∀ c1 c2: list bool,
∀ v x: sentence,
btree_decompose t1 c1 = Some (v, t2, x) →
btree_cnf g t1 →
broot t1 = n →
bcode t1 (c1 ++c2) →
c1 ≠ [] →
broot t2 = n →
bcode t2 c2 →
(∀ i: nat,
 ∃ t': btree _ _,
 btree_cnf g t' ∧
 broot t' = n ∧
 btree_decompose t' (iter c1 i) = Some (iter v i, t2, iter x i) ∧
 bcode t' (iter c1 i ++c2) ∧
 get_nt_btree (iter c1 i) t' = Some n).
```

The proof continues by showing that each of these new trees can be combined with tree t obtained before, thus representing strings $uv^iwx^iy, i \geq 0$ as necessary (steps 15 and 16).

Finally, we prove that each of these trees is related to a derivation in G', which is accomplished by lemma `btree_equiv_produces_g_cnf` (step 17).

The formalization of the Pumping Lemma follows closely the informal proof of Section 6.9.2. It is quite readable and could be easily modified to the alternative version described in Section 6.11.4. It builds nicely on top of the previous results on grammar normalization, which in turn is a consequence of grammar simplification. It is however long (`pumping_lemma` has 436 lines of Coq script) and the key insights for its formalization were (i) the construction of the library trees.v, specially the lemmas that relate binary trees to CNF grammars; (ii) the identification and isolation of lemma `pumping_aux`, to show the pumping of subtrees in a binary tree and (iii) the proof of lemma `pigeon`. None of these aspects are clear from the informal proof, they showed up only while working in the formalization.

6.10 Summary

The whole formalization consists of 23,984 lines of Coq script spread in 18 libraries (each library corresponds to a different file), not including the example files. This number can be considered too high, but is explained by the style adopted for the writing of the scripts: for the sake of clarity, each tactic is placed in its own line, despite the possibility of combining several tactics in the same line. Also, bullets (for structuring the code) were used as much as possible and the sequence tactical (using the semicolon symbol) was avoided in most cases. This duplicates parts of the code but has the advantage of keeping the static structure of the script related to its dynamic behaviour, which favors legibility and maintenance.

The 18 libraries contain 533 lemmas and theorems, 99 constructors, 63 definitions (not including fixpoints), 40 inductive definitions and 20 fixpoints among 1,067 declared names. In what follows, we list the 10 main libraries of the formalization, and for each library the main lemmas and theorems included in it (for more information about the definitions and lemmas included in each of these libraries, please refer to Appendix C):

- Library chomsky.v:
  - `g_cnf_ex`
- Library closure.v:
  - `l_clo_is_cfl`
  - `l_clo_correct`
  - `l_clo_correct_inv`
- Library concatenation.v:
  - `l_cat_is_cfl`
  - `l_cat_correct`
  - `l_cat_correct_inv`
- Library emptyrules.v:
  - `g_emp_correct`
  - `g_emp'_correct`
- Library inaccessible.v:
  - `g_acc_correct`
- Library pumping.v:
  - `pumping_lemma`
  - `pumping_lemma_v2`
- Library simplification.v:
  - `g_simpl_ex_v1`

  - `g_simpl_ex_v2`

- Library union.v:
  - `l_uni_is_cfl`
  - `l_uni_correct`
  - `l_uni_correct_inv`
- Library unitrules.v:
  - `g_unit_correct`
- Library useless.v:
  - `g_use_correct`

Table 6.1 provides extra information about the contents of each library/file. The 8 libraries that appear in this table and are not included in the list above correspond to libraries that have an auxiliary role in the formalization and are not directly related to the final results obtained – they are, nevertheless, of fundamental importance in achieving the objectives of the formalization. These 8 auxiliary libraries contain 11,781 lines of Coq script and correspond to almost half of the formalization (49.1%). Two of these auxiliary libraries (`cfg.v` and `trees.v`) sum, alone, 8,932 lines or more than one third (37.2%) of the total.

The files were created and compiled with the Coq Proof Assistant, version 8.4pl4 (June 2014), using CoqIDE for Windows. They are available for download at <https://github.com/mvmramos/v1> and can be compiled with the following commands under Cygwin, a Linux-like environment for Windows:

- `coq_makefile *.v > _makefile`
  (to create the make file)
- `make -f _makefile`
  (to compile the Coq source files)
- `make -f _makefile html`
  (to compile the Coq source files and generate the corresponding HTML documentation)

Table 6.2 lists the example files and relates them to the corresponding sections of this work.

Table 6.1: List of source files

File name	**Description**	**# of lines**
allrules.v	generates all sentential forms over a given alphabet up to a certain length	259
cfg.v	definitions and basic lemmas regarding context-free grammars and derivations	4,393
cfl.v	definitions and basic lemmas regarding context-free languages	177
chomsky.v	Chomsky grammar normalization	2,333
closure.v	closure of context-free languages over Kleene star	1,042
concatenation.v	closure of context-free languages over concatenation	1,142
emptyrules.v	elimination of empty rules in a context-free grammar	2,966
inaccessible.v	elimination of inaccessible symbols in a context-free grammar	384
misc_arith.v	generic arithmetic related lemmas	230
misc_list.v	generic list lemmas	1,868
misc_logic	generic logic lemmas	44
pigeon.v	pigeonhole principle	271
pumping	pumping lemma for context-free languages	1,052
simplification.v	unification of all previous results	1,029
trees.v	binary trees and their relation to Chomsky grammars	4,539
union.v	closure of context-free languages over union	1,140
unitrules.v	elimination of unit rules in a context-free grammar	464
useless.v	elimination of useless symbols in a context-free grammar	651
Total		23,984

Table 6.2: List of example files

File name	**Description**
example_6_2_1_a.v	first example of Section 6.2.1 (Grammars)
example_6_2_1_b.v	second example of Section 6.2.1 (Grammars)
example_6_2_2.v	example of Section 6.2.2 (Derivations)
example_6_3.v	example of Section 6.3 (Generic CFG Library)
example_6_5_1v̇	example of Section 6.5.1 (Union)

6.11 Discussion

In this section we review the main phases of this project, its major obstacles and dificulties, and the lessons that were learned from them. Section 6.11.1 gives a historical perspective of the project until it resulted in the present format. In Section 6.11.2 we present some general advices for newcomers, based on our own experience. Some design decisions of the present formalization, the related problems and the solutions that were considered for them, are discussed in Section 6.11.3. Hopefully, sharing all this experience will be benefitial for students and professionals willing to work with mathematical formalization. Finally, a discussion about the statement and proof of the Pumping Lemma is presented in Section 6.11.4.

6.11.1 History and Lessons

The development of this formalization faced two main challenges. The first was to acquire the necessary background in order to use the Coq proof assistant and understand the theory behind it. The second was to master the Coq proof assistant itself, in order to use it to represent the part of the context-free language theory considerd in this work. Hopefully, this theory was already mastered by the author (RAMOS; NETO; VEGA, 2009) before the formalization started.

The first challenge was partially fulfilled in the first years of the PhD program (2011-2012), by attending the classes of lambda calculus, set theory and introductory logic. The rest (natural deduction, type theory etc) was acquired autonomously in the following years (2013-2015), with the specific objective of learning the Coq proof assistant. A brief review of these topics is presented in Appendix A.

The second challenge (mastering the Coq proof assistant) proved quite difficult, despite having already acquired some background on the theory. This more or less agrees with the opinion of other researches, who state that they had to use Coq (or any other proof assistant) for many years before they could really say they had effectively mastered the tool.

This long learning period comes, apparently, from different aspects. To start with, the languages of Coq (Gallina and Vernacular) are large and complex. This, however, can be solved by studying the manuals, books and tutorials available, and also by doing small exercises. Second, as with any other language, there is a methodology issue: it is not enough to know the syntax and semantics of the commands, it is also necessary to know how to combine and use them to solve typical problems in the application area of the language. All this, however, is still not enough. We need an extra and fundamental ingredient that can not be obtained from the books alone, which leads to the following lesson:

> *One needs to have a previous hands-on experience in a real world formalization project of some complexity and size, preferably in a group willing to share its*

> *(supposedly) higher expertise and experience, before facing alone the challenges of a similar project.*

For the present formalization, this second aspect proved to be the most difficult of all. This is because trying to master the tool at the same time as trying to use it in a reasonably complex project is definitely not a good strategy.

Let us now review the phases the project went through. When the formalization effort started, in July of 2013, all we had was some background in the theories required by the tool and a formalization experience that consisted only of doing some very simples exercises in Coq (mosty proving basic theorems of propositional and predicate logic). This was not an adequate preparation to face a project with the size and complexity of the present one. Because of this, the mastering of the methodology issue (and also the Coq languages' syntax and semantics) had to be developed along with the development of the formalization itself. This, of course, made things much more complicated and reduced productivity in the early stages of the work. The lesson learned from this was:

> *Formalization projects (as with any other projects) should come in increasing size and complexity, allowing the person (or team) involved to be adequately prepared to cope with the new challenges.*

On the other hand, choosing an already mastered theory to be the object of the formalization proved to be a correct decision: this enabled the author to focus in the language and methodology issues, while being assured of the results that he wanted to pursue and consider alternatives for that. This could thus be understood as a third lesson from the present project:

> *Avoid formalizing a theory that you are not familiar with, unless you already master the proof assistant and have some experience with the formalization process. Otherwise, stick to a well-know theory and reduce the risks involved.*

The first formalization attempt started in July of 2013 and lasted until April of 2014. The focus of the formalization then was the regular language theory (regular expressions, finite automata, linear grammars etc). The approach adopted was to write functions that implemented operations on the objects of this theory, such as elimination of empty transitions and non-determinism in finite automata, mapping of regular expressions to finite automata and so on. This effort resulted in some 6,000 lines of fully executable Coq scripts, however no lemmas or theorems were proved.

This tentative was a direct consequence of our personal background as a programmer, which made us look at Coq, in the first moment, only as a (functional) programming language (a secondary characteristic of Coq) and not as a reasoning tool (its main characteristic). Naturally, the work could have evolved by pursuing proofs of the correctness of the functions implemented. However, when the focus changed from regular to context-free language theory, the approach adopted also change. In the second and final attempt, we avoided the creation of functions and pursued the proof of some basic lemmas. This resulted in a completely different nature of the formalization, when compared to the one obtained in the first tentative: the fully declarative approach could not be executed and prevented the extraction of certified programs. On the other hand, it was considered more natural and closer to the approach of most textbooks on the subject, and for this reason was kept to the end.

The present formalization was developed from April of 2014 until August of 2015, and started right after a meeting with Mr. Bertot at INRIA Sophia-Antipolis (France) in April 23rd, 2014. In this meeting, Mr. Bertot explained that the formalization of a theory should start with its simplest theorem, and then move gradually towards the more complex ones. As a result, the work concentrated successfully on the formalization of the closure property for grammar union, moving later to concatenation and Kleene star. The rest was formalized in the same sequence presented here (simplification, Chomsky and Pumping Lemma), which was considered a sequence of increasing complexity, where some of the early proofs were subsequently reused and adapted in the construction of the latter and more complex ones. This is also an important lesson:

> *The formalization of any theory should start with the shortest, simpler and more independent lemmas and theorems, and proceed towards the largest and more complex ones, benefiting from previous results.*

The limited time available for the formalization (16 months) did not allow for much of experimentation with different representation choices and proof strategies, after successful results were obtained. On the contrary, the decisions that proved successful were immediately adopted as they were in order to allow the for advance of the formalization and the fulfillment of its objectives. Thus, it is very possible that the formalization could have been optimized in many different ways. However, this is not really important because of the *proof-independence principle*, which claims for a single proof of every statement, no matter how it was built, as long as it is correct. Another consequence of this is the lack of more material to discuss the pros and cons of different choices and strategies. This is compensated, in our opinion, by the scope and number of results that were proved in the present formalization.

6.11.2 General Advices

This section reviews some fundamental aspects of the formalization process which may not be properly considered by newcomers, and thus constitute a source of extra problems to overcome. The following advices might seem obvious to experts, but they come out from our own experience with the observation that it is worth to dedicate enough time and thinking to them in order to increase productivity, maintainability and legibility from the beginning.

Make a deep review of the informal proof: Informal proofs from textbooks are normally too informal to be used as a guideline for a formalization. In many cases, extremely important details are simply missed or left implicit in the text of the informal proof. Thus, do not be satisfied if you have an informal proof of the statement you want to formalize: consider all the minimum steps involved in it, check how they could be represented or proved with your definitions and fill in the gaps by yourself. This will probably lead to a number of auxiliary lemmas that will have to be proved before you prove the main lemmas and theorems. Anyway, it is also a good idea to use different textbooks in order to get as much details and insights as possible before initiating the formalization.

Be sure of the statement to be proved: Quite often, the statement to be proved has many variables and it becomes rather difficult to consider all possible combinations of values for all variables in order to make sure that the statement is always true. As a consequence, it is not unusual for the user to try to prove an unprovable lemma or theorem (which can happen for a special and subtle combination of values). When it becomes too difficult to finish a proof, it is always a good idea to review the statement and make sure that it is true in all situations.

Use the cohesion and coupling principles: When the formalization increases in size, it is important to created modules according to the cohesion and coupling principles of the software engineering: each module should gather all related definitions, lemmas and theorems that refer to the same notion (an object or an operation). On the other hand, the interface between these modules should be as loose as possibile, in order to preserve independence. Coq offers many facilities for achieving these goals and it is important to make good use of them.

Choose a naming policy: A good naming policy, as in any other programming language, will help the user to identify definitions, lemmas and the theorems, among others, in a way that their semantics become obvious, their names can be easily remembered or inferred and similar names refer to similar objects. It is also a software engineering principle that is extremely benefitial in a formalization, specially when working with large name spaces.

Develop a writing style: Indentation, comments, line breaks, spaces and bullets can be very helpful when writing a Coq script. However, it is important to adopt a writing discipline in order to facilitate the reading and maintainance of the code. Since all these aspects offer a great degree of freedom to the user, he should experiment with some altrenatives and, at least informally, decide what fits best for his objectives. The use of a standardized style will prove helpful for the author itself and others that may have to review and/or change the formalization.

Be prepared for lots of trial and error: It is not usual that the proofs are finished and correct in the first attempt. On the contrary, it is likely that many possibilities have to be tried until one proves successful. Sometimes a simple case analysis is sufficient, in others and inductive proof has to be constructed. For statements that use different inductive type definitions (such as natural numbers and lists) and inductively defined predicates, this can be even more difficult: the decision on the object over which to reason inductively may prove successful or not, thus raising the need to consider other alternatives. Sometimes the default induction princple will not work, so a new and more appropriate has to be defined. Except for similar statements that might share similar proofs, each statement represents a new and unique challenge when trying to construct the corresponding proof term.

Do not underestimate the importance of the inductive definitions: Inductive definitions (of types and predicates) play a fundamental role in a formalization. They affect directly the structure, the size and the complexity of the proofs that will have to be construted. Also, incorrect definitions might lead to unprovable lemmas and theorems. Thus, it is of extreme importance that these definitions be thoroughly checked and tested before starting to prove lemmas based on them. It is desirable, as a rule of thumb, to have the least possible number of constructors, to make sure that the statements of the constructors do not overlap, and to make them as simple as possibile in order to ease the construction of the proofs. It is also a good idea to experiment with these definitions before making any proofs, for example by proving some small and simple lemmas, before proceeding to the more complex ones. Since changes to an inductive definition might have a considerable effect in the whole formalization, it is better to make sure that they are correct before doing the rest, under the risk of having to start all over again.

Get rid of useless code: During the process of a formalization, it is common to create definitions and prove lemmas and theorems that will not be used in the final version of the formalization after all. They normally correspond to successful intermediate results of insuccessful tentatives of proving other, more complex lemmas and theorems. While some of them might be valid for themselves, and thus worth keeping, most of them will prove useless and should thus be removed from the formalization in order to reduce its size and ease its understanding.

6.11.3 Design Choices

Despite the fact that, in most cases, different successful design choices were not considered in the formalization, there are still a few aspects related to this subject that are worth discussing.

Constructive versus non-constructive

We should first review some non-constructive aspects of the formalization, its consequences and ways to overcome them. The first is the fact that some lemmas have in their statement instances of the Law of the Excluded Middle (LEM), which is typical from classical reasoning. This happens, for example, in the statement of lemma `g_cnf_ex` (which asserts the existence of a CNF grammar). As mentioned in Section 6.7, this is due to the fact that the predicates used in the statement were not proved decidable yet. Classical reasoning has also been used explicitly in library pigeon.v, as mentioned in Section 6.9, in order to have decidability for the type of non-terminal symbols.

Besides that, some predicates used in the formalization are not constructive, in the sense that they do not explain how to build the objects that validates the corresponding propositions. An example is the predicate `empty`, used to assert that a symbol `s` (non-terminal or terminal) eventually derives the empty string in grammar `g` (see Section 6.6.1):

```
Definition empty
(g: cfg terminal _) (s: non_terminal + terminal): Prop:=
derives g [s] [].
```

Such definitions do not have computational content, and thus prevent code extraction from the scripts. An alternative approach would be to use an inductive and more informative definition, such as the one used for predicate `unit` in Section 6.6.2:

```
Inductive unit
(terminal non_terminal : Type)
(g: cfg terminal non_terminal)
(a: non_terminal)
: non_terminal → Prop:=
| unit_rule:
  ∀ (b: non_terminal),
  rules g a [inl b] → unit g a b
| unit_trans:
  ∀ b c: non_terminal,
  unit g a b → unit g b c → unit g a c.
```

While the definition of `empty` is clearly declarative and very short and abstract, the definition of `unit` is more detailed and explains exactly how to determine whether two arbitrary non-terminal symbols satisfy the relation.

The same effect can be obtained for the `empty` predicate if it is replaced, for example, by the following definition:

```
Inductive empty (g: cfg non_terminal terminal):
(non_terminal + terminal) → Prop:=
| empty_b: ∀ left: non_terminal,
           rules g left [] → empty g (inl left)
| empty_i: ∀ left: non_terminal,
           ∀ right: sf,
           rules g left right →
```

```
(∀ s: (non_terminal + terminal),
In s right → empty g s) → empty g (inl left).
```

The same situation happens with the definitions of predicates `useful` and `accessible`, respectively in Section 6.6.3 and Section 6.6.4.

Except for these, the formalization can be considered constructive. This does not mean, however, that it can be used to extract certified programs, as explained next.

Set versus Prop

The decision of representing grammar rules as propositions (that is, inhabitants of the sort `Prop`) has the consequence that it prevents direct extraction of executable code from the formalization. Despite the efficiency issues that are associated with code extraction, it would surely be desirable, however, to be able to obtain certified algorithms for the operations considered in this work, such as grammar union, concatenation and closure, as well as grammar simplification and Chomsky Normal Form.

To achieve this, an alternative would be to represent the set P of rules as a member of type `list (non_terminal * sf)` instead (that is, an inhabitant of the sort `Set`). In this case, P would be represented as a list of rules, and each rule as a pair where the first element is a non-terminal symbol and the second is a list of terminal and non-terminal symbols. Then, the inductive definitions used to construct new grammars would have to be replaced by functions that construct the new grammars from the previous one. Finally, the desired lemmas and theorems would have to be proved on top of these functions. After that, the extraction of certified programs from these functions could be obtained directly using Coq´s appropriate facilities.

All this, however, would have changed the declarative approach of the present work into an algorithmic approach, by creating functions that generate new grammars with the desired properties. On the other hand, the purely logical approach adopted was considered more appealing, since it maps directly from the textbooks, and thus was selected as the choice for the present formalization. Anyway, this is a possibility that should still be considered and can certainly be achieved with some effort.

Inductive versus Fixpoint

In general, a formalization can either (i) make heavy use of inductive definitions, or (ii) of fixpoint definitions (recursive functions) or (iii) a mixture of both. In the first case we get a declarative and thus more “logical” formalization which prevents code extraction, in the second we have lots of computational content (at the expense of having to build proofs of functions) and in the third we can obtain all these benefits in a balanced way.

The present formalization relies extensively on the use of inductive definitions, with great impact in the way the corresponding proofs were constructed (declarative in most cases). An alternative strategy would be to have a better balance of inductive and fixpoint definitions (recursive functions), in order to add computational content to the formalization and enable code extraction to obtain, for example, executable programs that perform transformations such as grammar closure, grammar simplification and Chomsky normalization.

A total of 40 inductive definitions have been used. They embed, in a certain way, the algorithms that make the desired mappings and constitute the basis of the whole development. These definitions were used mostly in the multiple instances of the `derives` predicate, on all definitions of new non-terminal symbols, on all definitions of new grammar rules and also on some definitions and predicates related to binary trees. On the other hand, only 20 recursive functions (Fixpoint definitions) have been used in the formalization.

As an example, the relatively complex inductive definitions used to specify the new grammar rules for closure, simplification and CNF could have been substituted by corresponding recursive functions that would explicitly execute the algorithms embedded in the definitions, thereby effectively constructing the new sets of rules instead of just declaring which are these rules. This approach would bring the formalization closer to the computer universe with the further advantage of allowing the extraction of certified programs. Naturally, other changes would have to follow, such as to the definition of context-free grammar (`cfg`). Also, proofs over inductive definitions would have to be replaced by proofs over fixpoint definitions (proofs of recursive functions).

Finiteness of the context-free grammar

Context-free grammars demand that the non-terminal and terminal symbol sets be finite. In the formalization, however, the non-terminal and terminal symbol sets, used to define grammars, can be a regular Coq type, meaning that they can have either a finite or an infinite number of inhabitants.

Thus, we should somehow avoid the use of a type with an infinte number of inhabitants in both cases. This was not considered an issue in the beginning of the formalization, and for this reason the representation choice for these sets was left as general types until later consideration. As an example, the idea was to use inductive types with a finite number of constructors only, as in:

```
Inductive non_terminal_1: Type:=
| S: non_terminal_1
| X: non_terminal_1
| Y: non_terminal_1
| Z: non_terminal_1.
```

for $N = \{S,X,Y,Z\}$ (with four inhabitants). However, there were no restrictions for using a definition like the following one (with an infinite number of inhabitants):

```
Inductive non_terminal_2: Type:=
| S: non_terminal_2
| X: non_terminal_2 → non_terminal_2.
```

for an hypotetical $N = \{S, XS, XXS, XXXS, ...\}$. This was obviously a problem, since the results obtained could not be claimed as valid for general context-free grammars, as they are defined in the literature.

The problem, however, extends further. Even if it were possibile to restrict the non-terminal and terminal sets to be finite, it would still be possible to define a grammar with an infinite number of rules because there was no upper limit on the length of the righ-hand side of the rules of a grammar (remember that `rules` is defined as `non_terminal → sf → Prop`, where `sf` can be any sentential form). This was also against the definition of context-free grammars, and a solution for this problem had to be implemented somewhow. This is indeed what happened, however in a moment when the formalization was in an advanced stage.

In order to preserve what had already been done, with minimum changes and extra effort, the solution was to introduce a predicate (`rules_finite`) in the definition of a context-free grammar (`cfg`). The predicate and its use are presented below:

```
Definition rules_finite_def (ss: non_terminal)
                            (rules: non_terminal → sf → Prop)
                            (n: nat)
                            (ntl: list non_terminal)
                            (tl: list terminal) :=
In ss ntl ∧
(∀ left: non_terminal,
 ∀ right: list (non_terminal + terminal),
 rules left right →
 length right ≤ n ∧
 In left ntl ∧
 (∀ s : non_terminal, In (inl s) right → In s ntl) ∧
 (∀ s : terminal, In (inr s) right → In s tl)).

Record cfg: Type:= {
start_symbol: non_terminal;
rules: non_terminal → sf → Prop;
rules_finite: ∃ n: nat,
              ∃ ntl: nlist,
              ∃ tl: tlist,
              rules_finite_def start_symbol rules n ntl tl
}.
```

This way, every new grammar that is defined must be accompanied by a proof that the corresponding statement `rules_finite` is true. For this, it is necessary to show that (i) there exists a natural number for this grammar such that the right-hand side of all rules of the grammar have a length that is equal or less than this number; (ii) there exists a list of non-terminal symbols for this grammar such that every symbol used in a rule of the grammar is part of this list and (iii) there exists a list of terminal symbols for this grammar such that every symbol used in a rule of the grammar in part of this list. The idea of using lists, in this case, is to ensure that the corresponding types have a finite number of inhabitants.

As a consequence of this condition, every new grammar defined in the formalization had to comply with the new predicate, which led to extra and long proofs in many cases. A simpler solution would be, of course, to introduce lists in the definition of the rules fom the beginning. This, however, would have a much higher impact in the formalization, demanding changes in most lemmas already proved. For this reason, the solution with an extra field in the record definition was adopted. As a result, every new grammar defined in the formalization is guaranteed to have a finite number of non-terminal symbols, a finite number of terminal symbols and a finite number of rules.

Variants of inductive predicate definitions

Most cases of using different design choices in the formalization happened with inductive definitions. The definition of the predicate `derives`, for example, is very representative because the initial definition motivated the inclusion of a few variants, each of which suited for a different use and each of them equivalent to all the others. As a consequence, seven different notions of a derivation in a context-free grammar were defined, namely `derives`, `derives2` etc, until `derives7`.

This allowed for the selection of the definition best suited for each proof. The statements to be proved, however, all use to the original `definition` in order to preserve readability. Then, during a proof, the original definition is substituted by an alternative definition, as long it is easier or more natural to proceed and complete the proof this way.

This strategy, however, needed proofs of the equivalence between these definitions in order to permit the desired substitution. This was accomplished by a number of lemmas dedicated specifically to prove, directly or indirectly, the equivalence of the definitions. Section 6.3 shows some of these definitions, describes some of the lemmas that prove their equivalence and presents different proofs for the same statement using different definitions.

A similar approach was used with other inductive definitions in the formalization as well.

Use of syntax trees in proofs

Trees are a fundamental structure in context-free language theory. However, they were introduced late in the formalization, and still only in the binary variant used to prove the existence of a Chomsky Normal Form.

General syntax trees (with any number of children) were considered for formalization, as well as their relation to derivation sequences. Specifically, the objective was to prove that every sentence of a language defined by a context-free grammar can be represented by a syntax tree, and the other way around. However, time limitations did not allow for the conclusion of the proof of these equivalence lemmas, and thus syntax trees still cannot be used to represent derivations (except for binary trees and derivations in a CNF grammar).

If general syntax trees were introduced earlier in the formalization, it would probably have benefited a lot from them in all phases. Instead of using the `derives` predicate (and their variants) to construct most of the proofs, the use of syntax trees would have permited the construction of simpler and more concise proofs by simple induction on the tree structure.

Support libraries

Nine libraries of the Standard Coq Library (INRIA, 2016) were used in the formalization:

1. `Classical_Prop`, for classical propositional logic;
   (used only in library pigeon.v);
2. `Compare_dec`, for comparison of natural numbers;
3. `Decidable`, for properties of decidable propositions;
4. `Even`, for definitions and properties of even and odd numbers;
5. `List`, for definitions, functions and properties of generic lists;
6. `NPeano`, for properties of natural numbers;
7. `NZPow`, for properties of the power operator;
8. `Omega`, for automatic decision procedure for Presburger arithmetic;
   (implements tactic `omega`)
9. `Ring`, for equations upon polynomial expressions of a ring (or semi-ring) structure;
   (implements tactic `ring`)

Eight out of the eighteen libraries that are part of the formalization (allrules.v, cfg.v, cfl.v, misc_arith.v, misc_list.v, misc_logic.v, pigeon.v and trees.v) refer to generic results that were used throughout the work and could also be used easily in other formalizations. They correspond to roughly half of the formalization effort (in number of lines) and only two of them (cfg.v and trees.v) account for 37% of the total number of lines or 35% of the total number of lemmas and theorems. Thus, they deserve some considerations of their own. Libraries misc_arith.v and misc_list.v, in particular, extend, respectively, libraries `NPeano` and `List` of the Coq Standard Library.

First of all, it is important to note that the development of these eight libraries was not planned from the beginning. On the contrary, the idea of creating such libraries started at the same time that the first generic results (on arithmetic, lists, context-free grammars etc) were neeed to prove the core lemmas of our formalization. This is more or less straighforward for results on arithmetic and lists, but not for context-free grammars and the relation between binary trees and CNF grammars for example. Differently from the two first cases, for the two latter cases it was very important to have a good background on the theory being formalized, in order to promptly identify, state, prove and isolate the lemmas and theorems that referred to generic results. This way, we could not only build the corresponding generic libraries, but also use the generic results to prove the specific results we were pursuing, while maintaining the elegance and legibility of the proof scripts. The construction of the generic libraries lasted until the end of the work, and can be viewed as a simultaneous activity to the formalization of the specific results.

Nevertheless, it could still be argued that part of results contained in these generic libraries could eventually be found in third-party previously developed libraries, without having to spend time and effort to construct them. For arithmetic and lists we don´t believe that this is the case, since extensive search was sistematically done in the standard libraries of Coq before new proofs were created. For context-free grammars and the relation of CNF grammars to binary trees, there is no similar work that could be used. This could be the case, perhaps, for binary trees and the pigeonhole principle. However, even so it was considered more effective to state and proof the results that were needed, when they were needed, than making time-consuming searches and adaptations in the standard and user contributed libraries of Coq. Anyway, the libraries of SSReflect are much more complete than those of Coq, and thus represent an additional motivation for searching and using them when we migrate our formalization to SSRefelct.

Previous works

The present work did not take advantage of previous works on the subject. Some possibilities for this would have been (see Chapter 4) the works of Filliâtre and Courant on regular and context-free languages, and of Doczkal, Kaiser and Smolka on regular languages, all of which have been done in the Coq proof assistant. Another possibility would have been the work of Norrish and Barthwal on context-free languages in HOL4. We shall thus discuss the reasons why this was not the case in all situations, and explain our decision of starting from scratch (except for some of the standard libraries).

From these three works, the one by Filliâtre and Courant would probably be the one with the highest likelihood of being adopted for our own work, due to the use of Coq and the formalization of some context-free language theory. However, three main aspects were considered in order to abandon this idea. The first is the lack of documentation on the context-free language theory part of their formalization (it exists only for one part of their work, and it is for the formalization of Kleene's theorem). The second is the statement made by Mr. Filliâtre in a personal correspondence dated September 25th, 2013:

"My own 1994 proof is definitely outdated and naive (Coq was young by that time and I was as well :-)). I'm sure you'll be able to do much a better work, using recent features of Coq in particular."

His formalization was indeed 19 years old in 2013, and Coq was only 5 years old. Since then, Coq evolved quite a lot and justifies his claims in this aspect. The third and final reason is not the most important, however was considered as an additional problem: the use of the french language in the names and comments of their formalization, with which we are not familiar. All considered, we decided not to continue where they stopped.

The second option would be the work of Doczkal, Kaiser and Smolka on regular languages. Despite being a very nice and complete work, we identified two situations that did not meet our expectations. The first is the fact that they made extensive of the SSReflect plugin for Coq. Although it is a plugin, the point is that SSReflect implements virtually a new language with a new set of commands and tactics, with different syntax and semantics when compared to Coq. Still, parts of Coq are also used in SSReflect. Besides that, SSReflect adopts a different approach towards proof construction, namely the *proof by reflection*. Thus, we would face the problem of not only having to learn Coq, but also SSReflect and its own proving methodology, something that the time limitations of our own work would not permit. The second is that they had the explicit objective of creating an extremely concise formalization, which indeed happened. This was never an objective in our case: the length of our formalization did not suffer from any constraints besides being correct and readable.

Third and final, the work of Norrish and Barthwal, despite being the most complete on context-free languages up to date, was formalized in HOL4, which is a higher-order logic and not a type theory (as in Coq). Thus, although it refers to the formalization of the same context-free language theory, it is a completely different approach towards a similar objective.

On the other hand, it is worth mentioning that to build our formalization from scratch, without any previous knowledge of Coq or any other proof assistant, and no previous experience with formal mathematics, was a hard and complex task filled with lots of challenges. All this, however, contributed to a better training and understanding of the formalization process and Coq, a byproduct that we consider relevant in the scope of the present work.

6.11.4 Statement and proof of the Pumping Lemma

It is common, in the literature, to find texts that list, among the many benefits of using interactive proof assistants, the deeper insight into the nature of the proofs that the user can acquire, and also, in some cases, the discovery of unnoticed errors in textbooks and informal proofs.

A similar situation happened during, for example, the proof of the Pumping Lemma for context-free languages (Section 6.9). Although the notion of a *path* inside a tree was used in the informal proof (RAMOS; NETO; VEGA, 2009), and translated in the formalization into the notion of a `bpath`, the need for a unique identification of this path was not clear from the informal proof. When introduced as a `bcode`, however, it allowed to overcome many of the problems faced in the formalization up to that moment.

The statement of the Pumping Lemma itself was perceived by the author with some extra information that is not available from textbooks. Let us recall the classical statement of the lemma, presented in Section 6.9 (`pumping_lemma`):

$$\forall\, \mathscr{L}, (\text{cfl}\ \mathscr{L}) \to \exists\, n\,|$$
$$\forall\, \alpha, (\alpha \in \mathscr{L}) \wedge (|\alpha| \geq n) \to$$
$$\exists\, u,v,w,x,y \in \Sigma^*\,|\,(\alpha = uvwxy) \wedge (|vx| \geq 1) \wedge (|vwx| \leq n) \wedge$$
$$\forall\, i, uv^iwx^iy \in \mathscr{L}$$

It contains a statement about the length of vx, but no statement about the length of uy. The development of the proof, however, shows clearly that the same condition applies to this string. Thus, the more complete version of the lemma (the one that was formalized) brings an extra insight:

$$\forall\, \mathscr{L}, (\text{cfl}\ \mathscr{L}) \to \exists\, n\,|$$
$$\forall\, \alpha, (\alpha \in \mathscr{L}) \wedge (|\alpha| \geq n) \to$$
$$\exists\, u,v,w,x,y \in \Sigma^*\,|\,(\alpha = uvwxy) \wedge (|vx| \geq 1) \wedge (|uy| \geq 1) \wedge (|vwx| \leq n) \wedge$$
$$\forall\, i, uv^iwx^iy \in \mathscr{L}$$

More interestingly, however, is to note that a variant of the Pumping Lemma, using a smaller value of n, has also been proved. This result is based on the informal proof published in Ramos, Neto and Vega (2009), and uses $n = 2^{k-1} + 1$ instead of $n = 2^k$ deployed in other textbooks (k is the number of non-terminal symbols in the CNF grammar). Since the proof needs a binary tree of height at least $k+1$ in order to proceed, and since trees of height i have as frontier strings of length maximum 2^{i-1}, it is clear that it is also possible to consider strings of length equal to or greater than $2^{k-1}+1$ (and not only of length equal to or greater than 2^k) in order to have the corresponding binary tree with height equal to or higher than $k+1$. This way, two slightly different proofs of the Pumping Lemma have been produced: one with $n = 2^k$ (`pumping_lemma`) and the other with $n = 2^{k-1} + 1$ (`pumping_lemma_v2`).

The statement of (`pumping_lemma_v2`) becomes:

$$\forall\, \mathscr{L}, (\text{cfl}\ \mathscr{L}) \to \exists\, n\,|$$
$$\forall\, \alpha, (\alpha \in \mathscr{L}) \wedge (|\alpha| \geq n) \to$$
$$\exists\, u,v,w,x,y \in \Sigma^*\,|\,(\alpha = uvwxy) \wedge (|vx| \geq 1) \wedge (|vwx| \leq (n-1) * 2) \wedge$$

$$\forall\, i, uv^iwx^iy \in \mathscr{L}$$

When compared to the statement of the original version (`pumping_lemma`), the difference is that $|vwx| \leq n$ has been replaced with $|vwx| \leq (n-1)*2$.

It is also important to note that the formal development of this variant permitted the discovery of an error in the informal proof contained in Ramos, Neto and Vega (2009). Indeed, in the statement of theorem 4.17 in page 359, $|vwx| \leq n$ should be replaced by $|vwx| \leq (n-1)*2$. Also in page 361, for the sake of clarity, it should be stressed that, since $2 \leq |vwx| \leq 2^k$ and $n = 2^{k-1}+1$, then $2^k = (n-1)*2$ and thus $|vwx| \leq (n-1)*2$ as specified in the statement of `pumping_lemma_v2`.

This variant of the Pumping Lemma is not the first formulation ever published that uses a value of n that is smaller than 2^k. It was used in the proof contained in Hopcroft and Ullman (1969, p. 57), where the corresponding statement (Theorem 4.7, "*uvwxy* Theorem") refers to two different constants p and q, with respectively values 2^{k-1} and 2^k. The same happens in the original proof by Hillel, Perles and Shamir (BAR-HILLEL, 1964, p. 130, Theorem 4.1) and in the book by Ginsburg (GINSBURG, 1966). The proof of Ramos, Neto and Vega (2009) was however developed independently of these, and this is the first formalization of this result as far as we are aware of.

6.11.5 Comparison

The results achieved by the present work can be compared with the ones obatined by Norrish and Barhwal, and also by Firsov and Uustalu. For this purpose, we add a third column to Table 4.1 of Section 4.3 in order to build Table 6.3:

Table 6.3: Context-free language theory formalizations of Norrish & Barthwal, Firsov & Uustalu and Ramos

	Norrish & Barthwal	**Firsov & Uustalu**	**Ramos**
Proof assistant	`HOL4`	`Agda`	`Coq`
Closure	✓	×	✓
Simplification	✓	*only empty and unit rules*	✓
CNF	✓	✓	✓
GNF	✓	×	×
PDA	✓	×	×
PL	✓	×	✓

The formalization of Norrish and Barthwal extends our own formalization with pushdown automata and the Greibach Normal Form and remains the most complete up to date. When compared to the newer results of Firsov and Uustalu, however, our formalization adds closure properties, the elimination of useless and inaccessible symbols and the Pumping Lemma as well.

Table 6.3 is a summary of the main formalizations of context-free language theory that have been done in the last five years. It shows that the formalization by Norrish and Barthwal and the present one are the most complete up to date, and for this reason both can be considered as important initiatives towards a first and complete formalization of the context-free language theory in the Coq and HOL4 proof assistants.

The main contributions of the present work should however also be clear at this point: it is at the same time (i) the first ever comprehensive formalization of an important subset of the context-free language theory in the Coq proof assistant and (ii) the first ever formalization of the Pumping Lemma for context-free languages in the Coq proof assistant. These results also hold when considering the use of a Type Theory as the underlying calculus of the formalization process.

In what follows, we take a more detailed look into the nature and the characteristics of the works of Barthwal and Firsov, relating them to our own work as much as possible.

Aditi Barthwal

The PhD thesis "A formalisation of the theory of context-free languages in higher order logic" of Aditi Barthwal (BARTHWAL, 2010) is the result of a research conducted at the Computer Sciences Laboratory of the Australian National University under the supervision of Dr. Michael Norrish (one of the developers of the HOL4 system) and was concluded in December of 2010. The papers Barthwal and Norrish (2010a), Barthwal and Norrish (2010b), Barthwal and Norrish (2014) contain parts of the work published in the thesis.

Besides the formalization of the theory, the author has also developed an application of the theory, consisting of a certified SLR parser generator, including proofs of its soundness and completeness and an executable version in SML (Standard ML).

The thesis includes an introduction (Chapter 1), three chapters on the formalization of the theory (Chapter 2 for context-free grammars, Chapter 3 for pushdown automata and Chapter 4 for properties of context-free languages), one chapter on the application (Chapter 4, for SLR parsing), considerations about the formalization process (Chapter 5) and ends with conclusions and future work in Chapter 6. We will not discuss or review on topics that do not have a correspondence with our own work (such as, for example, GNF and parsing).

Chapter 1 presents a concise yet very nice introduction to automated reasoning, theorem provers and the nature and scope of her work. She answers to interesting questions posed by herself, such such as *What makes a good theorem prover?*, *What are theorem provers good for?* and *So how good are the theorem provers with reasoning?* before making a very short introduction to HOL4 and then presenting the importance and scope of the work. For the scope, besides the items listed in Table 6.3, it should also be credited with the formalization of closure properties of CFLs under homomorphic substitution and inverse homomorphism, and the application of the theory in parser verification. The text of the thesis is incorrect, however, by stating that closure under intesection is also formalized (page 13), since it has been proved that

CFLs are not closed under this operator (SUDKAMP, 2006). As expected, no formalization for it exists in Chapter 4.

Chapters 2, 3 and 4 follow strictly the textbook by Hopcroft and Ullman (respectively Chapter 4, 5 and 6 of Hopcroft and Ullman (1979), and lemmas and theorem numbers of the book are explicitly referred to in the thesis.

Her statement about the objectives of the work coincides with ours:

> “The main contribution of this thesis is the infrastructure: definitions and proofs relating to context-free grammars and pushdown automata. These well-established fields are traditionally ignored as a possible subject for mechanisation. In some sense the proofs are seen as too well-understood to justify the large effort formalisation requires. On the contrary, the formalisation of these theories is vital because they form the basis for further works such as compiler proofs. The importance of CFGs in their many forms goes beyond their use in compiler theory. They are increasingly being used in problems such as pattern matching and natural language processing.
>
> The other area our work addresses is the goal of proof certification. In order for theorem provers to be used more extensively for guaranteeing proof correctness in various fields, we need to provide an extensive background of theories so that further automation efforts can start at a higher level of abstraction.”

We are happy to know this, specially when she mentions that such a work can have an impact in increasing the usage of proof assistants by making a new theory available.

Chapter 2 presents the definitions and main results related to context-free grammar simplification and normalization. A symbol can be of two different types (non-terminals and terminals), a rule pairs a non-terminal symbol and a list of non-terminal and terminal symbols (respectively the left and right-hand side of a rule) and a grammar is a list of rules paired with a non-teminal symbol (the start symbolf of the grammar).

Thus, the definitions used by Barthwal are quite similar to the ones used on our work, except for the fact that she uses lists and our approach is declarative (using the sort `Prop`). This eliminates the need for proving the finiteness of the sets Σ, N and P, something that we had to consider. On the other hand, she parametrizes these definitions over the types of the non-terminal and terminal symbol types, just as we did.

Following the natural order, Barthwal introduces a definition for predicate `derives`. It is very similar to our own, except for the fact that we used a constructor (`derives_step`) to obtain the closure of the operation, while she only defines a direct derivation (application of a single rule) and then uses a closure operator of the HOL4 system to obtain non-trivial derivations. After a few more definitions (such as the language defined by a grammar, leftmost derivation etc), she starts the discussion of grammar simplification.

By using a terminology the is different from our case, she first presents the elimination of useless symbols as a combination of “non-generating” (“useless” in our case) and non-reachable (“inaccessible in our case) symbols elimination. In both cases, the main result comes from theorems that assert the equivalence of the languages generated by the original grammar and the grammar without non-generating (or non-reachable) symbols. For grammar construction, the start symbol is the same and the rules are obtained by appropriate transformations that identify and select the rules of the final grammar. The elimination of empty and unit rules follows a similar strategy.

The whole simplification is presented very concisely in only 5 pages, and is restricted to the ways that the modified grammars are constructed, and also the statements that assert the preservation of the original language after the transformations are applied. There is no discussion about the proof strategies that were used to confirm the validity of these statements. The construction methods are similar to the ones that we have used, except that Barthwal requires that the grammar to have empty rules eliminated can not generate the empty string. This has the impact that the Chomsky normalization will only be possible for grammars in the same situation. This is something that we considered from the beginning, and is thus acceptable in our formalization. Also, we did not find any comments about the order in which these transformations can be applied, which, if not well selected, can surely result in grammars that do not satisfy all conditions of the simplification.

For the Chomsky Normal Form, the author starts from the original transformation algorithm of the textbook, which is first represented in HOL4 and then proved to be correct. The algorithm works in two steps, first by substituting terminals by new non-terminals, and then by splitting the rules with three or more symbols in the right-hand side. As mentioned before, the formalization has the drawback that the original grammar can not generate the empty string. This approach is radically different from the one we have used, which uses an inductive definition to simultaneously state all the rules that belong to the new grammar. The transformation algorithm is not explicit in our case and is, in a certain way, deeply hidden in this definition. While her approach is clearly constructive, based in classical textbook proofs, ours is essentially declarative and, because of this, original and, in our opinion, more elegant.

As a preparation to the Pumping Lemma, the author discusses the importance of generic parse trees (not necessarily binary) and formalizes its definition and some properties. Also, the relationship between valid parse trees and derivations in a grammar, and the notion of subtrees along with some functions.

For the Pumping Lemma itself, the strategy adopted in the proof is the same as ours. She uses the classical statement, however considering that the grammar is already in CNF, which means that the proof is valid only for grammars that do not generate the empty string. In comparison, our statement accepts any context-free grammar, we have an aditional clause to the statement (which is not present in the classical statement) and we also proof an alternative version of lemma (using a smaller value of n). Basically, she proceeds by showng that sufficiently long sentences have parse trees with a repeated non-terminal in some path, and then by showing that a subtree can be “pumped” in a way that the statement can be proved. She ends with some brief considerations about the complexity of the formalization in comparison to the informal proof. Still, the proof presented in the thesis is concise, using a mixture of informal arguments and HOL lemmas previously introduced.

Two errors were found, respectively in page 73 ("if $|z|$ is larger than 2^k" should read "if $|z|$ is larger than or equal to 2^k") and page 75 ("Since we pick z that is sufficiently long, i.e. $|z| \geq 2^{k-1}$" should read "Since we pick z that is sufficiently long, i.e. $|z| \geq 2^k$"). Of course the statement of page 75 is also correct if $|z| \geq 2^{k-1}+1$ is used instead of $|z| \geq 2^k$, but this does not seem to be the intention of the author (this corresponds, however, to the second version of the PL formalized in our work).

The use of generic parse trees in the proof and CNF grammars in the statement of the PL comes probably from the fact that they are also used in the SLR parser verification. Otherwise, as in our case, the simpler binary trees would suffice.

For closure properties, she starts with an interesting and useful discussion about changing the names of the non-terminal symbols in a grammar, in such a way that the language generated is preserved. This result, which involves the formalization of the procedure and a proof of its correctness, is important since it allows any two context-free grammars to be combined by a closure operator, while ensuring that there will be no name space collision. In contrast, our work does not offer a solution for this problem yet. Despite the fact that the types of the non-terminal symbols are defined independently, Coq does not permit that constructors (the values of our non-terminals) have the same name in different type definitions. For this reason, this is something that needs to be worked out in the future.

The construction of grammars for closure under union, concatenation and Kleene star is very similar to ours, as well as the proofs of their correctness and completeness. The presentation is however short (less than a page per operator) and there is no discussion about language closure (as in our work), only grammar closure.

Besides the formalization itself, another major contribution of this work are the considerations about the nature and causes of the complexity involved in the formalization of informal (textbook) proofs. As Barthwal states in the introduction:

> "Our main impetus with this work was to investigate the difficulties of formalisation and verification."

The result is the long, deep and insightful discussion presented in Chapter 6. She starts by assessing the subject of readability versus executability, curiosly with an example similar to the one we have used in Section 6.11.3 (the definition of the predicate `empty` in our formalization and the definition of the predicate `nullable` in her work). After that she addresses some other design choices, with special emphasis to the proof of the Pumping Lemma, which resulted first in a very long and complex proof using derivation lists (equivalent to our `sflist` predicate) and finally in a second and much simpler proof using trees (with about one fifth of the size). This, in a certain way, supports our decision of using trees as well. Another interesting discussion is the one about using lists instead of sets for representing the rules of a grammar, as it avoided the necessity of making various finiteness proofs (which we had to do since we have not used lists for this purpose).

The thesis is full of references to the complexity of the formalization activity, to the insights and great number of details that have to be added to textbook proofs and to the way that formal proofs expand in size very rapidly, an impression that we share by making ours her own words:

> "To a person who is not from a theorem proving background, it may seem that the ratio of number of supporting lemmas to proofs is very high. This is typical of the automation process."

Coq and HOL share many characteristics in common, in particular both are typed systems that use higher order logics. Coq, however, is based on the very powerful Calculus of Inductive Constructions (and thus supports constructive logic) and includes dependent types, a feature that is not offered by HOL. HOL uses classical higher-order logic with axioms of infinity, extensionality and choice, and is based on simply typed lambda-calculus with polymorphic type variables. Both systems follow the LCF approach, a set of ideas about the design of proof assistants with the objective of assuring a high level of confidence in their operation (ZAMMIT, 1997; WIEDIJK, 2006).

In summary, despite the differences between the logics of Coq and HOL4, we find many similarities in the more important definitions and the main proof strategies used in both works. The differences perceived, on the other hand, are very enlightening and can be mutually benefitial. The two works can, nevertheless, be considered as important contributions to the area of mathematical formalization and to the communities of users of both proof assistants.

Denis Firsov

Denis Firsov is a PhD student at the Institute of Cybernetics at Tallinn University of Technology (TUT) in Estonia, under the supervision of professor Tarmo Uustalu. Together, they have published on certified parser generation for regular languages (FIRSOV; UUSTALU, 2013), certified CYK parser generation for context-free languages (FIRSOV; UUSTALU, 2014) and certified Chomsky normalization of context-free grammars (FIRSOV; UUSTALU, 2015). All results were obtained with the Agda proof assistant. We will focus our discussion on their work on the normalization of context-free grammars, since the other results are not in direct relationship to our work.

The work consists of a constructive formalization of the fact that every context-free grammar can be transformed into an equivalent one that is normalized according to the Chomsky Normal Form. The transformation is accomplished in four steps, namely (i) elimination of empty rules, (ii) elimination of unit rules, (iii) splitting the rules with three or more symbols in the right-hand side and (iv) substituting terminals by non-terminals. The authors have not considered the elimination of useless (or non-generating) and inaccessible (or non-reachable) symbols in their formalization (differently from what we and Barthwal have done), since this is not a strict requirement for obtaining a CNF grammar.

For 3 out of the 4 steps, the authors have followed the classical algorithms of the textbook by Hopcroft and Ullman (HOPCROFT; ULLMAN, 1979).

Differently from the work of Barthwal, and similar to ours, the formalization works on any grammar, including those that generate the empty string. Also, the authors mention the necessity of making the transformations in a certain order, so as to ensure that the correct result is obtained. The constructive nature of the formalization, in combination to the possibility of extracting computer code from the proofs in Agda, makes it possible to obtain certified (total) functions that convert parse trees from the original grammar into parse trees of the CNF grammar, and vice-versa. This is an importan result, and the authors claim that it can be used with their previous work on certified CYK parsing of context-free languages (which requires the input grammar to be in CNF), among other possibilities. The same transformation steps have been formalized in our work (and also Barthwal's), however without any kind of code extraction.

The definitions of the non-terminal and terminal symbols, rules and grammars are exactly as in the work of Barthwal: a type defines the set of non-terminal symbols, another type the set of terminal symbols, a rule is a pair of a non-terminal symbol and a list of non-terminal and/or terminal symbols, and a grammar is a list of rules together with a non-terminal symbol. Besides this, the authors define the (mutually inductive type) of a generic tree, used to represent derivations in the grammars.

Working with lists demands that the types of the elements used in these lists be decidable. For this reason, the authors explicitly require that the types of the non-terminal and terminal symbols have a decidable equality. This requirement was not necessary in our formalization, except in the proof of the pigeonhole principle used in the proof of the Pumping Lemma. Anyway, we have plans to eliminate this requirement in the future as well.

For the elimination of empty rules, the authors start by constructively defining the notion of a *nullable* symbol (one that derives the empty string), proceed by showing how to generate all subsequences from a given sequence of symbols (by eliminating combinations of nullable symbols), and then how to use these in order to build the new set of rules. Next they prove, by induction on the height of the derivation tree of the normalized grammar, that it has a corresponding derivation tree in the original grammar. For the converse, they prove, also by induction of the height of the tree, that a parse tree of a non-empty string in the original grammar can be converted to a parse tree in the normalized grammar. For the elimination of unit rules, the proofs are also on the height of the corresponding derivation trees, and uses a description of how unit rules with a particular right-hand side (formed by a single non-terminal symbol) can be eliminated.

For the splitting of long (> 2) right-hand sides and substitution of terminal symbols, the authors first define a function that creates fresh new non-terminal symbols, then use it in the function that splits long rules into an equivalent set of rules and finally apply the transformation that substitutes the terminal symbols, resulting in the final set of rules of the new grammar. Correctness proofs assert the the new grammar is in CNF and also relate derivation trees in the normalized grammar to derivation trees in the original grammar.

The paper ends by discussing the correct order in which the transformations must be applied and presenting proofs of correctness and completeness form the their combination. The correct sequence should start with the elimination of empty rules and continue with the elimination of unit rules (since the first can introduce unit rules, but not the other way around) as the authors explain in the text of Section 6.1. However, the representation on Agda of this sequence of transformations is incorrectly printed in the same section as `norm = norm-u ∘ norm-e ∘ norm-t ∘ norm-l`, where it should be `norm = norm-e ∘ norm-u ∘ norm-t ∘ norm-l` (`norm-e` is the function used in the elimination of empty rules, `norm-u` is for unit rules, `norm-l` is for long right-hand sides and `norm-t` for terminals).

The logic of Agda is much closer to Coq than HOL. Both are typed (and allow the definition of dependent types), use a higher order logic and support a constructive logic (WIEDIJK, 2006). The work discussed in this section is essentially constructive, with functions making the transformations required on lists and lists of lists. The proofs are then based on the definitions of these functions and trees. This approach is radically different from the one we have used in our own work where, despite being in a constrcuttive setting as well, we used a much more declarative approach towards the transformations, which affected the way that the proofs were created.

When comparing their work to Barthwal's, the authors mention that in the latter case the use of a non-constructive logic "gave the advantage of the power of extensional and classical reasoning, but also the significant drawback that it did not deliver actual functions for normalizing grammars or converting parse trees between grammars". The comparison to our own work would probably be that, although working in a constructive logic and using a powerful and very expressive logic, we did not take advantage of this in our formalization in order to obtain certified functions that implement the transformations. We agree and have indeed already discussed this topic before (Section 6.11.3), besides considering it for future work (Section 7.2).

Except for his manifested interest in "algorithms, mathematically certified software, type theory and semantics of programming languages" (taken from his home page <http://cs.ioc.ee/~denis/>), there is no further information as to whether the papers published by Firsov and Uustalu are part of Firsov's PhD project under development, as well as about the nature of the thesis.

7 Conclusions and Further Work

The present work describes the formalization of an important subset of context-free language theory in the Coq proof assistant. To achieve this, several steps had to be fulfilled, including (i) understanding of the characteristics, importance and benefits of mathematical formalization, specially in computer science, (ii) familiarization with the underlying mathematical theories, (iii) familiarization with the Coq proof assistant, (iv) review of the strategies used in the proofs of the main results of the context-free language theory and finally (iv) selection and adequation of representation and proof strategies in the Coq proof assistant in order the achieve the desired objectives. The result is an important set of libraries covering the main results of context-language theory, with more than 500 lemmas and theorems fully proved and checked.

This work is in line with the growing importance of formal developments in both theoretical and technological environments, and reflects the current trend in both mathematics and computer science towards formalization using interactive theorem provers. In this sense, it represents an important initiative in creating a local culture of a rigorous approach to doing mathematics and software development.

Besides the main contributions discussed in Section 7.1, it is expected that the present work will attract the attention of students, researchers and professionals that can benefit form the authors' acumulated expertise, and will contribute to the further development and application of our formalization in different areas of knowledge. The final benefits of this work, which has applications in mathematical formalization and certified software development, are less errors, increased rigor and faster verification of mathematical proofs, and increased safety, increased user satisfaction and higher quality in general in software development. Plans for future developments over the current framework are discussed in Section 7.2.

7.1 Contributions

The result of this work is the creation of a set of libraries with context-free language theory definitions, lemmas, theorems and related machine-checked proofs. They can be used for a variety of purposes, including to get a deeper understanding of the nature of the proofs for the main results of the theory, to obtain certified executable programs for some of the operations included in this work, to further develop the theory including objects not considered to this point, and to teach and experiment with formal language theory, proof theory, proof assistants and logic in general.

The present work represents a relevant achievement in the areas of formal language theory and mathematical formalization. As explained in Chapter 4, there is no record that the author is aware of, of a project with a similar scope in the Coq proof assistant covering the formalization of context-free language theory (there is, however, the formalization in HOL4 of Norrish and Barthwal, which extends the scope of the present work). The results published so far are restricted to parser certification and theoretical results in proof assistants other than Coq. This is not the case, however, for regular language theory, and in a certain sense the present work can be considered as an initiative that complements and extends that work with the objective of offering a complete framework for reasoning with the two most popular and important, from the practical point of view, language classes. It is also relevant from the mathematical perspective, since there is a clear trend towards increased and widespread usage of interactive proof assistants and the construction of libraries for fundamental theories.

Besides the libraries that are the more visible result of this work, there are other important contributions directly related to expertise on interactive theorem proving acquired during the development period:

- *Pioneering*
  The conclusion of this work represents three important achievements: (i) it is the first formalization of a coherent and complete subset of context-free language theory in the Coq proof assistant; (ii) it is the first formalization ever (in Coq) and the second formalization ever (in any proof assistant) of the Pumping Lemma for context-free languages (the first is from 2010 in HOL4, see Section 6.11), perhaps the most important result in this theory and (iii) it is the second comprehensive formalization of an important subset of the context-free language theory in any proof assistant (after Norrish and Barthwal's). More important, we bring formalization into an area which has relied so far mostly in informal arguments.

- *Reasoning about context-free language theory*
  The present formalization can be very helpful to get insight into the nature and behaviour of the objects of context-free language theory, as well on the proofs of their properties. Also, it should ease the development of representations for new and similar devices, and proofs for new results of the theory. Finally, the formalization represents the guarantee that the proofs are correct and that the remaining errors in the informal demonstrations, if any, could finally and definitely be reviewed and corrected.

- *Learning and experimenting in an educational environment*

Teachers, students and professionals can use the formalization to learn and experiment with the objects and concepts of context-free language theory in a software laboratory, where further practical observations and developments could be done independently. Also, the material could be deployed as the basis for a course on the theoretical foundations of computing, exploring simultaneously or independently not only language theory, but also logic, proof theory, type theory and models of computation. It could also serve as the basis for a course on formal mathematics, interactive theorem provers and Coq.

- *New projects and theories*
  The essence of formalization comes into light with the accomplishment of this project. This enables the application of similar principles to the formalization of other theories, and allow for the multiplication of the knowledge among students and colleagues. Considering the growing interest in formalization in recent years, this project can be considered as a good technical preparation for dealing with the challenges of theory and computer program developments of the future.

7.2 Further Work

The perspectives for the further development of this work are diverse and can be grouped in three differente areas: inclusion of new devices and results, code extraction and general enhancements of the libraries, all of which area discussed in the following sections.

7.2.1 New Devices and Results

Although extensive, the current formalization encompasses only, as pointed out before, context-free grammars and the main results of the theory that relate directly to them. Other devices and results that were not considered in this work, but whose formalization would definitely contribute to the constitution of a compehensive logical framework for context-free language theory, are:

- Pushdown automata, including: definition, equivalence of pushdown automata and context-free grammars; equivalence of empty stack and final state acceptance criteria; non-equivalence of the deterministic and the non-deterministic models;
- Elimination of left recursion in context-free grammars and Greibach Normal Form;
- Derivation trees, ambiguity and inherent ambiguity;
- Decidable problems for context-free languages (membership, emptyness and finiteness for example);

- Odgen's Lemma.

Context-free grammars are a popular device for defining context-free languages, however not the only one and not the most adequate for all cases. It is therefore natural to plan for the formalization of other representations, specially the pushdown automata with the two different acceptance criteria (empty stack and final state). Besides the formalization of the pushdown automata itself, and the notions of *configuration* and *movement*, this effort should be capable of proving the equivalence of context-free grammars and pushdown automata in respect to the class of languages that they are capable or defining. In particular, that every context-free language can be defined either by a context-free grammar or a pushdown automata, and also that from one representation it is always possibile to obtain the other. The equivalence of the acceptance criteira and the non-equivalence of the deterministic and the non-deterministic models are important results that should be achieved as well.

The current work formalizes only the Chomsky Normal Form. Another important and well-known normal form for context-free grammars is the Greibach Normal Form, which in turn depends on the fact that context-free languages can be generated by context-free grammars without left recursions (that is, the grammar contains no non-terminal symbols X such that $X \Rightarrow^{+} X\gamma$). Both results should be achieved and fit naturally in this formalization.

Derivation trees are represented in the current formalization, however only in the form of binary trees for Chomsy Normal Form grammars (`btree`). It would be desirable to formalize generic derivation trees (trees whose nodes could have any number of children), to state the equivalence of derivation trees and derivation sequences (as represented by predicate `derives`), to formalize the idea of an ambiguous grammar (the one for which at least one sentence of the language generated by it has two or more different derivation trees) and also the notion of an inherently ambiguous context-free language (the one for which all context-free grammars are ambiguous).

With the previous results formalized, it should not be difficult to prove some decidable problems for context-free languages, among them the membership problem (it is always possible to determine if a string belongs to the language generated by a context-free grammar), the emptyness problem (it is always possible to check whether the language generated by a context-free grammar is empty) and also the finiteness problem (it is always possible to check whether the language generated by a context-free grammar is finite).

The Pumping Lemma for context-free languages is not sufficient to precisely define a context-free language, since many non context-free languages also satisfy the property. Ogden's Lemma (HOPCROFT; ULLMAN, 1979), on the other hand, is a stronger version of the Pumping Lemma and, although also not sufficient to fully characterize the context-free languages, can be used to prove that certain languages are not context-free, where the traditional Pumping Lemma fails. It can be shown that the Pumping Lemma is a special case of Ogden's Lemma, and therefore it is natural to consider the formalization of this lemma as a result to be pursued in the future, in the scope of the present work.

Although there is a lot of effort in each of these formalization topics, it is also a fact that the acquired experience, together with the current framework, should somehow ease the task. This is, specially, a consequence of the similarities between some definitions (for example, the rules of a grammar and the transitions of a pushdown automata) and also of the extensive library of generic lemmas on context-free language theory and lists that has already been completed.

7.2.2 Code Extraction

Most of this work could be adapted in order to obtain certified (that is, provably correct) programs that implement the operations that were considered in it. Namely, the certified construction of programs that execute the union, concatenation and Kleene star of gramars and languages, the simplification of grammars and the conversion to CNF grammars, all of which are of high interest in academy and industry, could be obtained with some efffort. For this purpose, the code extraction facility of Coq could be used after the formalization had been modified in a proper way. This could be achieved mainly by using a less declarative and more constructive approach which, as seen before, produces a formalization with higher computational content and is thus more suited for code extraction. Constructive efficient specifications, however, imply higher complexity of proofs.

Besides that, code extraction finds an important application when it comes to the parsing of context-free languages and compiler construction in general. The implementation of certified parsers (that is, parsers that have been proved to strictly follow some formal language description, such as a grammar or an automaton) is an area of active research and some remarkable results have already been obtained (see Chapter 4). Thus, the present work could also evolve into this area, since the already constructed framework can be modified to accomodate new developments in this direction.

7.2.3 General Enhancements

The size and complexity of this development call for a long a detailed review of its contents, with the objective of:

- Creating a naming policy that can be used rename the various objects and better identify their nature and intended use;
- Eliminating unnecessary definitions and lemmas;
- Making a better grouping of related objetcs and thus a better structuring of the whole formalization;
- Simplifying some proof scripts;
- Commenting the scripts in order to provide a better understanding of their nature.

Besides that, the following improvements of the formalization are also welcome:

- Substitution of the classical logic proof of the pigeonhole principle (lemma `pigeon` in library pigeon.v) for a constructive version, thus making the whole formalization constructive;
- Rewriting of the contents of the trees.v library, in order to allow that all definitions and results be parametrized on any two types, one for the leafs and the other for the internal nodes of a `btree` (a binary tree). As it is currently, leafs and internal nodes must be, respectively, of types `terminal` and `non_terminal`. This change would permit the true generalization of the results included in the library, which might as well serve for different applications demanding the use of binary trees.

These are time consuming tasks that could not, unfortunately, due to time restrictions, be properly conducted over the formalization period. Nevertheless, they are of great importance as they will create the basis for its continued development, allowing for a better understanding, maintenance and evolution of its contents, in particular for those described in the previous sections.

Finally, it is important to note that the present formalization was created using plain Coq and its standard libraries. The SSReflect extension for Coq, developed during the formalization of the Four Color Theorem in 2004, and used extensively in it, has since then gained general acceptance and widespread use. Built on top of Coq, it introduces a slightly different notation and a very new and different way of constructing proofs. Exploring the principle known as *proof by reflection*, SSReflect allows the construction of shorter, simpler and more intuitive proofs that rely extensively on computation. Also, SSReflect is accompanied by a new set of libraries, the Mathematical Components Library, which represents a big improvement over the corresponding libraries of Coq. Thus, it is natural to consider the possibility of reviewing and rewriting the whole formalization with the idea of exploring the benefits of SSReflect and the Mathematical Components Library.

References

ACM SIGPLAN. *Programming Languages Software Award*. 2013. Available in: <http://www.sigplan.org/Awards/Software/>. Accessed on: 26. Oct. 2015.

AGDA. *Agda Home Page*. 2015. Available in: <http://wiki.portal.chalmers.se/agda>. Accessed on: 8. Dec. 2015.

ALMEIDA, J. C. B. et al. Partial derivative automata formalized in Coq. In: *Proceedings of the 15th International Conference on Implementation and Application of Automata*. Berlin, Heidelberg: Springer-Verlag, 2011. (CIAA'10), p. 59–68. ISBN 978-3-642-18097-2.

ALMEIDA, M.; MOREIRA, N.; REIS, R. Testing the equivalence of regular languages. *Journal of Automata, Languages and Combinatorics*, v. 15, n. 1/2, p. 7–25, 2010.

AMARILLI, A.; JEANMOUGIN, M. A proof of the pumping lemma for context-free languages through pushdown automata. *CoRR*, abs/1207.2819, 2012. Available in: <http://arxiv.org/abs/1207.2819>. Accessed on: 25. Jan. 2016.

ANDRONICK, J.; CHETALI, B.; LY, O. Using Coq to verify Java Card applet isolation properties. In: BASIN, D.; WOLFF, B. (Ed.). *Theorem Proving in Higher Order Logics*. [S.l.]: Springer Berlin Heidelberg, 2003, (Lecture Notes in Computer Science, v. 2758). p. 335–351. ISBN 978-3-540-40664-8.

ANONYMOUS. The QED Manifesto. In: *Proceedings of the 12th International Conference on Automated Deduction*. London, UK, UK: Springer-Verlag, 1994. (CADE-12), p. 238–251. ISBN 3-540-58156-1. Available in: <http://dl.acm.org/citation.cfm?id=648231.752823>. Accessed on: 26. Oct. 2015.

ASPERTI, A. A compact proof of decidability for regular expression equivalence. In: BERINGER, L.; FELTY, A. (Ed.). *Interactive Theorem Proving*. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7406). p. 283–298. ISBN 978-3-642-32346-1.

ASPERTI, A.; RICCIOTTI, W. Formalizing Turing machines. In: ONG, L.; QUEIROZ, R. (Ed.). *Logic, Language, Information and Computation*. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7456). p. 1–25. ISBN 978-3-642-32620-2.

ASPERTI, A. et al. The Matita interactive theorem prover. In: BJØRNER, N.; SOFRONIE-STOKKERMANS, V. (Ed.). *Automated Deduction – CADE-23*. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6803). p. 64–69. ISBN 978-3-642-22437-9.

AWODEY, S. Type theory and homotopy. In: *Epistemology versus Ontology*. [S.l.: s.n.], 2012. p. 183–201.

BAR-HILLEL, Y. *Language and information: selected essays on their theory and application*. [S.l.]: Addison-Wesley Pub. Co., 1964. (Addison-Wesley series in logic).

BAR-HILLEL, Y.; PERLES, M. A.; SHAMIR, E. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, n. 14, p. 143–172, 1961.

BARENDREGT, H. Introduction to generalized type systems. *Journal of Functional Programming*, v. 1, n. 2, p. 125–154, 1991.

BARENDREGT, H.; GEUVERS, H. Proof assistants using dependent type systems. In: ROBINSON, A.; VORONKOV, A. (Ed.). *Handbook of Automated Reasoning*. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 2001. cap. Proof-assistants Using Dependent Type Systems, p. 1149–1238. ISBN 0-444-50812-0.

BARTHE, G. et al. Formalization in Coq of the Java Card Virtual Machine. In: *Formal Techniques for Java Programs*. [S.l.: s.n.], 2000. p. 50–56. Available in: <http://www.cs.ru.nl/ftfjp/2000/ftfjp00.pdf>. Accessed on: 26. Oct. 2015.

BARTHE, G. et al. A formal executable semantics of the JavaCard platform. In: SANDS, D. (Ed.). *Programming Languages and Systems*. [S.l.]: Springer Berlin Heidelberg, 2001, (Lecture Notes in Computer Science, v. 2028). p. 302–319. ISBN 978-3-540-41862-7.

BARTHE, G. et al. A formal correspondence between offensive and defensive JavaCard virtual machines. In: CORTESI, A. (Ed.). *Verification, Model Checking, and Abstract Interpretation*. [S.l.]: Springer Berlin Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2294). p. 32–45. ISBN 978-3-540-43631-7.

BARTHWAL, A. *A formalisation of the theory of context-free languages in higher order logic*. Thesis (PhD) — The Australian National University, 2010. Available in: <https://digitalcollections.anu.edu.au/bitstream/1885/16399/1/Barthwal%20Thesis %202010.pdf>. Accessed on: 27. Nov. 2015.

BARTHWAL, A.; NORRISH, M. A formalisation of the normal forms of context-free grammars in HOL4. In: DAWAR, A.; VEITH, H. (Ed.). *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23–27, 2010. Proceedings*. [S.l.]: Springer, 2010. (Lecture Notes in Computer Science, v. 6247), p. 95–109.

BARTHWAL, A.; NORRISH, M. Mechanisation of PDA and grammar equivalence for context-free languages. In: DAWAR, A.; QUEIROZ, R. J. G. B. de (Ed.). *Logic, Language, Information and Computation, 17th International Workshop, WoLLIC 2010*. [S.l.: s.n.], 2010. (Lecture Notes in Computer Science, v. 6188), p. 125–135.

BARTHWAL, A.; NORRISH, M. A mechanisation of some context-free language theory in HOL4. *Journal of Computer and System Sciences (WoLLIC 2010 Special Issue, A. Dawar and R. de Queiroz, eds.)*, v. 80, n. 2, p. 346 – 362, 2014. ISSN 0022-0000.

BERGHOFER, S.; REITER, M. Formalizing the logic-automaton connection. In: BERGHOFER, S. et al. (Ed.). *Theorem Proving in Higher Order Logics*. [S.l.]: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5674). p. 147–163. ISBN 978-3-642-03358-2.

BERTOT, Y.; CASTÉRAN, P. *Interactive Theorem Proving and Program Development*. [S.l.]: Springer, 2004. ISBN 978-3-540-20854-9.

BONARSKA, E. *An Introduction to PC Mizar*. [S.l.]: Fondation Philippe Le Hodey, 1990. (Mizar Users Group).

BOYER, R.; MOORE, J. *A Computational Logic Handbook*. [S.l.]: Academic Press, 1998. (Academic Press international series in formal methods). ISBN 9780121229559.

BRAIBANT, T.; POUS, D. An efficient Coq tactic for deciding Kleene algebras. In: KAUFMANN, M.; PAULSON, L. (Ed.). *Interactive Theorem Proving*. [S.l.]: Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6172). p. 163–178. ISBN 978-3-642-14051-8.

BRAIBANT, T.; POUS, D. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, v. 8, n. 1:16, p. 1–42, 2012. Available in: <http://arxiv.org/pdf/1105.4537.pdf>. Accessed on: 26. Oct. 2015.

BRIAIS, S. *Finite Automata Theory in Coq*. 2008. Available in: <http://sbriais.free.fr/>. Accessed on: 28. Jul. 2014.

BROOKSHEAR, J. G. *Theory of Computation: Formal Languages, Automata, and Complexity*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN 0-8053-0143-7.

BROUWER, L. E. J. On the foundations of mathematics. In: HEYTING, A. (Ed.). *Philosophy and Foundations of Mathematics*. [S.l.]: North-Holland, 1975. p. 11 – 101. ISBN 978-0-7204-2076-0. English translation of the 1907 original in dutch.

BRUIJN, N. G. de. AUTOMATH, a language for mathematics. In: SIEKMANN, J. H.; WRIGHTSON, G. (Ed.). *Automation of Reasoning*. [S.l.]: Springer Berlin Heidelberg, 1983, (Symbolic Computation). p. 159–200. ISBN 978-3-642-81957-5.

BRZOZOWSKI, J. A. Derivatives of regular expressions. *J. ACM*, ACM, New York, NY, USA, v. 11, n. 4, p. 481–494, out. 1964. ISSN 0004-5411. Available in: <http://doi.acm.org/10.1145/321239.321249>. Accessed on: 26. Oct. 2015.

CHLIPALA, A. *Certified Programming with Dependent Types*. 2015. Available in: <http://adam.chlipala.net/cpdt/>. Accessed on: 26. Oct. 2015.

CHOMSKY, A. N. Three models for the description of language. *Information Theory, IRE Transactions on*, v. 2, n. 3, p. 113–124, September 1956. ISSN 0096-1000.

CHOMSKY, A. N. On certain formal properties of grammar. *Information and Control*, v. 2, p. 137–167, 1959.

CHURCH, A. A set of postulates for the foundation of logic part i. *Annals of Mathematics*, v. 33, n. 2, p. 346–366, 1932.

CHURCH, A. A set of postulates for the foundation of logic part ii. *Annals of Mathematics*, v. 34, n. 2, p. 839–864, 1933.

CHURCH, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, v. 58, n. 2, p. 345–363, April 1936. ISSN 00029327.

CHURCH, A. A formulation of the simple theory of types. *Journal of Symbolic Logic*, v. 5, p. 56–68, 6 1940. ISSN 1943-5886.

COMPCERT. *Compiler Certification*. 2015. Available in: <http://compcert.inria.fr/>. Accessed on: 26. Oct. 2015.

CONSTABLE, R. L. et al. *Implementing Mathematics with the Nuprl Proof Development System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986. ISBN 0-13-451832-2.

CONSTABLE, R. L. et al. Constructively formalizing automata theory. In: *Proof, Language and Interaction: Essays in Honour of Robert Milner*. [S.l.: s.n.], 1997.

COQUAND, T.; HUET, G. *The Calculus of Constructions*. 1986. [Rapports de Recherche nº530, INRIA].

COQUAND, T.; SILES, V. A decision procedure for regular expression equivalence in type theory. In: JOUANNAUD, J.-P.; SHAO, Z. (Ed.). *Certified Programs and Proofs*. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 7086). p. 119–134. ISBN 978-3-642-25378-2.

CURRY, H. B. Functionality in Combinatory Logic. In: *Proceedings of the National Academy of Sciences of the United States of America*. [S.l.: s.n.], 1934. v. 20, n. 11, p. 584–590. ISSN 0027-8424.

CURRY, H. B.; FEYS, R. *Combinatory Logic, Volume I*. [S.l.]: North-Holland, 1958.

DALEN, D. van. *Logic and Structure*. [S.l.]: Springer-Verlag, 2008. ISBN 978-3-540-20879-2.

DAVIS, M. D.; SIGAL, R.; WEYUKER, E. J. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. 2nd. ed. San Diego, CA, USA: Academic Press Professional, Inc., 1994. ISBN 0-12-206382-1.

DENNING, P. J.; DENNIS, J. B.; QUALITZ, J. E. *Machines, Languages and Computation*. [S.l.]: Prentice-Hall, 1978. ISBN 978-0135422588.

DOCZKAL, C.; KAISER, J.-O.; SMOLKA, G. A constructive theory of regular languages in Coq. In: GONTHIER, G.; NORRISH, M. (Ed.). *Certified Programs and Proofs*. [S.l.]: Springer International Publishing, 2013, (Lecture Notes in Computer Science, v. 8307). p. 82–97. ISBN 978-3-319-03544-4.

FILLIÂTRE, J.-C. *Finite Automata Theory in Coq: A constructive proof of Kleene's theorem*. [S.l.], 1997. Available in: <ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR97/RR97-04.ps.Z>. Accessed on: 26. Oct. 2015.

FILLIÂTRE, J.-C. *Finite Automata Theory in Coq: A constructive proof of Kleene's theorem*. 1997. Available in: <https://www.lri.fr/%7efilliatr/pub/Filliatre97.html>. Accessed on: 26. Oct. 2015.

FIRSOV, D.; UUSTALU, T. Certified parsing of regular languages. In: GONTHIER, G.; NORRISH, M. (Ed.). *Certified Programs and Proofs: Third International Conference*. Melbourne, VIC, Australia: Springer International Publishing, 2013. (CPP '13), p. 98–113. ISBN 978-3-319-03545-1.

FIRSOV, D.; UUSTALU, T. Certified {CYK} parsing of context-free languages. *Journal of Logical and Algebraic Methods in Programming*, v. 83, n. 5–6, p. 459 – 468, 2014. ISSN 2352-2208. The 24th Nordic Workshop on Programming Theory (NWPT 2012).

FIRSOV, D.; UUSTALU, T. Certified normalization of context-free grammars. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*. New York, NY, USA: ACM, 2015. (CPP '15), p. 167–174. ISBN 978-1-4503-3296-5.

GEMALTO. *Security Technology with Java Card*. 2007. Available in: <http://www.gemalto.com/press/Pages/news%5f239.aspx>. Accessed on: 21. Sep. 2015.

GENTZEN, G. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, Springer-Verlag, v. 39, n. 1, 1935. ISSN 0025-5874.

GEUVERS, H. Proof assistants: History, ideas and future. *Sadhana*, Springer-Verlag, v. 34, n. 1, p. 3–25, 2009. ISSN 0256-2499.

GEUVERS, H.; NEDERPELT, R. N. G. de Bruijn's contribution to the formalization of mathematics. *Indagationes Mathematicae*, v. 24, n. 4, p. 1034 – 1049, 2013. ISSN 0019-3577. In memory of N.G. (Dick) de Bruijn (1918–2012).

GINSBURG, S. *The Mathematical Theory of Context-Free Languages*. New York, NY, USA: McGraw-Hill, Inc., 1966.

GONTHIER, G. Formal Proof – The Four-Color Theorem. *Notices of the American Mathematical Society*, v. 55, n. 11, p. 1382–1393, dez. 2008. Available in: <http://www.ams.org/notices/200811/tx081101382p.pdf>. Accessed on: 26. Oct. 2015.

GONTHIER, G. The four colour theorem: Engineering of a formal proof. In: KAPUR, D. (Ed.). *Computer Mathematics*. [S.l.]: Springer Berlin Heidelberg, 2008, (Lecture Notes in Computer Science, v. 5081). p. 333–333. ISBN 978-3-540-87826-1.

GONTHIER, G. *A computer-checked proof of the Four Colour Theorem*. 2015. Available in: <http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf>. Accessed on: 21. Sep. 2015.

GONTHIER, G. et al. A machine-checked proof of the Odd Order Theorem. In: BLAZY, S.; PAULIN-MOHRING, C.; PICHARDIE, D. (Ed.). *Interactive Theorem Proving*. [S.l.]: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7998). p. 163–179. ISBN 978-3-642-39633-5.

GORDON, M. Proof, language, and interaction. In: PLOTKIN, G.; STIRLING, C.; TOFTE, M. (Ed.). Cambridge, MA, USA: MIT Press, 2000. cap. From LCF to HOL: A Short History, p. 169–185. ISBN 0-262-16188-5.

HALES, T. Formal proof. *Notices of the American Mathematical Society*, v. 55, n. 11, p. 1370–1380, 2008.

HALES, T. *Dense Sphere Packings: A Blueprint for Formal Proofs*. New York, NY, USA: Cambridge University Press, 2012. ISBN 0521617707, 9780521617703.

HALES, T. *The Flyspeck Project*. 2015. Available in: <https://code.google.com/p/flyspeck/>. Accessed on: 26. Oct. 2015.

HALES, T. et al. *A formal proof of the Kepler Conjecture*. 2015. Available in: <http://arxiv.org/abs/1501.02155>. Accessed on: 26. Oct. 2015.

HARRISON, J. *Formalized Mathematics*. Turku, Finland, 1996.

HARRISON, J. Formal proof — theory and pratice. *Notices of the American Mathematical Society*, v. 55, n. 11, p. 1395–1406, 2008.

HARRISON, M. A. *Introduction to Formal Language Theory*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978. ISBN 0201029553.

HEYTING, A. Die formalen Regeln der intuitionistischen Logik. In: *Sitzungsber. Preuss. Akad. Wiss., Phys. - Math Kl.* [S.l.: s.n.], 1930. p. 42–56.

HINDLEY, J. R.; SELDIN, J. P. *Lambda-Calculus and Combinators, an Introduction*. [S.l.]: Cambridge University Press, 2008. ISBN 978-0-521-89885-0.

HOPCROFT, J. E. et al. *Introduction to Automata Theory, Languages and Computability*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201441241.

HOPCROFT, J. E.; ULLMAN, J. D. *Formal Languages and Their Relation to Automata*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1969.

HOPCROFT, J. E.; ULLMAN, J. D. *Introduction To Automata Theory, Languages and Computation*. [S.l.]: Addison-Wesley Publishing Co., Inc., 1979. ISBN 0-201-02988-X.

HOWARD, W. A. The formulas-as-types notion of construction. In: SELDIN, J. P.; HINDLEY, J. R. (Ed.). *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. [S.l.]: Academic Press, 1980. p. 479–490.

INRIA. *Coq received ACM Software System 2013 award*. 2013. Available in: <https://coq.inria.fr/news/119.html>. Accessed on: 26. Oct. 2015.

INRIA. *The Coq Proof Assistant*. 2014. Available in: <http://coq.inria.fr/>. Accessed on: 26. Oct. 2015.

INRIA. *Coq Users' Contributions*. 2015. Available in: <http://www.lix.polytechnique.fr/coq/pylons/contribs/index>. Accessed on: 26. Oct. 2015.

INRIA. *List of Coq Projects in Formal Mathematics*. 2015. Available in: <http://coq.inria.fr/cocorico/List%20of%20Coq%20Math%20Projects>. Accessed on: 26. Oct. 2015.

INRIA. *List of Coq Projects in Programming Language Research*. 2015. Available in: <https://coq.inria.fr/cocorico/List%20of%20Coq%20PL%20Projects>. Accessed on: 26. Oct. 2015.

INRIA. *List of Coq Projects in Teaching*. 2015. Available in: <http://coq.inria.fr/cocorico/CoqInTheClassroom>. Accessed on: 26. Oct. 2015.

INRIA. *The Coq Standard Library*. 2016. Available in: <http://coq.inria.fr/distrib/current/stdlib/>. Accessed on: 15. Jan. 2016.

JOURDAN, J.-H.; POTTIER, F.; LEROY, X. Validating LR(1) parsers. In: *Proceedings of the 21st European Conference on Programming Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2012. (ESOP'12), p. 397–416. ISBN 978-3-642-28868-5.

KALOPER, M.; RUDNICKI, P. *Minimization of Finite State Machines*. 1994. Available in: <http://mws.cs.ru.nl/mwiki/fsm%5f1.html>. Accessed on: 28. Jul. 2014.

KALOPER, M.; RUDNICKI, P. Minimization of finite state machines. *Formalized Mathematics*, v. 5, n. 2, p. 173–184, 1996.

KAUFMANN, M.; MOORE, J. S.; MANOLIOS, P. *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0792377443.

KLEIN, G. et al. sel4: Formal verification of an operating-system kernel. *Commun. ACM*, ACM, New York, NY, USA, v. 53, n. 6, p. 107–115, jun. 2010. ISSN 0001-0782. Available in: <http://doi.acm.org/10.1145/1743546.1743574>. Accessed on: 26. Oct. 2015.

KOPROWSKI, A.; BINSZTOK, H. TRX: A formally verified parser interpreter. In: *Proceedings of the 19th European Conference on Programming Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2010. (ESOP'10), p. 345–365. ISBN 3-642-11956-5, 978-3-642-11956-9.

KOZEN, D. C. *Automata and Computability*. [S.l.]: Springer, 1997. ISBN 978-1-4612-7309-7.

KRAUSS, A.; NIPKOW, T. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automated Reasoning*, Springer Netherlands, v. 49, n. 1, p. 95–106, 2012. ISSN 0168-7433.

KREITZ, C. *Constructive Automata Theory Implemented with the Nuprl Proof Development System*. Ithaca, NY, 1986.

LEROY, X. Formal verification of a realistic compiler. *Commun. ACM*, ACM, New York, NY, USA, v. 52, n. 7, p. 107–115, jul. 2009. ISSN 0001-0782.

LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elements of the Theory of Computation*. Second. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998. ISBN 0132624788.

MARTIN-LÖF, P. *Intuitionistic Type Theory*. 1980. [Notes by Giovanni Sambin of a series of lectures given in Padua].

MICROSOFT RESEARCH. *Coq Proof of the Four Color Theorem*. 2006. Available in: <http://research.microsoft.com/en-us/downloads/5464E7B1-BD58-4F7C-BFE1-5D3B32D42E6D/default.aspx>. Accessed on: 26. Oct. 2015.

MICROSOFT RESEARCH. *Theorem Proof Gains Acclaim*. 2012. Available in: <http://research.microsoft.com/en-us/news/features/gonthierproof-101112.aspx>. Accessed on: 21. Sep. 2015.

MICROSOFT RESEARCH-INRIA JOINT CENTRE. *Feit-Thomson proved in Coq*. 2012. Available in: <http://www.msr-inria.fr/news/feit-thomson-proved-in-coq/>. Accessed on: 26. Oct. 2015.

MICROSOFT RESEARCH-INRIA JOINT CENTRE. *The formalization of the Odd Order theorem has been completed*. 2012. Available in: <http://www.msr-inria.fr/news/the-formalization-of-the-odd-order-theorem-has-been-completed-the-20-septembre-2012/>. Accessed on: 26. Oct. 2015.

MICROSOFT RESEARCH-INRIA JOINT CENTRE. *Mathematical Components*. 2015. Available in: <http://www.msr-inria.inria.fr/projects/mathematical-components-2/>. Accessed on: 26. Oct. 2015.

MICROSOFT RESEARCH-INRIA JOINT CENTRE. *Microsoft Research-INRIA Joint Centre*. 2015. Available in: <http://www.msr-inria.fr/>. Accessed on: 21. Sep. 2015.

MIYAMOTO, T. *RegExp*. 2014. Available in: <http://www.lix.polytechnique.fr/coq/pylons/contribs/view/RegExp/trunk>. Accessed on: 26. Oct. 2015.

MOREIRA, N.; PEREIRA, D.; SOUSA, S. M. de. *On the Mechanization of Kleene Algebra in Coq*. [S.l.], 2009. Available in: <http://www.dcc.fc.up.pt/Pubs/>. Accessed on: 26. Oct. 2015.

MOREIRA, N.; PEREIRA, D.; SOUSA, S. M. de. Deciding regular expressions (in-)equivalence in Coq. In: KAHL, W.; GRIFFIN, T. (Ed.). *Relational and Algebraic Methods in Computer Science*. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7560). p. 98–113. ISBN 978-3-642-33313-2.

NIPKOW, T.; WENZEL, M.; PAULSON, L. C. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN 3-540-43376-7.

NORDSTRÖM, B.; PETERSSON, K.; SMITH, J. M. *Programming in Martin-Löf's Type Theory*. [S.l.]: Oxford University Press, 1990.

NORRISH, M. Mechanised computability theory. In: EEKELEN, M. et al. (Ed.). *Interactive Theorem Proving*. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6898). p. 297–311. ISBN 978-3-642-22862-9.

PELAYO, A.; WARREN, M. A. *Homotopy Type Theory and Voevodsky's Univalent Foundations*. 2012. Available in: <http://arxiv.org/abs/1210.5658>. Accessed on: 26. Oct. 2015.

PIERCE, B. C. *Software Foundations*. 2015. Available in: <http://www.cis.upenn.edu/%7ebcpierce/sf/current/>. Accessed on: 30. Sep. 2015.

PRAWITZ, D. *Natural deduction: a proof-theoretical study*. Thesis (PhD) — Almqvist & Wiksell, 1965.

PVS. *PVS Home Page*. 2015. Available in: <http://pvs.csl.sri.com/>. Accessed on: 8. Dec. 2015.

RAMOS, M. V. M.; NETO, J. J.; VEGA, I. S. *Linguagens Formais: Teoria Modelagem e Implementação*. [S.l.]: Bookman, 2009. ISBN 9788577804535.

RAMOS, M. V. M.; QUEIROZ, R. J. G. B. de. Formalization of closure properties for context-free grammars. *CoRR*, abs/1506.03428, 2014. Available in: <http://arxiv.org/abs/1506.03428>. Accessed on: 26. Oct. 2015.

RAMOS, M. V. M.; QUEIROZ, R. J. G. B. de. Formalization of simplification for context-free grammars. *CoRR*, abs/1509.02032, 2015. Available in: <http://arxiv.org/abs/1509.02032>. Accessed on: 26. Oct. 2015.

RIDGE, T. Simple, functional, sound and complete parsing for all context-free grammars. In: *CPP*. [S.l.: s.n.], 2011. p. 103–118.

SIPSER, M. *Introduction to the Theory of Computation*. Second. [S.l.]: International Thomson Publishing, 2005. ISBN 978-0534950972.

SORENSEN, M. H. B.; URZYCZYN, P. *Lectures on the Curry-Howard Isomorphism*. [S.l.]: Elsevier Science, 2006. ISBN 978-0444520777.

SUDKAMP, T. A. *Languages and Machines*. 3rd. ed. [S.l.]: Addison-Wesley, 2006. ISBN 978-0321322210.

THE COQ DEVELOPMENT TEAM. *The Coq Reference Manual, version 8.4pl6*. 2015. Available in: <https://coq.inria.fr/distrib/current/refman/>. Accessed on: 26. Oct. 2015.

THOMPSON, S. *Type Theory and Functional Programming*. [S.l.]: Addison-Wesley, 1991. ISBN 0-201-41667-0.

TROELSTRA, A. History of constructivism in the 20th century. *Lecture Notes in Logic*, Association for Symbolic logic, Ithaca NY and Cambridge University Press, Cambridge UK, v. 36, p. 150–179, 2011.

TURING, A. M. Computability and λ-definability. *Journal of Symbolic Logic*, v. 2, p. 153–163, 12 1937. ISSN 1943-5886.

VARIOUS. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. New York, NY, USA: Cambridge University Press, 1993. ISBN 0-521-44189-7.

VARIOUS. *Selected Papers on Automath*. [S.l.]: North-Holland, 1994. v. 133. (Studies in Logic and the Foundations of Mathematics, v. 133).

VOEVODSKY, V. *Homotopy Type Theory: Univalent Foundations of Mathematics*. [S.l.]: Univalent Foundations Program, 2013. Available in: <http://homotopytypetheory.org/book/>. Accessed on: 26. Oct. 2015.

VOEVODSKY, V. *Experimental library of univalent formalization of mathematics*. 2014. Available in: <http://arxiv.org/abs/1401.0053>. Accessed on: 26. Oct. 2015.

VOEVODSKY, V. *Homotopy Type Theory and Univalent Foundations*. 2015. Available in: <http://homotopytypetheory.org/>. Accessed on: 26. Oct. 2015.

VOEVODSKY, V. *Homotopy Type Theory Repository*. 2015. Available in: <https://github.com/HoTT/HoTT>. Accessed on: 26. Oct. 2015.

WHITESIDE, I.; ASPINALL, D.; GROV, G. An essence of SSReflect. In: JEURING, J. et al. (Ed.). *Intelligent Computer Mathematics*. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7362). p. 186–201. ISBN 978-3-642-31373-8.

WIEDIJK, F. *The Seventeen Provers of the World*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 3540307044.

WIEDIJK, F. The QED Manifesto Revisited. *Studies in Logic, Grammar and Rhetoric*, v. 10, n. 23, p. 121–133, 2007.

WIEDIJK, F. Formal proof — getting started. *Notices of the American Mathematical Society*, v. 55, n. 11, p. 1408–1414, 2008.

WIEDIJK, F. *The de Bruijn Factor*. 2012. Available in: <http://www.cs.ru.nl/%7efreek/factor/>. Accessed on: 26. Oct. 2015.

WIEDIJK, F. *Formalizing 100 Theorems*. 2015. Available in: <http://www.cs.ru.nl/%7efreek/100/>. Accessed on: 11. Dec. 2015.

WU, C.; ZHANG, X.; URBAN, C. A formalisation of the Myhill-Nerode theorem based on regular expressions (proof pearl). In: EEKELEN, M. et al. (Ed.). *Interactive Theorem Proving*. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6898). p. 341–356. ISBN 978-3-642-22862-9.

XU, J.; ZHANG, X.; URBAN, C. Mechanising Turing Machines and computability theory in Isabelle/HOL. In: BLAZY, S.; PAULIN-MOHRING, C.; PICHARDIE, D. (Ed.). *Interactive Theorem Proving*. [S.l.]: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7998). p. 147–162. ISBN 978-3-642-39633-5.

ZAMMIT, V. A comparative study of coq and hol. In: *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*. London, UK, UK: Springer-Verlag, 1997. (TPHOLs '97), p. 323–337. ISBN 3-540-63379-0.

Appendix

A
Formal Mathematics

The objective of the appendix is to discuss in some detail what formal mathematics is and how it can be used. This way we will gather the necessary conditions to understand the mathematical framework upon which the Coq proof assistant is built, as well as its full potential (in Appendix B), and finally the formalization that is the object of this work.

Common sense considers that mathematics is *per se* a formal system, with formal (i.e. complete and non-ambiguous) languages to describe the entities and properties of our theories, as well as formal reasoning and computation rules that leave no room for misinterpretation nor allow for inconsistencies.

This is not always true, however. Indeed, most of mathematics that has been developed up to the present days still relies on informal notations and reasoning/computation rules that have a negative impact in both human-human communication and, recently, also restrict human-machine communication. Besides this, as more and more is learned about mathematics and more widespread is its use in virtually all areas of human knowledge, the length and complexity of demonstrations is increasing, making it a very difficult task for a human being to develop and check a proof with its own resources and capabilities alone.

In this cenario, mathematical formalization is an important approach that promises a definite solution for these problems, while still opening new possibilities for mathematicians and computer scientists.

The essence and importance of mathematical formalization is summarized by John Harrison in Harrison (1996),

> "Mathematics is generally regarded as the exact subject par excellence. But the language commonly used by mathematicians (in papers, monographs, and even textbooks) can be remarkably vague; perhaps not when compared with everyday speech but certainly when compared with the language used by practitioners of other intellectual disciplines such as the natural sciences or philosophy...
> The formalization of mathematics addresses both these questions, precision and correctness. By formalization we mean expressing mathematics, both statements and proofs, in a (usually small and simple) formal language with strict rules of

> grammar and unambiguous semantics. The latter point bears repeating: just writing something in a symbolic language is not sufficient for formality if the semantics of that language is ill-defined...
> In formalizing mathematics, we must rephrase informal constructs in terms of formal ones, which is merely an extreme case of defining non-rigorous concepts rigorously."

In other words, he praises the simplicity and the rigor of the language in which mathematical reasoning happens, and remembers us that this is not the case in common mathematical practice. For this reason, formal mathematics has emerged as an area of knowledge whose objective is to establish simple, precise and non-ambiguous reasoning and computation rules that can be used by humans, with the aid of computers, in order to develop and check proofs with higher confidence, in shorter periods of time, thus increasing productivity and the capability of dealing with higher complexity matters.

Formalization is, according to Freek Wiedijk (WIEDIJK, 2008), "computer encoded mathematics". A formal proof is, in John Harrison's words (HARRISON, 2008):

> "a proof written in a precise artificial language that admits only a fixed repertoire of stylized steps. This formal language is usually designed so that there is a purely mechanical process by which the correctness of a proof in the language can be verified"

Formal mathematics is, thus, mathematics that can be mechanically checked. Among its main characteristics, when compared to traditional (or "informal") mathematics we find:

- The use of a uniform, simple and rigorous notation, which allows:
    - the construction of complete, detailed and non-ambiguous proofs,
    - the mechanical verification of proofs and
    - some degree of proof automation.

These characteristics, in turn, lead to the following advantages:

- Greater confidence in proofs;
- Faster verification of proofs;
- Easier reuse of previous results;
- Enhanced communication between users and users and machines.

Formal mathematics is done on top of some formal theory, which in turn is composed by a finite set of axioms and simple and intuitive inference rules that can be combined in order to express proofs and propositions. A formal proof is a sequence of axioms or application of inference rules that assert the validity of a proposition.

It is thus necessary to establish the formal theory (or calculus) that will be used as the basis for a formal mathematics activity. The formal theory used by the Coq proof assistant is called “Calculus of Inductive Constructions”. It is a Type Theory that extends the Curry-Howard Correspondence between terms of the Typed Lambda Calculus and proofs of natural deduction, as well as between specifications (type expressions) and propositions.

The more fundamental theories on which this calculus is based are briefly reviewed in the following sections, and have the final objective of making clear the semantics of the languages and rules used by the Coq proof assistant. It is this calculus that allows that both reasoning and computing be represented and can interact with each other in a mathematical formalization, enabling the definition and manipulation of complex objects in the statement and proof of complex propositions.

We start in Section A.1 with a presentation of the languages of propositional and predicate logic, which are fundamental in the statement of the lemmas and theorems of our theory. We then proceed to Section A.2 and show how a proposition can be proved using the simple technique of Natural Deduction, under the Tarski interpretation.

Although seemingly unrelated to the previous topics, we make an introduction to the Untyped Lambda Calculus in Section A.3. This is a very simple, elegant and powerful calculus that is used to represent computable functions and to study their properties. Its expressive power, however, leads to some inconsistencies that were fixed later in different ways, in particular by the Type Theories.

By introducing the notion of *type*, and by assigning types to terms of the calculus, which in turn lead to the idea of *type checking*, a Type Theory limits its expressiveness in terms of the representation of general computation, but still retains enough power to be considered interesting from the theoretical point of view. Due to its characteristics, type theories have even inspired the design of modern programming languages. The results of incorporating types in the Untyped Lambda Calculus resulted in a version known as the Typed Lambda Calculus, which is briefly presented in Section A.4.

Returning to the logic domain, we introduce in Section A.5 the important notions of *constructivism* and the BHK interpretation. The latter replaces the original idea of relating propositions to truth values (known as Tarski interpretation) by the more interesting and useful idea of relating propositions to proofs of their validity. The former enables the construction of proofs that have computational content, an idea that is explored with great advantages by many proof assistants, specially in computer science applications. Together, these notions represent the modern status of logical reasoning.

We now have, on one side, proofs and propositions (Section A.1, Section A.2 and Section A.5) and, in the other, terms and types (Section A.3 and Section A.4). It didn´t take long until it was observed that there were similarities in the syntactical structure of proofs (of Natural

Deduction) and terms (of the Typed Lambda Calculus). More than that, between the corresponding propositions and types. Because of this, this relationsip became the subject of deep study by different researchers, leading finally to the so-called Curry-Howard Isomorphism introduced in Section A.6. This opened up the possibility of relating and exploring the mechanisms of reasoning and computing in a way that brought many benefits to the development of formal proofs.

All these ideas were then collected and merged into the Calculus of Constructions (CC), which because of this became a powerful reasoning system with computing capabilities. Later extended with inductive definitions, it was renamed Calculus of Inductive Constructions (CIC) and constitutes nowadays the formal theory on which the Coq proof assistant is built upon. Some charateristics of this calculus are discussed in Section A.7.

Wiedijk explains why he considers formalization as one of the three most important revolutions that mathematics has undergone in its whole history, along with the introduction of the concepts of *proof* by the greeks in the 4th century and of *rigor* in the 19th century, and provides a synthetic yet strong expression of its importance in our times. In his own words (WIEDIJK, 2008):

> "Most mathematicians are not aware that this third revolution already has happened, and many probably will disagree that this revolution even is needed. However, in a few centuries mathematicians will look back at our time as the time of this revolution. In that future most mathematicians will not consider mathematics to be definitive unless it has been fully formalized."

John Harrison also notes, in the conclusion of Harrison (2008), that:

> "The use of formal proofs in mathematics is a natural continuation of existing trends towards greater rigor."

There is no doubt that the activity of mathematical formalization is mature enough and has reached a stage where robust proof assistants based on solid theories offer us a serious and radically new way of doing mathematics, with many advantages. As pointed out by Wiedijk and Harrison, this new era of mathematics has already started and will open up new horizons to mathematicians and computer scientists.

A.1 Propositional and Predicate Logic

To start with, we need a logic to write down our lemmas and theorems in a precise and unambiguous way. This same logic will then be used to reason about our statements in order to try to construct proofs for them. Predicate logic is a logic that has many important characteristics, including simplicity, expressiveness and widespread use, besides being a natural choice for expressing general propositions. For this reason, it lies as the fundamental theory on which most interactive proof assistants are built upon.

As an example, the following lemma (included in the present formalization) is a formula of the predicate logic that uses a predicate (`derives`), the universal quantifier (∀) and the connectives for conjunction (∧) and implication (→) to express the fact that two independent derivations in the same grammar (see Section 5.2) can be combined into a single derivation (`g` is a context-free grammar, `s1`, `s2`, `s3` and `s4` are sentential forms of this grammar and ++ is an operator for string concatenation):

```
∀ (g : cfg non_terminal terminal)
∀ (s1 s2 s3 s4 : sf),
derives g s1 s2 ∧ derives g s3 s4 → derives g (s1 ++s3) (s2 ++s4)
```

The objective of this section is to introduce the syntax and semantics of predicate logic, in order to enable the construction and interpretation of formulas like this, and thus allow the statement of our theory as a collection of lemmas and theorems that need to be proved. Eventually, this same logic will be used to represent program specifications, although this is not the main focus of our work. Propositional logic is a simpler logic, and will be introduced first, as a preparation for the presentation of predicate logic.

Considered by some the foundations of mathematics, and by others as a branch of mathematics itself, logic is concerned, in general, with the representation and validation of arguments. For this purpose, formal logic comes in different flavors, such as propositional logic (whose sentences consist of variables grouped around a small set of connectives) and first-order logic (whose sentences allow the use of the universal and the existential quantifiers in addition to the connectives of propositional logic) (DALEN, 2008). Since mathematical theorems are generally formulated as propositions, it is a natural choice to use the language of first-order logic to express them formally.

"Classical logic" refers to a logic where the law of the excluded middle is accepted. In this kind of logic, every proposition is either true or false, even if there is no evidence of the proposition being true nor evidence of the proposition being false.

"Intuitionistic logic", on the other hand, originally conceived by Brouwer (BROUWER, 1975) and further developed by Heyting (HEYTING, 1930) and Gentzen (GENTZEN, 1935), restricts classical logic by not allowing the use of the law of the excluded middle. In this logic, for a proposition to be considered true or false there must exist a proof of one or the other. As an example, in classical logic the proposition from complexity theory $P = NP$ or $P \neq NP$ is valid, although it is not known, for the time being, whether either of the disjuncts is true or false. Intuitionistic logic, on the contrary, does not consider this as a valid proposition, as no one yet knows of a proof of either one of the disjuncts.

Although not all propositions that can be proved in classical logic can also be proved in intuitionistic logic, the latter has the special property that from a proof one can effectively build the object that confirms the validity of the proposition (the proof behaves like a kind of a recipe for building such object). For this reason, intuitionistic logic is also called “constructive” logic. An important consequence of this is that computer programs can be extracted from constructive proofs, and for this reason this logic is known for having “computational content”, a property that, although being extensively investigated via continuations or negative translations, does not naturally arise out of classical logic. It is of great interest in computer science and is briefly discussed in Section A.5.

In the present context, propositions will mainly be used to state lemmas and theorems. A secondary use will be to specify computer programs (for example, the properties of the computation performed by a program, the relations between input and output data of a program etc), in which case propositions will be called *specifications* or *types*. Both activities are indeed equivalent, as prescribed by the Curry-Howard Correspondence (presented in Section A.6). Nevertheless, programs must be built and theorems must be proved, and one important tool for this comes from proof theory and is known as “Natural Deduction” (introduced in Section A.2).

A proposition is a formula that uses *variables* and *logical connectives*. The logic of propositions is known as *Propositional Logic*. The formulas of this logic are defined by the following rules:

$$\begin{array}{rcl} formula & ::= & variable \\ & | & \bot \\ & | & \top \\ & | & (formula \wedge formula) \\ & | & (formula \vee formula) \\ & | & (formula \Rightarrow formula) \\ & | & (formula \Leftrightarrow formula) \\ & | & (\neg\, formula) \\ variable & ::= & a\,|\,b\,|\,c\,|\,\ldots \end{array}$$

The logical connecives are:

- $\wedge$: Conjunction (“and”);
- $\vee$: Disjunction (“or”);
- $\Rightarrow$: Implication (“if-then”);
- $\Leftrightarrow$: Bi-implication ($(a \Leftrightarrow b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$) (“if-and-only-if”);
- $\neg$: Negation ($\neg\, a \equiv a \Rightarrow \bot$) (“not”).

The semantics associated to these connectives, and to formulas is general, is not unique. The traditional interpretation (due to Tarski) associates each variable with a true or a false value, and then proceeds by inferring the value of the whole formula considering the application of a "truth table" for each connective used in it. In this interpretation, for example, if a is true and b is true, then $a \wedge b$ is true.

Propositional logic also use the following symbols, representing, respectively, the false and the true propositions:

- $\bot$: False;
- $\top$: True ($\top \equiv \bot \Rightarrow \bot$).

In general, a formula of propositional logic is considered true when all combinations of values assigned to its arguments result in a true value after being evaluated by the truth tables of the corresponding connectives. As an example, the formula $a \wedge b \Leftrightarrow b \wedge a$ (which expresses the commutativity of the connective $\wedge$) can be proved true by considering all possibilities of values assigned to a and b, as in Table A.1:

Table A.1: Proof of $a \wedge b \Leftrightarrow b \wedge a$

a	b	$a \wedge b$	$b \wedge a$	$a \wedge b \Leftrightarrow b \wedge a$
$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$

A different interpretation considers that a formula is true only if we can present a proof of it (this corresponds to the BHK interpretation, introduced in Section A.5). For example, $a \wedge b$ would only be considered true if we could show proofs that both a and b are true.

Some examples of formulas of propositional logic are:

- $(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$
- $(a \wedge b) \Rightarrow (b \wedge a)$
- $(a \vee (a \wedge b)) \Rightarrow a$
- $(a \Rightarrow b) \Rightarrow (\neg\, b \Rightarrow \neg\, a)$

When the Propositional Logic is extended with *quantifiers* and *predicates*, it becomes known as Predicate Logic and is defined as follows:

$$
\begin{array}{rcl}
formula & ::= & variable \\
& | & \bot \\
& | & \top \\
& | & pred\_name\,(arg\_list) \\
& | & (formula \wedge formula) \\
& | & (formula \vee formula) \\
& | & (formula \Rightarrow formula) \\
& | & (formula \Leftrightarrow formula) \\
& | & (\neg formula) \\
& | & (\forall variable\,.\,formula) \\
& | & (\exists variable\,.\,formula)
\end{array}
$$

$$
\begin{array}{rcl}
pred\_name & ::= & P_0 \,|\, P_1 \,|\, P_2 \,|\, \ldots \\
arg\_list & ::= & term \,|\, arg\_list\,,\,term \\
term & ::= & fun\_name\,(arg\_list) \,|\, variable \,|\, constant \\
variable & ::= & a \,|\, b \,|\, c \,|\, \ldots \\
constant & ::= & c_0 \,|\, c_1 \,|\, c_2 \,|\, \ldots
\end{array}
$$

The logical quantifiers:

- $\forall$: Universal quantifier (“for all”);
- $\exists$: Existential quantifier (“exists”).

are key in the characterization of Predicate Logic, as they enable reasoning on collections of elements. The universal quantifier is used to make assertions over a (possibly infinite) collection of values. For example, $\forall x.x = x$. The existential quantifier, on the other hand, is useful in asserting that some property is valid for some value of a larger collection. For example, $\exists y.y^2 = 4$.

The introduction of these quantifiers creates the notions of *bound* and *free* variables. In general, a bound variable is one who is in the *scope* of a name immediately to the right of some quantifier. A free variable is a variable which is not bound. For example, x is bound in $\forall x.x = x$ and y is free is $\forall x, (x = y) \to (y = x)$. The notions of scope, bound and free variables, as well methods for determining them, can be found in Dalen (2008). They are very similar, however, to the equivalent notions of the Untyped Lambda Calculus, described in Section A.3.

A *predicate* is a function that takes a number of arguments and returns a proposition. Predicates are used to define properties that are shared by different objects. For example, *prime n* is a predicate that asserts that number *n* is prime, as in *prime 5*. A predicate (also called a *parametrized proposition*) represents a family of propositions.

In the previous grammar, *variable* represents logic variables, *pred_name* represents different predicate names and *arg_list* represents a list of arguments (also called *terms*) which are used in a function call or in a predicate instatiation. A term can contain nested function calls, as well as variable and constant names, respectively represented by *variable* and *constant*.

The following are examples of correctly constructed formulas in predicate logic:

- $\forall a\ b,\ (a \wedge b) \Rightarrow (b \wedge a)$
- $\forall x, x \neq 0 \Rightarrow \exists y,\ x * y = 1$
- $\forall x.R(x,x) \Rightarrow \forall x.\exists y.R(x,y)$
- $\forall x.R(f(x), g(f(x)) \Rightarrow \forall x.\exists y.R(f(x), y)$

Now that we have the tools to state our lemmas and theorems, it will be necessary to consider possible ways to prove that a formula is true. Since considering all possible value assignments for the variables of a formula might not be feasible (considering the Tarski interpretation), one should consider alternatives that make this task possible and easier. For this reason, we introduce Natural Deduction in the following section.

A.2 Natural Deduction

Natural Deduction is a calculus for theorem proving and is part of Proof Theory. It uses a small set of *inference rules* in order to find a logical path from some hypotheses (or none) to a conclusion that needs to be proved. The reasoning steps used in this process (directly related to the structure of the conclusion) are very simple and naturally related to common reasoning.

For each connective of the logic, the calculus presents an introduction and an elimination rule that are part of its set of inference rules. The proof of a theorem (a formula of predicate logic, generically referred to as a proposition) is a structured sequence of inference rules that validate the conclusion from the axions and/or the hypotheses.

An inference rule is a rule that allows a conclusion from a set of premises. The premises appear above an horizontal line, while the conclusion comes below the same line. The proof itself is represented as a tree that shows all the primitive reasoning steps used (one for each connective used in the formula) and how they transform the axioms and hypothesis into the conclusion.

Natural Deduction was originally developed for propositional logic (GENTZEN, 1935) and was later extended for predicate logic (PRAWITZ, 1965).

The following are the inference rules for the implication connective (I stands for "Introduction" and E stands for "Elimination"). It states that if a is a true proposition, and from it one can conclude that b is also true, then it is true that a implies b:

$$\dfrac{\begin{matrix}[a]\\ \vdots \\ b\end{matrix}}{a \Rightarrow b}\,(\Rightarrow I)$$

Notation $[a]$ is used to represent the fact that assumption a is no longer needed when the proof of $a \Rightarrow b$ is constructed, and for this reason it is *discharged.* In the following examples, the brackets will be omitted when this is clear from the context.

The elimination rule for implication states that from a true premise one may infer that the conclusion is also true:

$$\dfrac{a \Rightarrow b \qquad a}{b}\,(\Rightarrow E)$$

The tree presented next illustrates the use of these rules to construct a proof of the proposition $(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[a \Rightarrow (b \Rightarrow c)] \qquad [a]}{b \Rightarrow c}\,(\Rightarrow E) \qquad [b]}{c}\,(\Rightarrow E)}{a \Rightarrow c}\,(\Rightarrow I)}{b \Rightarrow (a \Rightarrow c)}\,(\Rightarrow I)}{(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}\,(\Rightarrow I)$$

The introduction rule for conjunction ($\wedge$), presented next, states that two true propositions make a new true proposition. For elimination, there are two rules: from a true conjunction it is possible to conclude that the first proposition is true, and also that the second proposition is true.

$$\dfrac{a \qquad b}{a \wedge b}\,(\wedge I)$$

$$\dfrac{a \wedge b}{a}\,(\wedge E_1)$$

$$\dfrac{a \wedge b}{b}\,(\wedge E_2)$$

The following tree illustrates the use of these rules in the proof of the proposition $(a \wedge b) \Rightarrow (b \wedge a)$:

$$\dfrac{\dfrac{\dfrac{a \wedge b}{b}\,(\wedge E) \qquad \dfrac{a \wedge b}{a}\,(\wedge E)}{b \wedge a}\,(\wedge I)}{(a \wedge b) \Rightarrow (b \wedge a)}\,(\Rightarrow I)$$

Introduction rules for the disjunction ($\vee$) come also in pair: they state that a true proposition can be combined with any other proposition (not necessarily true) and the resulting proposition will also be true.

$$\dfrac{a}{a \vee b}\,(\vee I_1)$$

$$\dfrac{b}{a \vee b}\,(\vee I_2)$$

The elimination rule for disjunction demands that some new proposition c be true, in the hypotheses that both a or b are true. Thus, the fact that $a \vee b$ is true can be substitued by the fact that c is true, and the connective $\vee$ becomes eliminated:

$$\dfrac{a \vee b \qquad \begin{array}{c}[a]\\ \vdots\\ c\end{array} \qquad \begin{array}{c}[b]\\ \vdots\\ c\end{array}}{c}\,(\vee E)$$

The following tree illustrates the use of these rules in the proof of the proposition $(a \vee (a \wedge b)) \Rightarrow a$:

$$\dfrac{\dfrac{a \vee (a \wedge b) \qquad \begin{array}{c}[a]\\ a\end{array} \qquad \dfrac{[a \wedge b]}{a}\,(\wedge E)}{a}\,(\vee E)}{(a \vee (a \wedge b)) \Rightarrow a}\,(\Rightarrow I)$$

Inference rule for false ($\perp$) exists only for elimination (*ex-falso quodlibet*). The interpretation, in this case, is that from a false statement one may consider any proposition true:

$$\dfrac{\perp}{a}\,(\perp E)$$

As seen is Section A.1, negation ($\neg$) is just an abbreviation for implication to false. Thus, the introduction and elimination rules for this connective are special cases of the rules introduced before for the implication connective. The introduction rule, in particular, states that if considering a proposition true leads to a false conclusion, then the proposition should be considered false (and thus its negation true):

$$\dfrac{\begin{array}{c}[a]\\ \vdots\\ \perp\end{array}}{\neg\, a}\,(\neg\, I, \text{same as } \Rightarrow I)$$

Negation elimination takes two contradictory propositions and uses implication elimination to prove absurd:

$$\frac{a \qquad \neg\, a}{\bot}\ (\neg\, E\text{, same as } \Rightarrow E)$$

The following tree illustrates the use of these rules in the proof of the proposition $(a \Rightarrow b) \Rightarrow (\neg\, b \Rightarrow \neg\, a)$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{a \Rightarrow b \qquad a}{b}\ (\Rightarrow E) \qquad \neg\, b}{\bot}\ (\neg\, E)}{\neg\, a}\ (\Rightarrow I)}{\neg\, b \Rightarrow \neg\, a}\ (\Rightarrow I)}{(a \Rightarrow b) \Rightarrow (\neg\, b \Rightarrow \neg\, a)}\ (\Rightarrow I)$$

Inference rules for the universal quantifier ($\forall$) are presented next. The introduction rule states that a proposition that contains a free variable can be generalized over all the values of this variable. However, no undischarged assumption on which $A(x)$ depends may contain x occurring free:

$$\frac{A(x)}{\forall x.A(x)}\ (\forall I)$$

The elimination rule simply instantiates the generalisation by substituting the bound variable by some value:

$$\frac{\forall x.A(x)}{A[t/x]}\ (\forall E)$$

Finally, inference rules for the existential quantifier ($\exists$) are presented next. The introduction rule states that it is enough to prove the formula true for one value of an argument in order to prove the existence of some value for this argument for which the formula is true:

$$\frac{A[t/x]}{\exists x.A(x)}\ (\exists I)$$

The elimination rule for the existential quantifier behaves in a similar way to the elimination of the disjunction: the proposition that can be inferred from the true proposition instantiated for some value of the argument of the existential quantifier can be proved true directly.

$$\frac{\exists x.A(x) \qquad \begin{array}{c}[A[t/x]] \\ \vdots \\ B\end{array}}{B}\ (\exists E)$$

In this case, B cannot have free variables introduced by A. The following tree illustrates their use in the proof of the proposition $\forall x.R(x,x) \Rightarrow \forall x.\exists y.R(x,y)$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\forall x.R(x,x)}{R(x,x)}\,(\forall E)}{\exists y.R(x,y)}\,(\exists I)}{\forall x.\exists y.R(x,y)}\,(\forall I)}{(\forall x.R(x,x) \Rightarrow (\forall x.\exists y.R(x,y)))}\,(\Rightarrow I)$$

It should be noted, in examples presented, that it was not necessary to assign values to all variables of the formulas that were proved, and then to consider all possibile combinations of these values in order to determine whether the proposition was true or not. This is the main advantage of the Natural Deduction technique.

Natural Deduction is a simple and powerful tool for proving formulas of predicate logic. However, it is just a more concise way for proving a formula, rather than having to make extensive and complex valuations for the variables in it. It is, therefore, based in the Tarski interpretation mentioned before, while our interest is in the BHK interpretation, since we follow the constructive approach.

The BHK interpretation demands the ability to construct proofs for a formula, based on the proofs (hopefully simpler) of its constituents. We are thus interested in relating a proof to a formula (or proposition). This will be done in Section A.5, but will demand some important concepts developed over the next two sections (Section A.3 and Section A.4). In these sections, we will relate terms (or programs) to types (or specifications), and present results and notations that will be extremely useful when relating proofs to propositions. The direct relationship between all these concepts, and its impact for our theory, will be further discussed in Section A.6. Natural Deduction will have a fundamental role in the establishment of this relationship.

The next section is about the Untyped Lambda Calculus, a system for representing computable functions, widely used as a model of computation. Although not initially directly related to our discussion of logic, it will later evolve into a model of our logic as well (in Section A.4).

A.3 Untyped Lambda Calculus

The Untyped Lambda Calculus is a formal system used in the representation of computations (HINDLEY; SELDIN, 2008). It is, thus, a model of computation, and is based on the abstraction and application of functions, which are considered as higher order objects. This means that functions can both be passed as arguments to other functions, and can also be returned as values from function calls.

The so-called lambda language (CHURCH, 1936) is a notation for the representation of lambda terms, which are then manipulated by the rules of the calculus. The power of the calculus is related to the simplicity of the lambda language, which uses only two operators to represent all computations. These are the function abstraction and the function application (call) operations.

The Lambda Calculus was invented by the American mathematician Alonzo Church in the 1930s (CHURCH, 1932; CHURCH, 1933; CHURCH, 1936) as a result of his research on the foundations of mathematics. His objective, by then, was to formalize mathematics through the use of “functions as rules” instead of the classical use of sets as in set theory. Although he did not fulfill his objective, his work was of high importance to computer science, specially in the theory, specification and implementation of computer programming languages (the functional ones have a direct influence of it), representation of computable functions, computer program verification and proof theory. The first undecidable mathematical problems ever proved to be so in the scientific literature were formulated via the Untyped Lambda Calculus, even before the invention of Turing machines, which became since then a preferred model for computability research. The equivalence of the Untyped Lambda Calculus and the Turing machine was later established by Turing (TURING, 1937).

A λ-term (also known as lambda-expression) is inductively defined over a set of identifiers $\{x, y, z, u, v...\}$ that represent variables:

- A variable or a constant (also called “atom”) is a λ-term;
- *Application*: if M and N are λ-terms, then (MN) is a λ-term; it represents the application of M to N (a function call of M with argument N);
- *Abstraction*: if M is a λ-term and x is a variable, then $(\lambda x.M)$ is a λ-term; it represents the function that evaluates to M with parameter x;

The lambda language is composed of all λ-terms that can be constructed over a certain set of identifiers; it is a language with only two operators: application (function call) and abstraction (function definition). It can be represented by the following grammar:

$$\begin{array}{lcl} V & \rightarrow & u\,|\,v\,|\,x\,|\,y\,|\,z\,|\,w\,|\,\ldots \\ T & \rightarrow & V \\ T & \rightarrow & (TT) \\ T & \rightarrow & (\lambda V.T) \end{array}$$

These are examples of λ-terms:

- x
- (xy)
- $(\lambda x.(xy))$
- $((\lambda y.y)(\lambda x.(xy)))$
- $(x(\lambda x.(\lambda x.x)))$

- $(\lambda x.(yz))$

Let P and Q be two λ-terms. The relation P "occurs" in Q (or P is contained in Q, Q contains P or P is a subterm of Q) is inductively defined as:

- P occurs in P;
- If P occurs in M or in N, then P occurs in (MN);
- If P occurs in M or $P \equiv x$ then P occurs in $(\lambda x.M)$.

As an example, the only occurrence of xy in $\lambda xy.xy$ is $(\lambda x.(\lambda y.(\underbrace{xy})))$. The occurrences of uv in $x(uv)(\lambda u.v(uv))uv$ are $((((x(\underbrace{uv}))(\lambda u.(v(\underbrace{uv}))))u)v)$. Finally, the term $\lambda u.u$ does not occur in $\lambda u.uv$ because $\lambda u.uv \equiv (\lambda u.(uv))$.

For a particular occurrence of $\lambda x.M$ in P, M is called the "scope" of the λx to its left. For example, be $P \equiv (\lambda y.yx(\lambda x.y(\lambda y.z)x))vw$. Then,

- The scope of the leftmost λy is $yx(\lambda x.y(\lambda y.z)x)$;
- The scope of λx is $y(\lambda y.z)x$;
- The scope of the rightmost λy is z.

The occurrence of a variable x in a term P is:

- "Bound" if it is in the scope of a λx in P;
- "Bound and binding" if and only if it is the x in λx;
- "Free" otherwise.

The set of all free variables of a term P is called $FV(P)$.

For all M, N, x, $[N/x]M$ is defined as the result of the substitution of all free occurrences of x in M by N, together with the change of bound variables in case this is necessary in order to avoid name collisions:

a. $[N/x]x \equiv N$;

b. $[N/x]a \equiv a$, for all atom $a \not\equiv x$;

c. $[N/x](PQ) \equiv ([N/x]P[N/x]Q)$;

d. $[N/x](\lambda x.P) \equiv \lambda x.P$;

e. $[N/x](\lambda y.P) \equiv \lambda y.P$, if $x \notin FV(P)$;

f. $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$, if $x \in FV(P)$ e $y \notin FV(N)$;

g. $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$, if $x \in FV(P)$ e $y \in FV(N)$.

In cases (e)-(g), $y \not\equiv x$; in case (g), z is the first variable $\notin FV(NP)$.

Let P be a term that contains an occurrence of $\lambda x.M$ and suppose that $y \notin FV(M)$. The substitution of $\lambda x.M$ by $\lambda y.[y/x]M$ is called *α-conversion* in P. If P can be converted in Q by means of a finite sequence of α-conversions, then P is *α-convertible* to Q, denoted $P \equiv_\alpha Q$.

A term of the form $(\lambda x.M)N$ is called *β-redex*, and the corresponding term $[N/x]M$ is called its *contractum*. If a term P contains an occurence of $(\lambda x.M)N$ and this is substituted by $[N/x]M$, generating P', we say that the redex occurrence in P was *contracted* and that P *β-contracts* to P', denoted $P \rhd_{1\beta} P'$. If a term P can be converted to term Q through a finite number of β-reductions and α-conversions, we say that P β-reduces to Q, denoted $P \rhd_\beta Q$.

Example:

$$\begin{aligned} (\lambda x.(\lambda y.yx)z)v &\rhd_{1\beta} [v/x]((\lambda y.yx)z) \equiv (\lambda y.yv)z \\ &\rhd_{1\beta} [z/y](yv) \equiv zv \end{aligned}$$

A term Q that has no β-redex is called a *β-normal form.* If a term P β-reduces to a term Q in β-normal form, then we say that Q is the *β-normal form* of P.

A term P is "β-equal" or "β-convertible" to term Q, denoted $P =_\beta Q$, if and only if Q can be obtained from P by a finite (possibile empty) sequence of β-reductions, reverse β-reductions and α-conversions. That is, $P =_\beta Q$ if and only if exists $P_0, ..., P_n (n \geq 0)$ such that:

$$(\forall i \leq n-1)(P_i \rhd_{1\beta} P_{i+1} \quad \text{or} \quad P_{i+1} \rhd_{1\beta} P_i \quad \text{or} \quad P_i =_\alpha P_{i+1},$$

$$P_0 \equiv P,$$

$$P_n \equiv Q.$$

A λ-term M is called *weakly normalizable* if it has a normal form. When all possible reduction sequences that start with M have finite length, then M is called a *strongly normalizable* term.

As an example, let us consider the λ-term that represents an ordered pair of elements:

- $\overline{pair} := \lambda x.\lambda y.\lambda z.zxy$

If m and n are terms, then the pair (m,n) is represented by the λ-term $\overline{pair}\ m\ n \equiv \lambda z.zmn$. The two projections of a pair are defined by the λ-terms:

- $\overline{first} := \lambda p.(p\ \overline{true})$;
- $\overline{second} := \lambda p.(p\ \overline{false})$.

where $\overline{true}$ and $\overline{false}$, also known an "Church Booleans", are the λ-terms that represent, respectively, the Boolean values true and false:

- $\overline{true} := \lambda x.\lambda y.x$
- $\overline{false} := \lambda x.\lambda y.y$

The fact that $\overline{first}\ (\overline{pair}\ m\ n)$ has m as its normal form is the result of the following sequence of reductions:

$$\begin{array}{rcl}
\overline{first}\ (\overline{pair}\ m\ n) & \equiv & (\lambda p.p\,\overline{true})(\lambda z.zmn) \\
& \rhd_{1\beta} & [(\lambda z.zmn)/p](\lambda p.p\,\overline{true}) \\
& \equiv & (\lambda z.zmn)\ \overline{true} \\
& \equiv & (\lambda z.zmn)(\lambda x.\lambda y.x) \\
& \rhd_{1\beta} & (\lambda x.\lambda y.x)mn \\
& \rhd_{1\beta} & m
\end{array}$$

Similarly, it can be shown that $\overline{second}\ (\overline{pair}\ m\ n) \rhd_{\beta} n$.

The rules of computation in the lambda calculus are also very simple, and can be grouped in a few cases. The idea of substitutions leads to the concept of reductions, which allows for the transformation of terms into equivalent ones. These reductions represent computations, and the equality of terms represents the equality of functions and values as we know from classical mathematics. Thus, a lambda expression represents a program, in the sense that it embeds an algorithm, takes arguments as inputs and produces a result. A reduction, on the other hand, represents a computation step, a change in the state of a computation. The result of a computation is represented by a term that can not be further reduced, and for this reason is called a "normal form". In summary:

LAMBDA EXPRESSION: Represents a *program*, an algorithm, a procedure to produce a result;

β-REDUCTION: Represents a *computation*, a passage from a program state no another, within the process of generating a result;

NORMAL FORM: The *result* of a computation, a value that cannot be further simplified;

β-EQUALITY: Relation between programs that produce the *same results*.

The lambda language and the theory of lambda calculus is powerful enough to represent data types as we know from programming languages, such as booleans and natural numbers, and also to represent the operations usually applied to them. Thus, lambda expressions can be seen as abstract representations of programs, with the same computational power one would expect from them.

Despite the simplicity and elegance of the Untyped Lambda Calculus, it was proved that equality of lambda terms can not be decided algorithmically (CHURCH, 1936). Also, it can not be decided if an untyped lambda term has a normal form or not. These results, discovered by Church, were the first undecidable problems to be proved in history. They were also used to prove the undecidability of first order predicate logic.

A more restricted version of the Untyped Lambda Calculus, the Typed Lambda Calculus was later developed in order to incorporate the notion of types, with important consequences for programming language design and Proof Theory. This calculus is the subject of the next section.

A.4 Typed Lambda Calculus

A *Type Theory* is a formal system that can be used to represent the foundations of mathematics, as an alternative for the more traditional set theory. It makes extensive use of the idea of *types*, which are abstract tags that are assigned to terms, specially those that are passed as arguments to functions and those that are reurned by function calls. This way, type checking can be performed on the terms constructed in such a theory, and this helps avoiding some of the inconsistencies found in other base theories, such as set theory. The Typed Lambda Calculus of Church, presented in this section, is itself a Type Theory, one of the first ones to be formulated.

Type Theory has its origins before Church, however. It was invented by Bertrand Russell in the 1910s, as an attempt to fix the inconsistencies in set theory, as a consequence of Russell's Paradox stated in the beginning of the 20th century ("is the set composed of all sets that are not members of themselves a member of itself?"). Since then, many different type theories have been created, used and developed.

The Typed Lambda Calculus is a variant from the untyped version, with type tags associated to the lambda terms (HINDLEY; SELDIN, 2008). Thus, all atomic elements have a type, and the type of a lambda term can be inferred from the type of its subterms and atomic elements. Types, in this case, are used to ensure that only arguments of the correct nature are passed to functions, and that functions are used in a consistent way inside other functions. The intention is to avoid inconsistencies and paradoxes, as they might occur in the Untyped Lambda Calculus. Its was created by Church in 1940 (CHURCH, 1940).

In the Typed Lambda Calculus, variables have base types and abstractions and applications create new types accordingly. Types must match in order for lambda terms to be considered well-formed. New types are obtained through the use of special inference rules, which in this case are called *typing rules*. A typing rule has the same aspect of an inference rule (as discussed in Section A.2), with the premises placed above an horizontal line and the conclusion placed below it.

The Typed Lambda Calculus is a less powerful model of computation than the corresponding untyped version, but offers type checking capabilities that can be used to ensure the correct use of applications and abstractions. When applied to computer programming language development, it leads to safer languages that can detect more errors, both statically and dynamically. Indeed, most of the safe languages used nowadays for computer programming take advantage of this type feature and implement it in different ways.

Differently from the Untyped Lambda Calculus, equality of lambda terms is decidable in the typed version. Also, it is decidable if a term has a normal form or not. It can be checked, for example, if two programs are equal (in the sense that they produce the same results for the same inputs) and also if a program produces a result for a given input (in this case it is enough to check if the corresponding expression reduces to a normal form).

It can also be proved that the set of all typed terms is strongly normalizable (HINDLEY; SELDIN, 2008). Thus, the Typed Lambda Calculus (at least in its traditional form) can not represent the class of partial recursive functions (i.e, the computable functions), but only a subclass of them consisting of the total recursive functions. Still, it is a very powerful calculus that can be used to model a large and important part of mathematics and computer sience. For these reasons, there is a lot of interest in the use of the Typed Lambda Calculus as a model of computation, despite being more restricted than the untyped version.

We present first, in Section A.4.1, the inference rules for the function type (represented by the symbol $\rightarrow$), which characterizes the so called *Simply Typed Lambda Calculus*. This calculus is then extended with new types in Section A.4.2, thus making it more expressive and easier to manipulate. Finally, the notion of a *dependent product*, which will considerably increase the power of the Typed Lambda Calculus, is introduced in Section A.4.3.

In what follows, "computer programs" is a synonym for "lambda terms" (typed or untyped, depending on the context), and "computation" means the application of a series of reductions to a lambda term (idem).

A.4.1 Simply Typed Lambda Calculus

This calculus has only two rules. The first is the introduction rule (or function abstraction, represented by $\rightarrow I$). This rule states that, if in a context where variable x has type σ, term t (eventually depending on x) has type τ, then the function that maps x to t has type $\sigma \rightarrow \tau$:

$$\frac{\begin{array}{c}[x:\sigma]\\ \vdots \\ t:\tau\end{array}}{\lambda x^{\sigma}.t : \sigma \rightarrow \tau}\ (\rightarrow I)$$

The notation $(\lambda x^{\sigma}.t^{\tau})^{\sigma\rightarrow\tau}$ is also used to express the fact that the abstraction has type $\sigma \rightarrow \tau$. The second rule is the elimination rule (or function application, represented by $\rightarrow E$), and asserts that the type of the application of a function to an argument of the correct type produces a result of the corresponding type:

$$\frac{f:\sigma\rightarrow\tau \qquad a:\sigma}{fa:\tau}\ (\rightarrow E)$$

Notation $(f^{\sigma\to\tau}a^{\sigma})^{\tau}$ is also used to express the fact that fa has type τ.

A *type inference tree* is a structure that has the objective of determining if a term is correctly type according to the rules of the system (in our present case, the two rules above), and also its type. As an example, we present next the type inference tree for the term:

$$\lambda x.\lambda y.\lambda z.xzy$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x : a \to (b \to c)] \qquad [z : a]}{xz : b \to c}\,(\to E) \qquad [y : b]}{xzy : c}\,(\to E)}{\lambda z^{a}.xzy : (a \to c)}\,(\to I)}{\lambda y^{b}.\lambda z^{a}.xzy : (b \to (a \to c))}\,(\to I)}{\lambda x^{a\to(b\to c)}.\lambda y^{b}.\lambda z^{a}.xzy : (a \to (b \to c)) \to (b \to (a \to c))}\,(\to I)$$

As a result, it can be noted that types $a \to (b \to c)$, b and a were associated, respectively, to variables x, y and z, and also that the type of the term $\lambda x.\lambda y.\lambda z.xzy$ is well-formed and corresponds to $(a \to (b \to c)) \to (b \to (a \to c))$.

A.4.2 New Types and Terms

We now introduce the inference rules used to represent new types and terms, in addition to the function type operator of Section A.4.1, and thus extend the Simply Typed Lambda Calculus into a more expressive calculus. The new types include the Cartesian product type (used to represent the logical conjunction), the disjoint union type (used to represent the logical disjunction), the negation, and the universal and existential quantifiers.

The Cartesian product type $(\times)$ is a λ-term that represents the pair formed by the argument terms. Its introduction rule is:

$$\dfrac{x : \sigma \qquad y : \tau}{\overline{conj}\; x\; y : \sigma \times \tau}\,(\times I)$$

where:

- $\overline{conj}$ is a primitive term that represents the pair formed by terms x and y, $\overline{conj}\; x\; y \equiv (x, y)$.

The elimination rules have as conclusions λ-terms that project the first and second elements of the pair:

$$\dfrac{p : \sigma \times \tau}{\overline{fst}\; p : \sigma}\,(\times E_1)$$

$$\dfrac{p : \sigma \times \tau}{\overline{snd}\; p : \tau}\,(\times E_2)$$

The following is an example of a type inference tree for the function that exchanges the positions of the elements in an ordered pair:

$$\lambda x^{a\times b}.\overline{conj}(\overline{snd}\,x)(\overline{fst}\,x)$$

$$\dfrac{\dfrac{\dfrac{[x : a\times b]}{\overline{snd}\,x : b}\,(\times E) \qquad \dfrac{[x : a\times b]}{\overline{fst}\,x : a}\,(\times E)}{\overline{conj}(\overline{snd}\,x)(\overline{fst}\,x) : b\times a}\,(\times I)}{\lambda x^{a\times b}.\overline{conj}(\overline{snd}\,x)(\overline{fst}\,x) : (a\times b)\to(b\times a)}\,(\to I)$$

The rules for the disjoint union type (represented by symbol $+$) are presented next. The first ones are the introduction rules, which have as conclusions terms that correspond to pairs:

$$\dfrac{x : \sigma}{\overline{inl}\,x : \sigma+\tau}\,(+I_1)$$

$$\dfrac{y : \tau}{\overline{inr}\,y : \sigma+\tau}\,(+I_2)$$

where $\overline{inl}$ and $\overline{inr}$ are defined as:

- $\overline{inl}\ x \equiv\ \overline{conj}\ \overline{0}\ x$;
- $\overline{inr}\ x \equiv\ \overline{conj}\ \overline{1}\ x$;
- $\overline{0} := \lambda x^{d\to d}.\lambda y^{d}.y : (d\to d)\to d\to d$;
- $\overline{1} := \lambda x^{d\to d}.\lambda y^{d}.xy : (d\to d)\to d\to d$.

where $\overline{0}$ and $\overline{1}$, also known an "Church Numerals", are the λ-terms that represent, respectively, the numerals 0 and 1.

The elimination rule, presented below, is explained in the following paragraphs after some introductory definitions:

$$\dfrac{p : \sigma+\tau \qquad \begin{matrix}[x : \sigma]\\ \vdots \\ q : \mu\end{matrix} \qquad \begin{matrix}[y : \tau]\\ \vdots \\ r : \mu\end{matrix}}{\overline{case}\,p\,(\lambda x^{\sigma}.q)\,(\lambda y^{\tau}.r) : \mu}\,(+E)$$

where $\overline{case}$ is defined as:

- $(\overline{case}\ u\ v\ w \equiv \overline{if}\,(\overline{zero}\,(\overline{fst}\,u))(v\,(\overline{snd}\,u))(w\,(\overline{snd}\,u)))$

where $\overline{zero}$ is the function that tests if the argument is zero and evaluates to $\overline{true}$ or $\overline{false}$, and $\overline{if}$ selects the second or third argument according the value of the first. In what follows, we consider:

- $\overline{true} := \lambda x^a.\lambda y^a.x : a \to a \to a$
- $\overline{false} := \lambda x^a.\lambda y^a.y : a \to a \to a$

and $d \equiv a \to a \to a$. Thus, we have:

- $\overline{zero} := \lambda x^{(d\to d)\to d\to d}.x(\lambda y^d.\overline{false})\,\overline{true} : ((d \to d) \to d \to d) \to d$
- $\overline{if} := \lambda x^d.\lambda y^a.\lambda z^a.xyz : d \to a \to a \to a$

For example, the following reductions show that $\overline{true}$ and $\overline{false}$ are, respectively, the normal forms for $\overline{zero}\,\overline{0}$ and $\overline{zero}\,\overline{1}$:

$$\begin{array}{rcl}
\overline{zero}\,\overline{0} & \equiv & (\lambda x^{(d\to d)\to d\to d}.x(\lambda y^d.\overline{false})\,\overline{true})(\lambda x^{d\to d}.\lambda y^d.y) \\
& \rhd_{1\beta} & (\lambda x^{d\to d}.\lambda y^d.y)(\lambda y^d.\overline{false})\,\overline{true} \\
& \rhd_{1\beta} & \overline{true} \\
\overline{zero}\,\overline{1} & \equiv & (\lambda x^{(d\to d)\to d\to d}.x(\lambda y^d.\overline{false})\,\overline{true})(\lambda x^{d\to d}.\lambda y^d.xy) \\
& \rhd_{1\beta} & (\lambda x^{d\to d}.\lambda y^d.xy)(\lambda y^d.\overline{false})\,\overline{true} \\
& \rhd_{1\beta} & (\lambda y^d.\overline{false})\,\overline{true} \\
& \rhd_{1\beta} & \overline{false}
\end{array}$$

Also, it is possible to show that $(\overline{if}\,\overline{true}\,m\,n)$ and $(\overline{if}\,\overline{false}\,m\,n)$ reduce, respectively, to m and n (if both m and n are of type a):

$$\begin{array}{rcl}
\overline{if}\,\overline{true}\,m\,n & \equiv & (\lambda x^d.\lambda y^a.\lambda z^a.xyz)(\lambda x^a.\lambda y^a.x)\,m\,n \\
& \rhd_{1\beta} & (\lambda x^a.\lambda y^a.x)\,m\,n \\
& \rhd_{1\beta} & m \\
\overline{if}\,\overline{false}\,m\,n & \equiv & (\lambda x^d.\lambda y^a.\lambda z^a.xyz)(\lambda x^a.\lambda y^a.y)\,m\,n \\
& \rhd_{1\beta} & (\lambda x^a.\lambda y^a.y)\,m\,n \\
& \rhd_{1\beta} & n
\end{array}$$

Returning to rule $+E$, we can now make $u = p$, $v = (\lambda x^\sigma.q)$ and $w = (\lambda y^\tau.r)$ and see that the conclusion $\overline{case}\,p\,(\lambda x^\sigma.q)\,(\lambda y^\tau.r)$ becomes:

$$\overline{if}\,(\overline{zero}\,(\overline{fst}\,p))((\lambda x^\sigma.q)\,(\overline{snd}\,p))((\lambda y^\tau.r)\,(\overline{snd}\,p)))$$

This term clearly states that the resulting term is a function application conditioned to the value of the first element of p. If it is $\overline{0}$, then the second element of p is of type σ, and so the term that maps x to q is used. Otherswise, the second element of p is of type τ, and so the term that maps y to r is used. In both cases, the resulting term has type μ.

As another example, the following is a type inference tree for the function that evaluates to the element that is either the first element of the disjoint union or the first element of the ordered pair that is the second element of the disjoint union:

$$\lambda p^{a+(a\times b)}.(\overline{case}\,p\,(\lambda x^a.x)(\lambda y^{a\times b}.\overline{fst}\,y))$$

$$\cfrac{\cfrac{p : a+(a\times b) \qquad \begin{array}{c}[x:a]\\ x:a\end{array} \qquad \cfrac{[y:a\times b]}{\overline{fst}\,y : a}\,{\scriptstyle(\times E)}}{\overline{case}\,p\,(\lambda x^a.x)(\lambda y^{a\times b}.\overline{fst}\,y) : a}\,{\scriptstyle(+E)}}{\lambda p^{a+(a\times b)}.(\overline{case}\,p\,(\lambda x^a.x)(\lambda y^{a\times b}.\overline{fst}\,y)) : (a+(a\times b)) \to a}\,{\scriptstyle(\to I)}$$

False (represented by symbol $\bot$) has no introduction rule, and its eliminaton rule (*ex-falso quodlibet*) is:

$$\cfrac{x:\bot}{\lambda\bot.x^{\bot} : P}\,{\scriptstyle(\bot E)}$$

In this case, $\lambda\bot.x^{\bot}$ should not be interpreted as a regular λ-term. Instead, it is just a representaton for an arbitrary proof of P, which is obtained in the presence of an invalid context.

Negation type (represented by symbol $\neg$) has introduction and elimination rules that are similar to those of the function type:

$$\cfrac{\begin{array}{c}[x:P]\\ \vdots\\ f:\bot\end{array}}{\lambda x^P.f : \neg P}\,{\scriptstyle(\neg I,\ \text{same as } \Rightarrow I)}$$

$$\cfrac{x:A \qquad y:\neg A}{yx:\bot}\,{\scriptstyle(\neg E,\ \text{same as } \Rightarrow E)}$$

The following is an example of a type inference tree for the function:

$$\lambda x^{a\to b}.\lambda z^{\neg b}.\lambda y^a.z(xy)$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[x:a\to b] \qquad [y:a]}{xy:b}\,{\scriptstyle(\to E)} \qquad [z:\neg b]}{z(xy):\bot}\,{\scriptstyle(\neg E)}}{\lambda y^a.z(xy):\neg a}\,{\scriptstyle(\to I)}}{\lambda z^{\neg b}.\lambda y^a.z(xy):\neg b\to\neg a}\,{\scriptstyle(\to I)}}{\lambda x^{a\to b}.\lambda z^{\neg b}.\lambda y^a.z(xy):(a\to b)\to(\neg b\to\neg a)}\,{\scriptstyle(\to I)}$$

The inference rules for the universal quantifier type (represented by symbol $\forall$) are:

$$\dfrac{\begin{array}{c}[x:D]\\ \vdots\\ f:P(x)\end{array}}{\lambda x^{D}.f:\forall x:D.P(x)}\;(\forall I)$$

This rule introduces the universal quantifier ∀ as a new type. This type corresponds to the *dependent product*, which is discussed in more detail on Section A.4.3, and allows for the construction of more general logic formulas. In particular, a term of this type is a function that maps the argument of the function to a type that depends on the value of the argument. The dependent product is a more general case of the arrow type, and can be reduced to a simple function type (respectively implication) when the body of the function does not depend on the argument.

The elimination rule corresponds to the application rule discussed before, and merely instantiates the universal quantifier for some value t with the correct type D, thus generating a new type $P(t)$:

$$\dfrac{t:D \qquad r:\forall x:D.P(x)}{rt:P(t)}\;(\forall E)$$

The inference rules for the existential quantifier type (represented by symbol ∃) are presented next. The introduction rule has as conclusion a term that represents the pair formed by the witness and the proof that this witness satisfies the property:

$$\dfrac{a:D \qquad f(a):P(a)}{\overline{conj}\;a\;f(a):\exists x:D.P(x)}\;(\exists I)$$

The elimination rules use the two pair projections in order to independently retrieve the witness and the proof that the property holds for this witness:

$$\dfrac{t:\exists x:D.P(x)}{\overline{fst}\;t:D}\;(\exists E_1)$$

$$\dfrac{t:\exists x:D.P(x)}{\overline{snd}\;t:P[\overline{fst}\;t/x]}\;(\exists E_2)$$

As a final example, we present the type inference tree for the term:

$$\lambda r^{\forall x:D.R(x,x)}.\lambda t^{D}.\overline{conj}\;t\;(rt)$$

$$\dfrac{\dfrac{\dfrac{[t:D] \qquad \dfrac{[r:\forall x:D.R(x,x)] \qquad [t:D]}{rt:R(t,t)}\;(\forall E)}{\overline{conj}\;t\;(rt):\exists y:D.R(t,y)}\;(\exists I)}{\lambda t^{D}.\overline{conj}\;t\;(rt):\forall t:D.\exists y:D.R(t,y)}\;(\forall I)}{\lambda r^{\forall x:D.R(x,x)}.\lambda t^{D}.\overline{conj}\;t\;(rt):\forall x:D.R(x,x)\to\forall t:D.\exists y:D.R(t,y)}\;(\to I)$$

This example tells us that the term that was typed is a proof of the proposition $\forall x:D.R(x,x)\to\forall t:D.\exists y:D.R(t,y)$.

A.4.3 Dependent Product

A *dependent product* is a type of the form $\forall v : A, B$, where A and B are types and v is a bound variable whose scope covers B. Note that v can have free occurrences in B.

The product $\forall v : A, B$ is the type of the functions that map any v of type A to a term of type B, where v may occur in B. This definition has the consequence that whenever v is not free in B, then the type of the result is fixed (B). Otherwise, the type of the result depends on the value of v. Thus, in the first case, $\forall v : A, B$ becomes the type $A \to B$, the type of functions that map A to B.

Also, according to the Curry-Howard Isomorphism (see Section A.6), the product $\forall v : A, B$ can be interpreted as the type of the functions that map any proof of A to a proof of B, where B depends on v. If B does not depend on v, then $\forall v : A, B$ reduces to a simple implication $A \Rightarrow B$, and the function simply maps a proof of A to a proof of B.

The introduction rule for the dependent product is:

$$\dfrac{\begin{array}{c}[v : A] \\ \vdots \\ t : B\end{array}}{\lambda v^A . t : \forall v : A, B}\ (\forall I)$$

The *Lam* rules, introduced in the previous sections, are particular cases of this more general rule, in the case where B does not mention v.

As an example, suppose that x is a natural number and from it we can conclude that $x \leq x$. Then, the introduction rule allows the generalization:

$$\dfrac{\begin{array}{c}x : nat \\ \vdots \\ H : x \leq x\end{array}}{\lambda x^{nat} . H : \forall x : nat, x \leq x}\ (\forall I)$$

The elimination rule for the dependent product is:

$$\dfrac{t_1 : \forall v : A, B \qquad t_2 : A}{t_1 t_2 : B\{t_2/v\}}\ (\forall E)$$

Similarly, in the case that B does not mention v, all substitutions will return the same B as the result. This rule corresponds to the more general case of the App rules introduced in the previous sections.

As an example, suppose we have a proof H of $\forall x : nat, x \leq x$. Then, the elimination rule asserts, for example that $H5$ is a proof of $5 \leq 5$:

$$\dfrac{H : \forall x : nat, x \leq x \qquad 5 : nat}{H5 : 5 \leq 5}\ (\forall E)$$

Dependent products are used in many different situations and have very different applications, depending on the implementation. Coq, in particular, makes extensive use of it, and some practical examples are presented in Section A.7.5.

A.5 Constructivism and BHK

So far, we have discussed how to state lemmas and theorems as formulas of predicate logic in Section A.1. Then, we introduced Natural Deduction in Section A.2 as a tool for proving formulas of the predicate logic using the Tarski interpretation. In Section A.3 we introduced a calculus for the representation of computable functions, which evolved in Section A.4 into a less powerful however safer calculus with the same purpose. Also in Section A.4, we showed how this calculus could be used to model parts of our predicate logic.

In the present section we will formally introduce the idea of a proof of a proposition, and how proofs for complex formulas can be obtained from proofs of its constituents. Also, we will discuss how this interpretation led to the idea of constructivism, an approach toward reasoning that is of special interest to computer science.

A constructive (also called intuitionistic) type theory is a type theory that uses a constructive (or intuitionistic) logic (THOMPSON, 1991). In a constructive logic, propositions are not simply associated to true or false, as in the Tarski tradition. Instead, the BHK (Brouwer, Heyting and Kolgomorov) interpretation is used, which asks for the *proof* of the validity of a statement. In a constructive type theory, every true proposition must be accompanied by a corresponding proof, and every logical connective used to build a new proposition is associated to an inference rule that is used to build the proof of the new proposition. The proof must explain how to build the object that validates the proposition. Thus, in a constructive logic, a proposition is considered true only if it is accompanied by the corresponding proof. In classical logic, in opposition, every proposition can be assigned a value (true or false), even if it can´t be proved.

In a constructive logic, the expression $x : \sigma$ is interpreted as "x is a proof of σ". A proof of $A \wedge B$, for example, is a pair (a,b) where a is a proof of A and b is a proof of B. Thus, $(a,b) : A \wedge B$. In the same way, every logical connective has an associated inference rule. Different rules apply for other logical connectives such as disjunction, implication and negation, and also the universal and existential quantifiers.

We will now revisit the inference rules and the examples of Section A.2, however showing for each connective how to build a proof of the proposition that uses such a connective, starting from the proofs of the corresponding argument propositions. The technique of Natural Deduction will be used again, however in an extended way. Observe that now every proposition is accompanied by the respective proof to the left of the ":" symbol.

The following are the inference rules for the implication connective ($\Rightarrow$). The first is the introduction rule:

$$\dfrac{\begin{array}{c}[x : a] \\ \vdots \\ y : b\end{array}}{\lambda x : a, y : a \Rightarrow b}\ (\Rightarrow I)$$

The idea, in this case, is very intuitive: a proof of an implication is a function that maps a proof of the premise into a proof of the conclusion. For the elimination rule, a simple application (function call) shows how to build a proof of the conclusion once a proof of the premise is available:

$$\frac{x : a \Rightarrow b \qquad y : a}{xy : b}\,(\Rightarrow E)$$

As an example, the following is a tree that shows the construction of a proof for the proposition $(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[x : a \Rightarrow (b \Rightarrow c)] \qquad [y : a]}{xy : b \Rightarrow c}\,(\Rightarrow E) \qquad [z : b]}{xyz : c}\,(\Rightarrow E)}{\lambda y.xyz : a \Rightarrow c}\,(\Rightarrow I)}{\lambda z.\lambda y.xyz : b \Rightarrow (a \Rightarrow c)}\,(\Rightarrow I)}{\lambda x.\lambda z.\lambda y.xyz : (a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}\,(\Rightarrow I)$$

The introduction rule for the conjunction ($\wedge$), presented below, states that a proof of a conjunction $a \wedge b$ is a pair of terms (x, y) where x is a proof of a and y is a proof of b:

$$\frac{x : a \qquad y : b}{(x, y) : a \wedge b}\,(\wedge I)$$

The elimination rule, on the other hand, states that a proof of either a or b can be obtained from a proof of $a \wedge b$ by using projections that select, respectively, the first and second elements of the proof:

$$\frac{x : a \wedge b}{fst\ x : a}\,(\wedge E_1)$$

$$\frac{x : a \wedge b}{snd\ x : b}\,(\wedge E_2)$$

The following illustrates the use of these rules in the construction of a proof term for the proposition $(a \wedge b) \Rightarrow (b \wedge a)$:

$$\cfrac{\cfrac{\cfrac{[x : a \wedge b]}{snd\ x : b}\,(\wedge E) \qquad \cfrac{[x : a \wedge b]}{fst\ x : a}\,(\wedge E)}{(snd\ x, fst\ x) : b \wedge a}\,(\wedge I)}{\lambda x.(snd\ x, fst\ x) : (a \wedge b) \Rightarrow (b \wedge a)}\,(\Rightarrow I)$$

The introduction rules for the disjunction ($\vee$) state that a proof is a pair of terms, where the first is a tag that indicates which argument of the disjuction the second element refers to. In other words, a proof of a disjunction is a proof of one of its arguments together with an information about the origin of the corresponding term.

$$\frac{x : a}{(\overline{0},x) : a \vee b}\ (\vee I_1)$$

$$\frac{x : b}{(\overline{1},x) : a \vee b}\ (\vee I_2)$$

The elimination rule for the disjunction looks for a common conclusion that can be obtained from both of the arguments. Thus, this conclusion is always valid in the presence of the disjunction and therefore it can be eliminated. The term to be selected as a proof of this common conclusion depends, of course, on the first element of the proof of the disjunction:

$$\frac{x : a \vee b \qquad \begin{matrix}[snd\ x : a]\\ \vdots \\ z_1 : c\end{matrix} \qquad \begin{matrix}[snd\ x : b]\\ \vdots \\ z_2 : c\end{matrix}}{if\ (fstx = \overline{0})\ then\ z_1\ else\ z_2 : c}\ (\vee E)$$

The following tree illustrates the use of these rules in the construction of the proof of the proposition $(a \vee (a \wedge b)) \Rightarrow a$:

$$\cfrac{\cfrac{[x : a \vee (a \wedge b)] \qquad \begin{matrix}[snd\ x : a]\\ snd\ x : a\end{matrix} \qquad \cfrac{[snd\ x : a \wedge b]}{fst\ (snd\ x) : a}\ (\wedge E)}{if\ (fst\ x = \overline{0})\ then\ (snd\ x)\ else\ (snd\ (fst\ x)) : a}\ (\vee E)}{\lambda x.if\ (fst\ x = \overline{0})\ then\ (snd\ x)\ else\ (snd\ (fst\ x)) : (a \vee (a \wedge b)) \Rightarrow a}\ (\Rightarrow I)$$

Inference rule for false ($\perp$) elimination states that from a proof of false any proposition can be proved. The term $\lambda \perp .x$ should be interpreted in an abstract way, and not as a proper lambda term as defined before. False can not be introduced as there is not proof of it.

$$\frac{x : \perp}{\lambda \perp .x : a}\ (\perp E)$$

For the negation ($\neg$), the rules are based on those of the implication ($\Rightarrow$):

$$\frac{\begin{matrix}[x : a]\\ \vdots \\ y : \perp\end{matrix}}{\lambda x.y : \neg\, a}\ (\neg\, I, \text{same as } \Rightarrow I)$$

$$\frac{x : \neg\, a \qquad y : a}{xy : \perp}\ (\neg\, E, \text{same as } \Rightarrow E)$$

The following illustrates the use of these rules in the construction of the proof of the proposition $(a \Rightarrow b) \Rightarrow (\neg\, b \Rightarrow \neg\, a)$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[x:a\Rightarrow b] \qquad [y:a]}{xy:b}\,(\Rightarrow E) \qquad [z:\neg\, b]}{z(xy):\bot}\,(\neg\, E)}{\lambda y.z(xy):\neg\, a}\,(\Rightarrow I)}{\lambda z.\lambda y.z(xy):\neg\, b\Rightarrow\neg\, a}\,(\Rightarrow I)}{\lambda x.\lambda z.\lambda y.z(xy):(a\Rightarrow b)\Rightarrow(\neg\, b\Rightarrow\neg\, a)}\,(\Rightarrow I)$$

Introduction rule for the universal quantifier ($\forall$) states that its proof consists of a function that maps any proof x of A to a proof p of P, where p and P might depend on x. No other asumptions used in p, besides $x:A$, might contain variable x free.

$$\cfrac{\begin{matrix}[x:A]\\ \vdots \\ p:P\end{matrix}}{(\lambda x:A.p):(\forall x:A.P)}\,(\forall I)$$

Elimination rule is simply a function application that instantiates the generalisation to a particular case:

$$\cfrac{t:(\forall x:A.P) \qquad a:A}{ta:P[a/x]}\,(\forall E)$$

The introduction rule for the existential quantifier ($\exists$) demands an element a (called *witness*) and a proof that P holds for this witness. This pair (formed by the witness and the proof) are then taken as the proof of the existential quantifier in this particular case:

$$\cfrac{a:A \qquad t:P[a/x]}{(a,t):\exists x:A.P}\,(\exists I)$$

The elimination rule uses two projections to extract, from a proof of an existential quantifier, the witness and the proof that the proposition holds for this witness:

$$\cfrac{t:(\exists x:A.P)}{fst\ t:A}\,(\exists E_1)$$

$$\cfrac{t:(\exists x:A.P)}{snd\ t:P[fst\ t/x]}\,(\exists E_2)$$

The following illustrates the use of these rules in the construction of the proof of the proposition $\forall x.R(x,x)\Rightarrow\forall x.\exists y.R(x,y)$:

$$\cfrac{\cfrac{\cfrac{\cfrac{[t:(\forall x:A.R(x,x))] \qquad [x:A]}{tx:R(x,x)}\,(\forall E)}{(x,tx):\exists y.R(x,y)}\,(\exists I)}{\lambda x.(x,tx):\forall x.\exists y.R(x,y)}\,(\forall I)}{\lambda t.\lambda x.(x,tx):(\forall x.R(x,x))\Rightarrow(\forall x.\exists y.R(x,y))}\,(\Rightarrow I)$$

As explained in Section A.1, constructivism does not use the Law of the Excluded Middle ($p \vee \neg p$) which belong to classical logic only. The reason is that this law enbodies exactly the fundamental idea that the constructivism rejects: that every proposition can be assigned a value that is either true or false, without any further justification (or proof). This law eventually shows up in equivalent statements, such as:

- Double negation $\neg(\neg p) \Rightarrow p$;
- Proof by contradiction $(\neg a \Rightarrow b) \wedge (\neg a \Rightarrow \neg b) \Rightarrow a$;
- Peirce's Law $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$.

The main characteristic of the constructivist ideology is, according to Troelstra (TROELSTRA, 2011):

> ... the insistence that mathematical objects are to be constructed (mental constructions) or computed; thus theorems asserting the existence of certain objects should by their proofs give us the means of constructing objects whose existence is being asserted.

Thus, the idea is that the truth of every proposition should only be inferred from the conclusion of a mental process, and not from arbitrary assignments. This mental process, in turn, should be built from logical steps derived from a precise calculus.

A constructive proof is said to have *computational content*, as it is possible to "construct" the object that validates the proposition (the proof is a recipe for building this object). A constructive proof enables (computer) code *extraction* from proofs, thus the interest for it in computer science.

Martin-Löf's Intuitionistic Type Theory (MARTIN-LÖF, 1980; NORDSTRÖM; PETERSSON; SMITH, 1990) is another example of an important Type Theory, and is widely used as the basis of modern constructive mathematics and also as the fundamental theory of many proof assistants. It was derived from the mathematical constructivism that appeared in the end of the 19th century.

Martin-Löf's Intuitionistic Type Theory is based on the first-order logic to represent logical propositions and types, and the Typed Lambda Calculus to represent programs and proofs of theorems. The whole idea is structured around the Curry-Howard Correspondence, which relates proofs to programs and propositions to types. It is a powerful theory for software development and interactive theorem proving, and is also used as a formal theory to represent the foundations of mathematics.

At his point, we are able of building proofs for formulas of predicate logic, using the BHK interpretation, and this would in principle be enough as an underlying theory for our formalization purposes. However, there are other possibilities that can significantly improve our ability to derive proofs in a constructive way. These possibilities arise from the direct relationship between types and propositions, and also between proofs and terms. The study of this relationship, which is summarized in the next section, has the important consequence of allowing the use of a mixture of reasoning and computing techniques in the formalization process.

A.6 Curry-Howard Correspondence

There is no reason, in principle, to consider that the two fundamental tools of mathematics - reasoning and computing - are related in any way. This is not the case, however, as stated by the Curry-Howard Correspondence (also known as "Equivalence" or "Isomorphism") (SORENSEN; URZYCZYN, 2006).

Initially observed by Haskell Curry in 1934 (CURRY, 1934), and later developed by Curry and Feys in 1958 (CURRY; FEYS, 1958) and William Howard in 1969 (HOWARD, 1980), the correspondence establishes a direct relationship between the logic and the computation models (the latter represented by the Typed Lambda Calculus in the present context).

On one hand, it is a fact that every typed lambda term can be associated to a type expression, which expresses its type (see Section A.4). This type expression, in our context, is called a "specification", because it denotes the signature of the corresponding program (i. e., the number and types of the arguments and the type of the result). On the other hand, it is also a fact that the proof of a proposition, represented by the tree built by a Natural Deduction process, puts one (the proof) in direct relation to the other (the proposition) (see Section A.5). Thus, (typed lambda) terms have specifications (types), and proofs (natural deduction trees) confirm the validity of propositions (theorems).

The Curry-Howard Correspondence is a syntactic analogy between terms and proofs, and also between specifications and propositions, which has deep consequences in the way one can reason about programs and develop proofs of theorems. Basically, it states that proofs and terms have a direct relationship: a term that satisfies a specification (that is, a typed lambda term that has a certain type) is, at the same time, a representation of a proof in natural deduction of the proposition represented by the specification. The converse is also true: the proof of a proposition can be taken as a term whose specification is the proposition itself.

Going further, the Curry-Howard Correspondence establishes that to prove a theorem (proposition) is the same as finding a term whose type (specification) matches the proposition. Thus, finding a proof of a theorem is the same as building a program. Also, to build a term that satisfies a certain specification is the same as proving the proposition that matches this specification. According to it, to build a proof of a proposition is the same as to build a program of a specific type, where the proposition represents the type of the program.

Another important consequence of the Curry-Howard Correspondence is that proof validation can be done via simple program type checking routines, which is usually an easy and efficient task. In summary, the Curry-Howard Correspondence states that building a program is the same as finding a proof, and both activities are interchangeable and highly interconnected.

The Correspondence opens up an immense area of research by exploring the relationships between programs and proofs and also between specifications and propositions. In particular, it allows program development to be considered as a formal mathematical activity, with all the related benefits. Also, it allows for computer programming skills to be introduced in the process of theorem proving, with many advantages as well. Is is also known as *Proofs-as-programs* or *Propositions-as-types* notions, as the four objects involved in it are *proofs*, *propositions (or theorems)*, *programs (or lambda terms)* and *types (or specifications)*.

As an example, the tree presented in Section A.1 explicits the relationship between a simple theorem and its proof (expressed as a tree in natural deduction). The relation between a program and a type can be examined from the following type derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x : a \to (b \to c)] \qquad [z : a]}{xz : b \to c}\,(\to E) \qquad [y : b]}{xzy : c}\,(\to E)}{\lambda z^{a}.xzy : (a \to c)}\,(\to I)}{\lambda y^{b}.\lambda z^{a}.xzy : (b \to (a \to c))}\,(\to I)}{\lambda x^{a \to (b \to c)}.\lambda y^{b}.\lambda z^{a}.xzy : (a \to (b \to c)) \to (b \to (a \to c))}\,(\to I)$$

which shows that the type of the term $\lambda x.\lambda y.\lambda z.xzy$ is $(a \to (b \to c)) \to (b \to (a \to c))$.

This tree and the tree of Section A.1 have as conclusions syntactically equivalent expressions, respectively:

- $(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$ (a proposition);
- $(a \to (b \to c)) \to (b \to (a \to c))$ (a type).

Thus, these formulas could be considered as a single object, and then interpreted in one way (as a proposition) or another (as a type) according to the context. Also, one can easily observe the similiarity between the structure of a proof tree in Natural Deduction (above) and the structure of a type inference tree for the same type/proposition in the type lambda calculus, reproduced again below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[a \Rightarrow (b \Rightarrow c)] \qquad [a]}{b \Rightarrow c}\,(\Rightarrow E) \qquad [b]}{c}\,(\Rightarrow E)}{a \Rightarrow c}\,(\Rightarrow I)}{b \Rightarrow (a \Rightarrow c)}\,(\Rightarrow I)}{(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}\,(\Rightarrow I)$$

It should be noted that, from the first to the second, it is enough to eliminate the λ-terms in the proof tree. The converse can be obtained by adding the corresponding λ-terms to the type inference tree. This is indeed a general rule which states that (i) to obtain a program from a proof it is only necessary to add λ-terms with the corresponding types, and (ii) to obtain a proof from a program it is enough to eliminate the λ-terms and keep the corresponding types.

The same phenomenon is observed when other logical connectives and type constructors are considered, besides only the implication/function type operator, and this is summarized in Table A.2.

Table A.2: Curry-Howard Correspondence

Logic	Typed lambda calculus
$\Rightarrow$ (implication)	$\rightarrow$ (function type)
$\wedge$ (and)	$\times$ (product type)
$\vee$ (or)	$+$ (sum type)
$\forall$ (forall)	Π (pi type)
$\exists$ (exists)	Σ (sigma type)
$\top$	unit type
$\bot$	bottom type

Types and propositions are indeed different interpretations of the same syntactic structure, an important consequence of the Curry-Howard Correspondence. Because of this, we can review the rest of the previous examples and examine the direct relationship between the formulas that represent types and propositions:

- $(a \times b) \rightarrow (b \times a)$ (type)
- $(a \wedge b) \Rightarrow (b \wedge a)$ (proposition)
- $(a + (a \times b)) \rightarrow a$ (type)
- $(a \vee (a \wedge b)) \Rightarrow a$ (proposition)
- $(a \rightarrow b) \rightarrow (\neg b \rightarrow \neg a)$ (type)
- $(a \Rightarrow b) \Rightarrow (\neg b \Rightarrow \neg a)$ (proposition)
- $\Pi x.R(x,x) \rightarrow \Pi x.\Sigma y.R(x,y)$ (type)
- $\forall x.R(x,x) \Rightarrow \forall x.\exists y.R(x,y)$ (proposition)

The same analogy can be established for the proofs and programs of these examples, as explained before. For proofs and programs, we have that:

- To build a program that satisfies a specification (type):
  - Interpret the specification as a theorem (proposition);

  - Build a proof tree for this theorem;
  - Add terms with the corresponding types.

- To build a proof of a theorem:
  - Interpret the theorem as a specification;
  - Build a program that meets the specification;
  - Remove the terms from the derivation tree.

The Curry-Howard Correspondence can then be summarized as follows:

- Types and propositions have the same syntactic stucture;
- Programs and proofs have the same syntactic stucture;
- To build a program is the same as to build a proof;
- To verify a program is the same as to verify a proof;
- Both verifications (proof and program) can be done via simple and efficient type checking algorithms.

At his point we have languages, calculus and techniques that can be extremely powerful for the statement of our theory, and also for proving it. The topics presented so far are, however, abstract theories that need to be implemented in order to become useful as a practical formalization tool. The Calculus of Constructions, presented in the next section, was developed exactly with this objective in mind.

A.7 Calculus of Constructions

The Calculus of Inductive Constructions, also known as CIC, (THE COQ DEVELOPMENT TEAM, 2015) is an extension of the Calculus of Constructions (CC) developed by Thierry Coquand (COQUAND; HUET, 1986), and is a constructive type theory that was designed specifically to be used as the mathematical language of the Coq proof assistant. It is the most powerful type system of all the versions considered in the lambda cube of Barendregt (BARENDREGT, 1991).

The Calculus of Constructions provides a uniform notation to represent programs, proofs, types and propositions. Logical connectives can be easily defined, as well as mathematical objects and their corresponding types. Reasoning and computing is accomplished with a small set of inference rules. After some time, an induction mechanism was added to it, allowing types and propositions to be defined inductively, along with mechanisms to reason about these inductively defined objects, thus leading to the name Calculus of Inductive Constructions. Basically, it is a Typed Lambda Calculus with a rich type system extended with inductive features.

All logical operators ($\rightarrow, \wedge, \vee, \neg$ and $\exists$) are defined in terms of the universal quantifier ($\forall$), using "dependent types" (see Section A.7.5), a key characteristic of the Calculus of Inductive Constructions along with higher-order types.

This section makes a short introduction to the Calculus of Constructions, presenting its main characteristics and inference rules. Rules for inductive definitions are not included and will be informally presented when the Coq implementation of the Calculus of Constructions is discussed in Section B.5.

We start in Section A.7.1 by introducing the terminology and notation used. The notions of types and sorts are introduced in Section A.7.2. Then, we present the basic type operator (the function type or implication, depending on the sort considered) in Section A.7.3 and Section A.7.4 (respectively). They correspond to the rules of the Simply Typed Lambda Calculus and serve to introduce the more complex and powerful type constructs, including polymorphism, dependent types and higher-order types, which are briefly discussed in Section A.7.5.

A.7.1 Terminology and Notation

Different names are used to denote a variety of notions in the CC. The following is a brief description of the main names used in the next sections (in alphabetical order), along with the corresponding notions.

CONCLUSION the bottom part of an inference rule, obtained from the premises on top.

CONTEXT the local declarations and definitions on which a proof script is built.

ENVIRONMENT the global declarations and definitions on which a proof script is built.

EXPRESSION the λ-term that represents a program.

INFERENCE RULE a rule used to ensure that a term is correcly formed; it consists of a set of premises and a conclusion separated from each other by an horizontal line.

PREDICATE a parametrized proposition.

PREMISE the upper part of an inference rule, the information used to derive the conclusion in the bottom.

PROGRAM a term that produces a value (a normal form) by means of a series of reductions, as in the Typed Lambda Calculus.

PROOF a term that asserts the validity of a proposition.

PROOF SCRIPT a sequence of high-level commands (tactics) that are based on the inference rules of the system and aid in the construction of a proof.

PROPOSITION a formula that represents a logical statement.

SORT the type of a type.

SPECIFICATION the type of a program, a formula that describes a property of a program.

TERM a well-formed formula, that is, a formula that satisfies the rules of the type system. A term is a generic designation that encompasses formulas representing proofs, programs, types and propositions.

TYPE a description of the nature and the set of the values manipulated by a program; in the Calculus of Constructions, every term is typed, every type is a term and therefore every type also has a type.

UNIVERSE infinite hierarchy of sorts.

All inference rules presented so far use formulas and terms. The Calculus of Constructions, however, uses a different notation based on the notion of *sequents*.

In the inference rules presented in the next sections, E and Γ represent, respectively, the *Environment* and the *Context* for the inference. The environment includes all global name definitions and the context includes all local definitions.

The empty context is represented by $[]$. If x has type A (or x is a proof of proposition A), this is a declaration that is represented by $x : A$. Multiple declarations are denoted $[x_1 : A_1; \ldots; x_n : A_n]$.

In what follows, a *typing judgement* is an expression $\Gamma \vdash t : T$, used to state the fact that the term t has type T in the context Γ. Valid typing judgements are obtained by the successive application of a set of different *typing (or inference) rules*. A typing rule is a rule that allows one to infer the type of an expression from a set of premises. As before, the premises are placed above an horizontal line and the conclusion appears below the same line.

The fact that a declaration belongs to some context is represented by $(x : A) \in \Gamma$. A typing judgement in some combination of an environment and context is denoted $E, \Gamma \vdash t : B$.

The sequent notation is equivalent to the notation that uses formulas and terms. For example, the $\rightarrow I$ rule presented in Section A.4 is presented again in Section A.7.3 using the new notation, under the name *Lam*. The same happens with rules $\rightarrow E$ and App.

The typing rules presented next are not exhaustive and cover only the simply Typed Lambda Calculus (with the arrow type) and the dependent product.

A.7.2 Types and Sorts

The notation of the Calculus of Constructions allows types (propositions) and programs (proofs) to have the same syntactical structure. All terms are typed in the CC, including the types themselves. Since every type is also a term, types are also assigned to types. The type of a type is called its "sort", and the set of all sorts is formed by *Prop*, *Set* and $Type(i), i \geq 1$, with the following relation:

$$Prop : Type(1)$$

$$Set : Type(1)$$

$$Type(i) : Type(i+1), i \geq 1$$

The terms $Types(i)$ are called *universes* and constitute an infinite family of types that are used to define the types of all terms in the system.

The nature and uses of the sorts *Set* and *Prop* are described in the following sections, along with the inference rules used in each case for constructing, respectively, correct specifications and propositions (as in the Simply Typed Lambda Calculus).

A.7.3 Set and Function Types

Set is the sort of data types and program specifications. For example, *nat* (which is the type of natural numbers) is of sort *Set*, and this is denoted $nat : Set$. The type of Boolean values is *bool*, which is also of sort *Set*, represented by $bool : Set$.

Set represents well-formed specifications (type expressions) and a term of type *Set* is called a specification. A term whose type is an specification is a program, and this specification represents the type of the corresponding program. For example, $nat \rightarrow bool$ is a type expression for a function that takes a natural number as argument and returns a Boolean value. Thus, $nat \rightarrow bool : Set$ and any term f such that $f : nat \rightarrow bool$ is a program that takes a natural number as argument and returns a Boolean value. In other words, $nat \rightarrow bool$ is the type of all programs that take a natural number as argument and return a Boolean value.

The symbol $\rightarrow$ is used, in this context, to represent the *function type*, that is, the type of the functions that maps values from one data type to values of another data type. The type $A \rightarrow B$, for example, is the type of all functions that map from type A to type B.

The *Var* rule states that if $x : A$ is placed in either the environment or the context, then x is recognized as a variable of type A in the union of both.

$$Var \ \frac{(x : A) \in E \cup \Gamma}{E, \Gamma \vdash x : A}$$

As an example, consider that $x : nat$ is placed in the context. Then, x is recognized as a variable of type *nat* in the union of the environment to the current context.

An expression is built from identifiers and some operators. The two fundamental operators for expressions are abstraction and application, represented by the following inference rules.

The *Lam* rule is used to build the term that represents abstraction and its type. It expresses the fact that, if the identifier v has type A in the environment or the context, and from this fact it is possible to conclude that term e has type B, then it is possible to build a function that maps v to e and has type $A \rightarrow B$. Types A and B denote single identifiers or new types constructed with the $Prod-set$ rule presented below.

$$Lam \quad \frac{E,\Gamma :: (v:A) \vdash e:B}{E,\Gamma \vdash fun\ v:A \Rightarrow e:A \rightarrow B}$$

As an example, suppose the context contains $x:nat$ and from it one can conclude that $y:bool$. Then, the *Lam* rule states that $fun\ x:nat \Rightarrow y$ is a well-formed term and has type $nat \rightarrow bool$.

The App rule is used to represent function application and its type.

$$App \quad \frac{E,\Gamma \vdash e_1 : A \rightarrow B \quad E,\Gamma \vdash e_2 : A}{E,\Gamma \vdash e_1 e_2 : B}$$

Suppose that $f:nat \rightarrow bool$ and $x:nat$ are placed in the context. Then, the App rules states that the term fx has type $bool$.

Finally, it should be stressed that in the CC all type expressions are also terms. Since all terms are typed, then these type expressions should also have their own types. The inference rule used to determine the type of a type specification built with the function type operator is $Prod-Set$.

$$Prod-Set \quad \frac{E,\Gamma \vdash A:Set \quad E,\Gamma \vdash B:Set}{E,\Gamma \vdash A \rightarrow B:Set}$$

This rule states that all type specifications built from the function type operator are elements of Set. For example, since $nat:Set$ and $bool:Set$, then the type expression $nat \rightarrow bool$ has type Set, denoted $nat \rightarrow bool:Set$.

It is useful to review the different categories of terms considered in the section, along with the corresponding types and sorts. Note that terms at level i have their types as terms at level $i+1$, and also that level 0 is not the type of any term in the system.

$$\begin{cases} Level\ 0: & Programs\ and\ values \\ Level\ 1: & Specifications \\ Level\ 2: & Set \\ Level\ 3: & Type(1) \\ \ldots & \\ Level\ i+2: & Type(i) \end{cases}$$

A.7.4 Prop and Implication

Prop represents the type of the propositions. That is, all propositions belong to type *Prop* and all elements of the sort *Prop* are well-formed propositions. A term whose type is a proposition is a proof of that proposition. As an example, let P and Q be propositions, $P : Prop$ and $Q : Prop$. Then, the implication $P \rightarrow Q$ is also a proposition, $P \rightarrow Q : Prop$. Also, any term H such that $H : P$ is called a “proof” of proposition P.

A well-formed proposition is not necessarily a provable proposition. A well-formed proposition is true only when accompanied by the corresponding proof.

In what follows, E denotes the set of axioms (global definitions) and Γ the set of assumptions (local definitions) available to the proof. Due to the Curry-Howard Correspondence, it is not a coincidence that the names of the rules presented in this section coincide with the names used in Section A.7.3.

The symbol $\rightarrow$ is used, in this context, to represent the logical connective *implication*. It represents the type of all functions that map a proof of proposition A to a proof of proposition B. It is in direct analogy to the function type operator introduced in the previous section.

The *Var* rule states that if $x : P$ is placed in either the environment or the context, then x is a proof of the proposition P in the union of both.

$$Var \ \frac{(x : P) \in E \cup \Gamma}{E, \Gamma \vdash x : P}$$

As an example, consider that $H : P$ is in the context. Then, H is recognized as a proof of proposition P in the union of the environment to the current context.

The *Lam* rule expresses the fact that, if P and Q are propositions and H is a proof of P, and from this fact it is possible to obtain a proof t of Q, then the function that maps H to t is a proof of the implication $P \rightarrow Q$.

$$Lam \ \frac{E, \Gamma :: (H : P) \vdash t : Q}{E, \Gamma \vdash fun\ H : P \Rightarrow t : P \rightarrow Q}$$

As an example, suppose the context contains $H_1 : P$ and from it one can conclude that $H_2 : Q$. Then, $fun\ H_1 : P \Rightarrow H_2$ is a well-formed term and is a proof of the implication $P \rightarrow Q$.

The *App* rule implements the *modus ponens* rule by stating that a proof of proposition Q can be obtained from a proof of proposition P.

$$App \ \frac{E, \Gamma \vdash t : P \rightarrow Q \quad E, \Gamma \vdash t' : P}{E, \Gamma \vdash tt' : Q}$$

Suppose that P and Q are propositions and $H_1 : P \rightarrow Q$ and $H_2 : P$ are in the context. Then, the *App* rule states that $H_1 H_2$ is a proof of Q.

The following rule asserts that the new propositions, built from previously existing propositions and the implication logical connective, are also in *Prop*:

$$Prod-Prop \quad \frac{E,\Gamma \vdash P : Prop \quad E,\Gamma \vdash Q : Prop}{E,\Gamma \vdash P \rightarrow Q : Prop}$$

For example, if P and Q are propositions, then $P \rightarrow Q$ is also a proposition.

The hierarchy of terms and corresponding types and sorts of this section can be summarized as follows. Similarly to the previous section, terms at level i have their types as terms at level $i+1$, and level 0 is not the type of any term in the system.

$$\begin{cases} Level\ 0: & Proofs \\ Level\ 1: & Propositions \\ Level\ 2: & Prop \\ Level\ 3: & Type(1) \\ \ldots & \\ Level\ i+2: & Type(i) \end{cases}$$

A.7.5 Dependent Product

Dependent products are used, in general, to specify functions whose return type depends on an argument, and has many different applications as explained next.

The special case where the return type depends on the type of the argument corresponds to the definition of *higher-order types*. Higher-order types is a powerful mechanism, used to represent operators on data types (such as lists, cartesian products and disjoint unions) and also standard logical connectives and quantifiers (such as and, or and existential quantifier).

Dependent products are used to represent universal quantification over values and types, leading to rich statements that are useful for describing complex theorems and program specifications. In the special case where the type of the argument is *Set* and the type of the result is either *Set* or *Prop*, this characterizes *dependent types* which can be used, for example, to respectively represent parametrized data types (data types that depend on a value) and predicates (propositions that depend on a value).

A *dependent type* is a type that depends on some value. Dependent types is an important feature of the Calculus of Constructions, as it allows the definition of richer types, which in turn can be used to make more complete and detailed descriptions of the properties of programs. For example, a natural value can be used to select a specific proposition from a set of different possible propositions. Since propositions are types, this means that we can select from a set of different types.

The App and *Lam* rules presented below use the sequent notation adopted by the Calculus of Constructions, and the Gallina syntax for term construction, and correspond, respectively, to the $\forall E$ and $\forall I$ rules of Section A.4.3:

$$App \quad \frac{E,\Gamma \vdash t_1 : \forall v : A, B \quad E,\Gamma \vdash t_2 : A}{E,\Gamma \vdash t_1 t_2 : B\{t_2/v\}}$$

$$Lam\ \frac{E,\Gamma :: (v : A) \vdash t : B}{E,\Gamma \vdash fun\ v : A \Rightarrow t : \forall v : A, B}$$

For the *App* rule, since v might appear free in B, it is necessary to substitute all occurrences of v by t_2 in B.

In the special case that v does not occur in B, then t_1 is of type $A \to B$ and the rule reduces to the form presented in Section A.7.3. The same happens with rule *Lam* if v does not appear in B. In this case, similarly, $fun\ v : A \Rightarrow t$ becomes of type $A \to B$.

The following discussion is based on the most recent source of information on the Calculus of Constructions, chapter 4 of the Coq Reference Manual (THE COQ DEVELOPMENT TEAM, 2015). There are seven inference rules for the dependent product (expanded from the three general cases considered in the Manual):

1.
$$\frac{E,\Gamma \vdash (A : Set) \quad E,\Gamma :: (a : A) \vdash B : Prop}{E,\Gamma \vdash \forall a : A, B : Prop}$$

2.
$$\frac{E,\Gamma \vdash (A : Prop) \quad E,\Gamma :: (a : A) \vdash B : Prop}{E,\Gamma \vdash \forall a : A, B : Prop}$$

3.
$$\frac{E,\Gamma \vdash (A : Type) \quad E,\Gamma :: (a : A) \vdash B : Prop}{E,\Gamma \vdash \forall a : A, B : Prop}$$

4.
$$\frac{E,\Gamma \vdash (A : Set) \quad E,\Gamma :: (a : A) \vdash B : Set}{E,\Gamma \vdash \forall a : A, B : Set}$$

5.
$$\frac{E,\Gamma \vdash (A : Prop) \quad E,\Gamma :: (a : A) \vdash B : Set}{E,\Gamma \vdash \forall a : A, B : Set}$$

6.
$$\frac{E,\Gamma \vdash (A : Type) \quad E,\Gamma :: (a : A) \vdash B : Type}{E,\Gamma \vdash \forall a : A, B : Type}$$

7.
$$\frac{E,\Gamma \vdash (A : Type) \quad E,\Gamma :: (a : A) \vdash B : Set}{E,\Gamma \vdash \forall a : A, B : Set}$$

Let´s now consider the most important rules, the types that they represent and some examples corresponding to their practical use. Rules 3 and 7, respectively, cause the types *Prop* and *Set* to be *impredicative*. An impredicative type is a type that allows the construction of a new element of the type by generalizing over all the elements of the same type. These two rules were included in Coq until version 7. From version 8 on, rule 7 was dropped and can now be invoked only via a special command explicitly issued by the user. *Prop* (due to rule 3) remains the only impredicative type.

1. This rule can be used to represent propositions quantified over a data type (an element of the sort *Set*).

   - The proposition that asserts the reflexivity of the relational operator less than or equal to ($\leq$) is $\forall a : nat, a \leq a$. To check that this proposition is well-formed, make $A = nat$ and $B = a \leq a$:

$$\frac{E, \Gamma \vdash (nat : Set) \quad E, \Gamma :: (a : nat) \vdash a \leq a : Prop}{E, \Gamma \vdash \forall a : nat, a \leq a : Prop}$$

   - The simplification of the Boolean conjunction with *true* can be stated as $\forall a : bool, a \wedge true = a$. To check that this statement is well-formed, make $A = bool$ and $B = a \wedge true = a$:

$$\frac{E, \Gamma \vdash (bool : Set) \quad E, \Gamma :: (a : bool) \vdash a \wedge true = a : Prop}{E, \Gamma \vdash \forall a : bool, a \wedge true = a : Prop}$$

2. This rule can be used to represent logical implications.

   - To check that the proposition $P \to Q \to R$ is well-formed, make $A = P$ and $B = Q$:

$$\frac{E, \Gamma \vdash (P : Prop) \quad E, \Gamma :: (a : P) \vdash Q : Prop}{E, \Gamma \vdash \forall a : P, Q : Prop}$$

   Since Q does not depend on a, then $\forall a : P, Q$ can be simplified to $P \to Q$:

$$\frac{E, \Gamma \vdash (P : Prop) \quad E, \Gamma :: (a : P) \vdash Q : Prop}{E, \Gamma \vdash P \to Q : Prop}$$

   Make $A = P \to Q$ and $B = R$:

$$\frac{E, \Gamma \vdash (P \to Q : Prop) \quad E, \Gamma :: (a : P \to Q) \vdash R : Prop}{E, \Gamma \vdash \forall a : P \to Q, R : Prop}$$

   Since R does not depend on a, then $\forall a : P \to Q, R$ can be simplified to $P \to Q \to R$:

$$\frac{E, \Gamma \vdash (P \to Q : Prop) \quad E, \Gamma :: (a : P \to Q) \vdash R : Prop}{E, \Gamma \vdash P \to Q \to R : Prop}$$

   - The proposition that asserts the commutativity of the conjunction can be defined as $P \wedge Q \to Q \wedge P$. To check that this proposition is well-formed, make $A = P \wedge Q$ and $B = Q \wedge P$:

$$\frac{E, \Gamma \vdash (P \wedge Q : Prop) \quad E, \Gamma :: (a : P \wedge Q) \vdash Q \wedge P : Prop}{E, \Gamma \vdash \forall a : P \wedge Q, Q \wedge P : Prop}$$

Since $Q \wedge P$ does not depend on a, then $\forall a : P \wedge Q, Q \wedge P$ can be simplified to $P \wedge Q \rightarrow Q \wedge P$:

$$\frac{E,\Gamma \vdash (P \wedge Q : Prop) \quad E,\Gamma :: (a : P \wedge Q) \vdash Q \wedge P : Prop}{E,\Gamma \vdash P \wedge Q \rightarrow Q \wedge P : Prop}$$

4. This rule can be used to represent function types.

- The type of the function that returns the sum of two natural numbers is $nat \rightarrow nat \rightarrow nat$. To check that this type is well-formed, make $A = B = nat$ and apply the rule:

  $$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash nat : Set}{E,\Gamma \vdash \forall a : nat, nat : Set}$$

  Since nat does not depend on a, change $\forall a : nat, nat$ by $nat \rightarrow nat$:

  $$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash nat : Set}{E,\Gamma \vdash nat \rightarrow nat : Set}$$

  Make $A = nat$ and $B = nat \rightarrow nat$ and apply the rule:

  $$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash nat \rightarrow nat : Set}{E,\Gamma \vdash \forall a : nat, nat \rightarrow nat : Set}$$

  Since $nat \rightarrow nat$ does not depend on a, change $\forall a : nat, nat \rightarrow nat$ by $nat \rightarrow nat \rightarrow nat$:

  $$\frac{E,\Gamma \vdash nat : Set \quad E,\Gamma :: (a : nat) \vdash nat \rightarrow nat : Set}{E,\Gamma \vdash nat \rightarrow nat \rightarrow nat : Set}$$

- The type of the function that compares two natural numbers is $nat \rightarrow nat \rightarrow bool$. To check that this type is well-formed, make $A = nat$ and $B = bool$ and apply the rule:

  $$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash bool : Set}{E,\Gamma \vdash \forall a : nat, bool : Set}$$

  Since $bool$ does not depend on a, change $\forall a : nat, bool$ by $nat \rightarrow bool$:

  $$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash bool : Set}{E,\Gamma \vdash nat \rightarrow bool : Set}$$

  Make $A = nat$ and $B = nat \rightarrow bool$ and apply the rule:

  $$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash nat \rightarrow bool : Set}{E,\Gamma \vdash \forall a : nat, nat \rightarrow bool : Set}$$

Since $nat \rightarrow bool$ does not depend on a, change $\forall a : nat, nat \rightarrow bool$ by $nat \rightarrow nat \rightarrow bool$:

$$\frac{E,\Gamma \vdash nat : Set \quad E,\Gamma :: (a : nat) \vdash nat \rightarrow bool : Set}{E,\Gamma \vdash nat \rightarrow nat \rightarrow bool : Set}$$

6. This rule can be used to represent higher-order types, including the type of logical connectives, polymorphic data types and predicates.

- The conjunction propositional operator ($\wedge$) has type $Prop \rightarrow Prop \rightarrow Prop$. To check that this type is well-formed, make $A = B = Prop$:

$$\frac{E,\Gamma \vdash (Prop : Type) \quad E,\Gamma :: (a : Prop) \vdash Prop : Type}{E,\Gamma \vdash \forall a : Prop, Prop : Type}$$

Since $Prop$ does not depend on a, then $\forall a : Prop, Prop$ can be simplified to $Prop \rightarrow Prop$:

$$\frac{E,\Gamma \vdash (Prop : Type) \quad E,\Gamma :: (a : Prop) \vdash Prop : Type}{E,\Gamma \vdash Prop \rightarrow Prop : Type}$$

Make $A = Prop \rightarrow Prop$ and $B = Prop$:

$$\frac{E,\Gamma \vdash (Prop \rightarrow Prop : Type) \quad E,\Gamma :: (a : Prop \rightarrow Prop) \vdash Prop : Type}{E,\Gamma \vdash \forall a : Prop \rightarrow Prop, Prop : Type}$$

Since $Prop$ does not depend on a, then $\forall a : Prop \rightarrow Prop, Prop$ can be simplified to $Prop \rightarrow Prop \rightarrow Prop$:

$$\frac{E,\Gamma \vdash (Prop \rightarrow Prop : Type) \quad E,\Gamma :: (a : Prop \rightarrow Prop) \vdash Prop : Type}{E,\Gamma \vdash Prop \rightarrow Prop \rightarrow Prop : Type}$$

Other logical connectives can be defined in a similar way.

- The type of the Cartesian product ($*$) can be defined as $Set \rightarrow Set \rightarrow Set$. To check that this type is well-formed, make $A = B = Set$:

$$\frac{E,\Gamma \vdash (Set : Type) \quad E,\Gamma :: (a : Set) \vdash Set : Type}{E,\Gamma \vdash \forall a : Set, Set : Type}$$

Since Set does not depend on a, then $\forall a : Set, Set$ can be simplified to $Set \rightarrow Set$:

$$\frac{E,\Gamma \vdash (Set : Type) \quad E,\Gamma :: (a : Set) \vdash Set : Type}{E,\Gamma \vdash Set \rightarrow Set : Type}$$

Make $A = Set \rightarrow Set$ and $B = Set$:

$$\frac{E,\Gamma \vdash (Set \rightarrow Set : Type) \quad E,\Gamma :: (a : Set \rightarrow Set) \vdash Set : Type}{E,\Gamma \vdash \forall a : Set \rightarrow Set, Set : Type}$$

Since Set does not depend on a, then $\forall a : Set \rightarrow Set, Set$ can be simplified to $Set \rightarrow Set \rightarrow Set$:

$$\frac{E,\Gamma \vdash (Set \rightarrow Set : Type) \quad E,\Gamma :: (a : Set \rightarrow Set) \vdash Set : Type}{E,\Gamma \vdash Set \rightarrow Set \rightarrow Set : Type}$$

Other polymorphic data types can be defined in a similar way.

- The predicate *less than or equal to* ($\leq$) over natural numbers has type $nat \rightarrow nat \rightarrow Prop$. To check that this type is well-formed, make $A = nat$ and $B = Prop$:

$$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash Prop : Type}{E,\Gamma \vdash \forall a : nat, Prop : Type}$$

Since $Prop$ does not depend on a, $\forall a : nat, Prop$ can be simplified to $nat \rightarrow Prop$:

$$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash Prop : Type}{E,\Gamma \vdash nat \rightarrow Prop : Type}$$

Make $A = nat$ and $B = nat \rightarrow Prop$:

$$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash nat \rightarrow Prop : Type}{E,\Gamma \vdash \forall a : nat, nat \rightarrow Prop : Type}$$

Since $nat \rightarrow Prop$ does not depend on a, $\forall a : nat, nat \rightarrow Prop$ can be simplified to $nat \rightarrow nat \rightarrow Prop$:

$$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash nat \rightarrow Prop : Type}{E,\Gamma \vdash nat \rightarrow nat \rightarrow Prop : Type}$$

The formation of some types require the application of more than one rule. The following examples show how this can be done:

- Suppose *array* represents the type of an aggregate with n elements of the same data type. Thus, *array* has type $nat \rightarrow Set$. To check this is a well-formed type, make $A = nat$ and $B = Set$ in rule 6:

$$\frac{E,\Gamma \vdash (nat : Type) \quad E,\Gamma :: (a : nat) \vdash Set : Type}{E,\Gamma \vdash \forall a : nat, Set : Type}$$

Since Set does not depend on a, then $\forall a : nat, Set$ can be simplified into $nat \rightarrow Set$:

$$\frac{E,\Gamma \vdash (nat : Type) \quad E,\Gamma :: (a : nat) \vdash Set : Type}{E,\Gamma \vdash nat \rightarrow Set : Type}$$

The check the fact that all arrays have a well-formed type, despite of the value of n, make $A = nat$ and $B = array\ a$ in rule 4:

$$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (a : nat) \vdash array\ a : Set}{E,\Gamma \vdash \forall a : nat, array\ a : Set}$$

- Now suppose that $matrix$ represents the type of a bidimensional aggregate. Thus, $matrix$ has type $nat \rightarrow nat \rightarrow Set$, which can be checked with two applications of rule 6:

$$\frac{E,\Gamma \vdash (nat : Type) \quad E,\Gamma :: (a : nat) \vdash Set : Type}{E,\Gamma \vdash nat \rightarrow Set : Type}$$

$$\frac{E,\Gamma \vdash (nat : Type) \quad E,\Gamma :: (a : nat) \vdash nat \rightarrow Set : Type}{E,\Gamma \vdash nat \rightarrow nat \rightarrow Set : Type}$$

A function that multiplies two $matrix$ has type $\forall\ m\ n\ p, matrix\ m\ n \rightarrow matrix\ n\ p \rightarrow matrix\ m\ p$. To check that this type is well-formed, and has type Set, make $A = matrix\ m\ p$ and $B = matrix\ n\ p$ in rule 4:

$$\frac{E,\Gamma \vdash (matrix\ n\ p : Set) \quad E,\Gamma :: (a : matrix\ n\ p) \vdash matrix\ m\ p : Set}{E,\Gamma \vdash \forall a : matrix\ n\ p, matrix\ m\ p : Set}$$

Since $matrix\ m\ p$ does not mention a, then:

$$\frac{E,\Gamma \vdash (matrix\ n\ p : Set) \quad E,\Gamma :: (a : matrix\ n\ p) \vdash matrix\ m\ p : Set}{E,\Gamma \vdash matrix\ n\ p \rightarrow matrix\ m\ p : Set}$$

Similarly:

$$\frac{E,\Gamma \vdash (matrix\ m\ n : Set) \quad E,\Gamma :: (a : matrix\ m\ n) \vdash matrix\ n\ p \rightarrow matrix\ m\ p : Set}{E,\Gamma \vdash matrix\ m\ n \rightarrow matrix\ n\ p \rightarrow matrix\ m\ p : Set}$$

Generalisation over p is obtained by a new application of rule 4:

$$\frac{E,\Gamma \vdash (nat : Set) \quad E,\Gamma :: (p : nat) \vdash matrix\ m\ n \rightarrow matrix\ n\ p \rightarrow matrix\ m\ p : Set}{E,\Gamma \vdash \forall p : nat, matrix\ m\ n \rightarrow matrix\ n\ p \rightarrow matrix\ m\ p : Set}$$

Similarly, two new applications of this rule result in $\forall m : nat, \forall n : nat, \forall p : nat$, $matrix\ m\ n \rightarrow matrix\ n\ p \rightarrow matrix\ m\ p : Set$.

These examples demonstrate the power, the versatility and the expressiveness of the dependent product in the Calculus of Constructions, with direct consequence over the main characteristics of the Coq proof assistant and its use in different application domains.

The Calculus of Constructions is a large and complex subject that needs a lot of effort and experience in order to be adequately mastered. This section introduced some of its main characteristics, without trying to be exahustive. The intention is that it will enough to enable the understanding of the basics of the Coq proof assistant, specially the semantics of its tactics language (part of Vernacular, in Section B.4) and the syntax and semantics of its term specification language (Galina, in Section B.5).

The Calculus of Constructions, the last topic of this chapter, was presented on the ground of other more basic theories reviewed before, and is the formal system that we will use to represent the objects of our theory, the statements that we want to prove about these objects and also to construct the proofs of these statements. The Coq proof assistant is an implementation of the Calculus of Constructions which is present, directly or indirectly, in almost every interaction of the user with the system. We are now, thus, ready to characterize the theory that we want to formalize (in Chapter 4 and Chapter 5). The corresponding formalization – the main result of this work – will then be presented in Chapter 6.

B

The Coq Proof Assistant

The Coq proof assistant is a software tool that implements the Calculus of Inductive Constructions presented in Section A.7. Its popularity and importance is confirmed by various recent large and complex formalization projects in the areas of mathematics and software development (see Chapter 3). The purpose of this appendix is to make an introduction to the two languages of Coq, with examples of their usage, and also to the IDE used in the interaction with the user, with the objective of facilitating the understanding of the present formalization.

The main characteristics of Coq are discussed in Section B.1. In it we present some ideas that are key to the understanding of its dynamic behaviour, such as the notions of direct and indirect proof construction, forward and backward reasoning and interactivity.

The contents of a typical formalization file is examined in Section B.2, where we define the important notions of Environment and Context, introduce some commands used to organize and structure the formalization, as well as some basic commands the create definitions and declarations.

The more practical aspect of the usage of the Coq proof assistant is discussed in Section B.3, with a presentation of the CoqIDE graphical user interface. Although it offers a large set of buttons and menus in order to simplify the user experience and increase his productivity, we will concentrate on the explanation of the three main areas of the interface and on the navigation of the script written by the user.

Now we can discuss ways to textually express reasoning and computation in Coq. For this purpose, Coq uses two different languages with different objectives: the language used to define objects, build proof scripts and interact with the user is called *Vernacular*, and the formal language, used to represent proofs, propositions, terms and types is called *Gallina*. They are introduced, along with examples, respectively in Section B.4 and Section B.5. *Gallina* and *Vernacular* are big and complex languages that require some effort and time in order to be adequately mastered. In these sections we will introduce the main features of them, with special focus on what was more important and more frequently used in this formalization. The discussion is not intended to be exhaustive, but instead have the objective of explaining the general idea behind the representation and interpretation of a Coq formalization.

Next, we give an overview of the way that Coq can be used to reason about programs and proofs in Section B.6. This is accomplished with a complete example of the proof of a lemma, and a step-by-step explanation of the effect of each of the tactics used in the construction of the proof term for this lemma. It includes also a second example for which an inductive proof has been developed. These are small and simple examples, however illustrative of the nature of the formalization activity used in the present project.

The computing capabilities of Coq, which can be explored independently or as an accessory to the reasoning activity, is the subject of Section B.7 at the end of this appendix.

For pedagogical questions we have tried, as much as possible, to make the presentation of Coq independent of the presentation of the Calculus of Constructions, despite their intimate connection. Also, for the same reason, we have tried to describe the two languages of Coq in an independent way. These approaches are in contrast with the published literature, where they are usually intermixed, but we hope it will contribute to a better understanding of the characteristics and role of each, as well as the interaction between the CC, *Vernacular* and *Gallina*.

Comprehensive and detailed information on Coq, along with lots of examples, can be found in the Reference Manual (THE COQ DEVELOPMENT TEAM, 2015), in the books by Bertot and Castéran (BERTOT; CASTÉRAN, 2004), by Benjamin Pierce and others (PIERCE, 2015) and by Adam Chlipala (CHLIPALA, 2015), as well as in the many reports, articles, slides and tutorials available online.

The Coq proof assistant was initially developed at INRIA by G. Huet and T. Coquand in 1984, and the first version was released in 1989. In 1991, with the addition of inductive types, it was renamed Coq. Since then, many other people have contributed to its development, adding increasingly new features and functionalities to the tool. The whole development history is described in the Coq Reference Manual (THE COQ DEVELOPMENT TEAM, 2015). In 2013, Coq received the Programming Languages Software Awards from the ACM for its significant impact in programming language research, implementations and tools (INRIA, 2013). The ACM page for the award (ACM SIGPLAN, 2013) states that:

> As a software system, Coq has been in continuous development for over 20 years, a truly impressive feat of sustained, research-driven engineering. The Coq team continues to develop the system, bringing significant improvements in expressiveness and usability with each new release.
>
> In short, Coq is playing an essential role in our transition to a new era of formal assurance in mathematics, semantics, and program verification.

Coq is nowadays considered one of the most important interactive theorem provers around and is being deployed in significant formalization projects, as discussed in Chapter 3.

B.1 Main Characteristics

Since Coq is a direct implementation of the Calculus of Constructions (Section A.7), it inherits all of its characteristics and has therefore the same power and expressiveness. Since CC is itself an intuitionistic type theory, this permits Coq to extract correct (or certified) code from proofs, which constitutes another of its important features. Besides that, Coq offers support for both reasoning and computing, and can also be used as a functional programming language. The possibility of doing computation during proofs and using reasoning techniques in the development of correct programs is also of high importance.

Gallina is a powerful language, used to state theorems and to write program specifications. Depending on the case (theorems or specifications), it is also the object language in which proofs of the theorems and programs that satisfy the specifications are written. The latter activity is also known as *certified program development*. Proofs and programs can be constructed directly, using Gallina, or can be developed indirectly, interactively using a set of "tactics" that assist the user in the process, with some advantages. In the indirect form of usage:

1. The initial goal is the theorem or specification supplied by the user;
2. The initial context of the proof is usually empty;
3. The application of a "tactic" (a command issued by the user, see Section B.4.1), on either the current goal or one of the hypotheses, substitutes the current goal for zero ou more subgoals;
4. This creates the notion of a stack of subgoals, all of which have to be proved in reverse order.
5. The context changes and may incorporate new hypotheses;
6. The process is repeated for each subgoal, until no subgoal remains;
7. The proof/expression is then constructed, checked and saved by the proof assistant from the sequence of tactics used.

The inference rules of the CC are behind the tactics that are used to construct proofs and expressions in Coq. They map premises to conclusions, and can be used in two different ways in Gallina:

- *Forward reasoning* is the process of moving from premises to conclusions (example: from a proof of a and a proof of b one can prove $a \wedge b$);
- *Backward reasoning* is the process of moving from conclusions to premises (example: to prove $a \wedge b$ one has to prove a and also b);

Coq uses a mixture of *backward* and *forward* reasoning, depending on whether the inference rules are applied to the current goal or to hypotheses in the context. In the first case, the current goal is reduced to its subgoals, if any. In the second, changes are made to the context by adding new hypotheses or changing the existing ones.

Gallina is a dependently typed functional programming language that combines a higher-order logic language and a typed functional language, and is used to represent expressions, proofs, types and propositions in the system. Another language, called *Vernacular*, is used to interactively assist the user in the development of his/her proofs.

Tactics are commands that, when interpreted, apply one or more rules of the CC to the current context or the current goal. The initial goal is the proposition to be proved, which can be a single theorem (or lemma) or the specification of some property of a program. The application is in reverse order, so that the current goal is substituted by one or more (simpler) subgoals and possibly some changes to the context, thus leading to the concept of a proof tree. When no new subgoals are generated, the initial goal is considered valid and a proof, constructed from the application of the rules according to the tactics used, is built.

A type-checking algorithm ensures that proofs are correctly constructed and, consequently, that the corresponding programs satisfy their specifications. In this sense, Coq is suited not only for developing abstract mathematical theories, but also for the development of correct programs and the certification of already developed programs. As discussed in Chapter 3, many important theories and systems have been successfully developed over the recent years with Coq.

Coq uses a constructive logic and comes with a large standard library that defines fundamental data types (natural numbers, integers, booleans, lists etc), implements most operations on them and proves their most important properties (INRIA, 2016). Also, there is a large set of user contributions from the most diverse areas, which can be freely accessed and used (INRIA, 2015). Finally, Coq can be extended with different plug-ins that offer the user alternative (usually higher-level) script languages (e.g. SSReflect, (WHITESIDE; ASPINALL; GROV, 2012)).

Theories can be compiled and stored in object files for efficient reuse in the Coq environment. Also, Coq supports program extraction from terms/proofs into different commercial computer programming languages.

Coq is implemented in the Ocaml functional programming language and supports higher-order logic, dependent types, code extraction, proof automation and proof by reflection, thus being a powerful tool suited for a wide range of applications. Despite its many advantages, some critics consider the foundations of Coq too complicated to be understood and be used by mathematicians. Also, the facts that Coq's logic is constructive (except if explicitly required by the user, one can not use the Law of the Excluded Middle) as opposed to classical logic, and also that equality is intensional, as opposed to extensional (where different reduction strategies do not have to be considered), are considered unrealistic and too distant from the universe of the practicing mathematician (WIEDIJK, 2007).

B.2 Organization

A formal development is normally constituted of a set of text files, identified by their names, each of which groups all the declarations, definitions, lemmas and theorems (among others) that have some main notion (an object, an operation etc) in common. These high-level text files (with the extension ".v") can then be compiled into corresponding low-level ones (with the suffix ".vo"). These low-level files can be imported with the command `Require Import <name>` into the high-level ones, thus making all of its contents available. The same effect can be obtained by using the command `Include <name>`, which acts as a macro expansion by inserting the contents of the specified high-level text file in the exact point where the command was issued. This, however, creates larger files and demands more processing.

The word *Environment* is used to denote the set of all global declarations and definitions in a source file. Local declarations and definitions (with a restricted scope) can also exist, and for that purpose Coq offers the `Section` mechanism. Sections can be nested and all declarations and definitions inside a section have their scope limited to it. The set of declarations and definitions that are placed inside a section is called the *Context*. Thus, the construction of a proof usually takes into account declarations and definitions of the current *Context* in addition to the corresponding *Environment* (that is, global and local declarations and definitions).

A declaration is used to associate a type to an identifier. The identifier must be unique and the association to the type is permanent. Differently from usual imperative programming languages, values can not be assigned to these identifiers. For globally declared identifiers, the keyword `Parameter` should be used:

```
Parameter n: nat.
Parameter f: nat → nat.
```

For locally declared identifiers, the keyword `Parameter` should be substituted by `Variable`. It is recommended, however, that the keywords `Axiom` and `Hypothesis` be used when introducing global and local declarations in the case that they have a proposition as type. In this situation they replace, respectively, the keywords `Parameter` and `Variable` with no change in the semantics of the command.

As an example:

```
Axiom H: A ∧ B → B ∧ A.
```

should be preferred instead of:

```
Parameter H: A ∧ B → B ∧ A.
```

in an enviornment, and

```
Hypothesis H: A ∧ B → B ∧ A.
```

should be preferred instead of:

```
Variable H: A ∧ B → B ∧ A.
```

in a context.

Definitions are used to a associate a term to an identifier, and have the same syntax both for global and local scopes. The identifier is then called a *constant*.

```
Definition one:= S 0.
Definition square:= fun n: nat ⇒ n ∗ n.
```

The types of the identifiers are obtained from the corresponding terms, which must be well-formed in the current environment and context. However, the syntax also allows for the explicit type definition of the constant:

```
Definition one: nat:= S 0.
Definition square: nat → nat:= fun n: nat ⇒ n ∗ n.
```

Lemmas and theorems are global definitions with the particular characteristic that their type is a proposition, the proposition that one wants to prove valid.

A section begins with the command `Section <name>` and ends with the command `End <name>`. Sections can be nested and are also used to simplify the parametrization of definitions and others, by considering the variable declarations that occur in it and their usage inside it.

As an example, consider the definition of `pair` as the cartesian product of two types (in this example, * represents the pair operator, not the multiplication symbol):

```
Definition pair (A B: Type): Type:= (A ∗ B).
```

In this definition, `pair` is explicitly parametrized with types `A` and `B`. However, the same definition could be made simpler with the aid of a section, as in:

```
Section example.
Variables A B: Type.
Definition pair: Type:= (A ∗ B).
End example.
```

In both cases, `pair` represents the function that takes two types and returns the pair type:

```
pair = fun A B : Type ⇒ (A ∗ B)
```

It is usual to start a source (".v") file with the description of all files that will have to be loaded (either ".v", if in source format, or ".vo" if in object format), followed by all the declarations and definitions that will be used in it, then the most simple and independent lemmas and finally the longest and more complex theorems. The order in which of these objects are introduced reflects the dependency among them, since there is no forward referencing. A set of high-level files is considered a valid formalization if all references, including those among different files are correctly resolved and all names are correctly defined.

In the following example, `Require Import` and `Include` are Vernacular commands used, respectively, to load external object and source files:

```
...
Require Import <id>.
...
Include <id>.
...
Parameter <id>:<type>.
...
Definition <id>:=<term>.
...
Lemma <id>: <type>.
Proof. <proof> Qed.
...
Theorem <id>: <type>.
Proof. <proof> Qed.
...
```

Environments and contexts are considered to be well-formed when the identifiers used in declarations and definitions are all different and all terms are well-formed.

As a conclusion, it can be noted that Coq uses a simple and small set of commands to organize the scripts. The mechanisms for splitting the formalization in different files are easy to use, as well as the structuring command, which resembles the block commands found in most imperative programming languges.

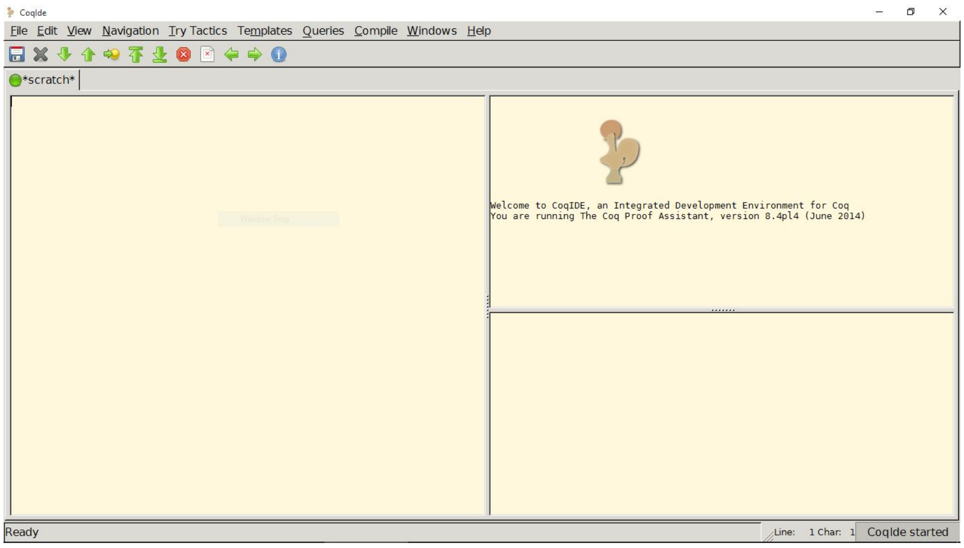
B.3 Graphical Interface

Coq can be used directly, as a text application, or indirectly, via a graphical interface. Graphical interfaces are often more comfortable to use, help to improve the user productivity and offer additional tools, such as menus and shortcuts to most the used commands. Among the most popular graphical interfces for Coq, we find the Proof General Emacs extension and CoqIDE. In both cases, the layout of the user interface consists of three different areas in the same window, as shown in Figure B.1.

The left-hand side of the window is used to place and edit the current script. It is the area where the user writes down his declarations, definitions, statements, programs, proof scripts etc. Its contents are processed line by line and gets locked once it has been accepted by the system (Figure B.2). Errors are informed to the user in the lower right-hand side of the window, and the user is prevented of advancing.

The upper right-hand area is reserved to display the current set of assumptions along with the current goal (the one that lies in the top of the stack). This area is affected by the processing of the contents in the left-hand side. Depending on the tactics or commands used, there might be changes, deletion or creation of new assumptions, as well as changes in the current goal,

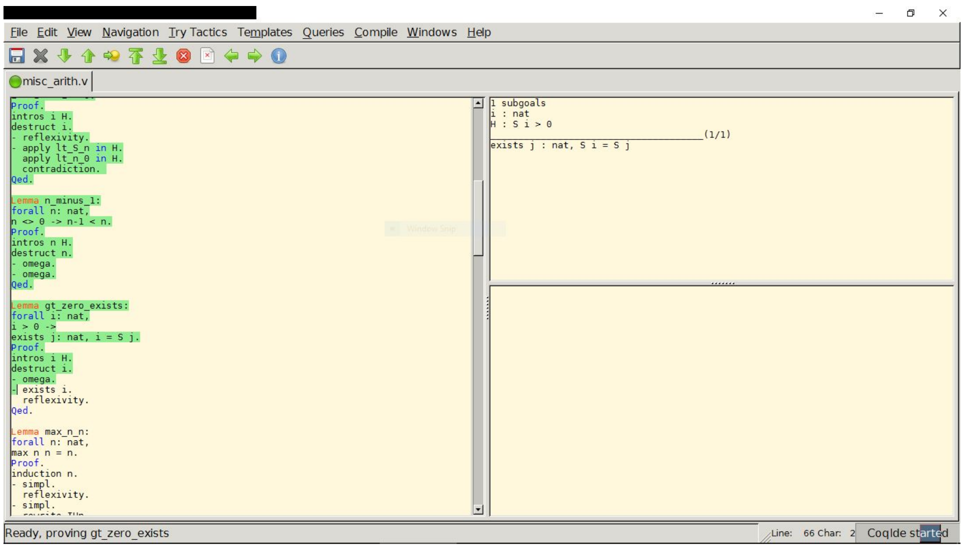
Figure B.1: Layout of the CoqIDE


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch
Ready
Line: 1 Char: 1
CoqIde started

```
Welcome to CoqIDE, an Integrated Development Environment for Coq
You are running The Coq Proof Assistant, version 8.4pl4 (June 2014)
```

Source: the author

Figure B.2: Layout of the CoqIDE during a proof


[redacted]
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
misc_arith.v
Ready, proving gt_zero_exists
Line: 66 Char: 2
CoqIde started

```
Proof.
intros i H.
destruct i.
- reflexivity.
- apply lt_S_n in H.
  apply lt_n_0 in H.
  contradiction.
Qed.

Lemma n_minus_1:
forall n: nat,
n <> 0 -> n-1 < n.
Proof.
intros n H.
destruct n.
- omega.
- omega.
Qed.

Lemma gt_zero_exists:
forall i: nat,
i > 0 ->
exists j: nat, i = S j.
Proof.
intros i H.
destruct i.
- omega.
- exists i.
  reflexivity.
Qed.

Lemma max_n_n:
forall n: nat,
max n n = n.
Proof.
induction n.
- simpl.
  reflexivity.
- simpl.
```

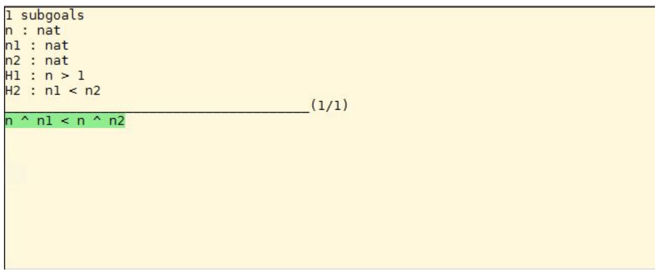
```
1 subgoals
i : nat
H : S i > 0
______________________________________(1/1)
exists j : nat, S i = S j
```

Source: the author

substituting it by a set of new subgoals or simply by eliminating it.

The upper right-hand area is furher detailed in Figure B.3. In the top we find the set of named assumptions available to the current proof. This set might be initially empty or contain axioms that will be taken into account during the proof. The order of the assumptions is not important and can be easily changed with specific commands. The line in the bottom separates the assumptions from the current goal. Initially, the current goal is the original proposition to be proved. A set of tactics is then applied in order to create new assumptions, in case that is necessary, and to prove the current goal. This can be done by generating new assumptions that get closer to the current goal (*forward* reasoning), or by simplifying the current goal in order

Figure B.3: Hypotheses and current goal during a proof



```
1 subgoals
n : nat
n1 : nat
n2 : nat
H1 : n > 1
H2 : n1 < n2
______________________________________(1/1)
n ^ n1 < n ^ n2
```

Source: the author

to approximate it to the available assumptions (*backward* reasoning). In the latter case, the simplification may generate multiple new subgoals, which are kept in a stack and proved in reverse order, one by one, according to the corresponding set of assumptions. A proof is finished when the last subgoal is proved.

The IDE is intuitive and can be easily mastered. It can be used to interactively write scripts, check their correctness and make the necessary corrections in the same environment. It includes shortcuts and buttons that make navigation even easier, and menus that can be used to further improve productivity. More information about CoqIDE can be found in the Coq Reference Manual (THE COQ DEVELOPMENT TEAM, 2015).

B.4 Vernacular

The *Vernacular* language comprises a set of commands that can be issued the by user with different purposes. They can be used to manipulate goals and hypotheses, to define new objects, to structure and organize the source files and to retrieve information, among others. Examples of the first category will be described in Section B.4.1 and of the others in Section B.4.3. The mechanism offered by Vernacular to improve the structure of the proof scripts, which has been extensively used in this formalization, is briefly presented in Section B.4.2. An extensive list of commands and their variants, with detailed semantics, can be found in the Reference Manual (THE COQ DEVELOPMENT TEAM, 2015).

All commands start with a keyword, are followed by zero or more arguments and are usually finished with the dot symbol (·).

B.4.1 Tactics

Commands used to indirectly construct proof terms are called *tactics*. A collection of tactics is related by a set of *tacticals* that establish the execution order for these tactics. The most important tactical is the *sequence tactical*: the next tactic is executed only when the previous finishes its execution.

Tactics implement inference rules of the CC, and serve to ease the construction of proof terms by abstracting details and allowing the user to concentrate on the higher level aspects of his proofs. This way, proof terms do not have to be constructed directly by the user. Instead, they are automatically generated as the result of the sequence of tactics supplied by the user.

Tactics manipulate the hypotheses in the environment/context, the current goal or both, giving rise to new hypotheses and new subgoals. Eventually, a tactics may prove the curent goal and thus finish the proof in case there are no pending subgoals.

Coq has a large set of tactics, with many variants and some times a complex semantics. The most frequently used tactics in this formalization are summarized and briefly described below, in alphabetical order. Most of them have arguments and variants, which are not described here. Some refer to the corresponding inference rule(s) of the Natural Deduction that they implement (see Section A.5).

- `apply`: used to prove the current goal by matching it to the conclusion of an implication. In this case, the premise of the implication becomes the new goal to be proved. Can also be used to generate a new hypothesis by matching the premise of the implication to some other hypothesis already in the context. Corresponds to the application of the elimination rule for the universal quantifier ($\forall E$) from Natural Deduction.

- `assert`: used to introduce a new named hypothesis in the context; the newly introduced proposition is put on top of the current stack of subgoals and must be proved before the others.

- `change`: used to substitute a subterm of another term by another subterm, if this new subterm is well-formed and the subterms are mutually convertible. Can be applied to the current goal or any hypothesis.

- `congruence`: used when the current subgoal is an equality. Tries to prove the current goal by finding in the context a set of equalities from which a proof of the current subgoal can be inferred.

- `contradiction`: used when the context contains a proof of `False` or any term that can be reduced to it. The current subgoal is automatically proved. Corresponds to the application of the elimination rule for False ($\perp E$) from Natural Deduction.

- `destruct`: takes an inductive type as argument and substitutes the current goal by as many new subgoals as there are constructors in the type definition, one for each constructor. Can also be applied to hypotheses in the context. Corresponds, among others, to the application of the elimination rule for the conjunction ($\wedge E_1$ and $\wedge E_2$), the elimination rule for the existential quantifier ($\exists E$) and the elimination rule for the disjunction ($\vee E$) from Natural Deduction.

- `discriminate`: used to prove the current goal in the case that the context contains an equality of structurally different terms constructed from the same inductive definition.
- `exact`: proves the current subgoal by naming the corresponding proof term in the context.
- `exists`: used to supply a value to a variable of an existential quantifier in the current subgoal. Corresponds to the application of the introduction rule for the existential quantifier ($\exists I$) from Natural Deduction.
- `generalize`: used to generalize a subterm of the current subgoal by creating a new bound variable and introducing a universal quantification.
- `induction`: applies an induction principle associated to an inductive type or inductive proposition and generates new subgoals. Used to construct proofs by induction (see Section B.6).
- `intros`: used mainly when the current goal is a dependent or a non-dependent product; it extracts bound variables and premises and transfers them to the local context as named hypotheses, thus simplifying the current goal and populating the context. Corresponds to the application of the introduction rule for the universal quantifier ($\forall I$) from Natural Deduction.
- `inversion`: is applied to an inductive proposition in the context, and is used to generate new subgoals, one for each constructor of the corresponding inductive definition. New hypotheses are also generated in order to fully characterize the use of each constructor.
- `left`: used to prove the left-hand side of the current subgoal, if it is a disjuction. Corresponds to the application of the introduction rule for the disjunction ($\vee I_1$) from Natural Deduction.
- `omega`: proves the current subgoal in case it is (i) quantifier free, (ii) contains only common logical connectives, (iii) uses equalities and inequalities with the base type `nat`, and (iv) the facts available in the context provide the necessary conditions for the proof.
- `reflexivity`: proves the current subgoal if it is an equality with the same term on both sides.
- `remember`: extracts a subterm of the current subgoal or hypotheses and associates it to a new name in the current context.

- `replace`: used to substitute a subterm of a term by another subterm, if this new subterm is well-formed and the subterms are mutually convertible. Can be applied to the current goal or any hypothesis and, differently from `change`, a new subgoal is generated and the equalitiy of the subterms must be proved before the substitution occurs.

- `rewrite`: substitutes one subterm by another, based on an equality of these subterms already present in the context.

- `right`: used to prove the right-hand side of the current subgoal, if it is a disjuction. Corresponds to the application of the introduction rule for the disjunction ($\vee I_2$) from Natural Deduction.

- `simpl`: used to simplify the current subgoal or any hypotheses.

- `specialize`: used to provide a proof of a premise to a proof of an implication already in the context. This latter proof is then transformed in a proof of its conclusion only.

- `split`: used when the current subgoal is a conjunction; creates two new subgoals, one for each component of the conjunction. Corresponds to the application of the introduction rule for the conjunction ($\wedge I$) from Natural Deduction.

- `symmetry`: changes the order of the subterms in an equality in the current subgoal.

- `unfold`: substitutes a name for its definition in the curent subgoal or hypotheses.

B.4.2 Structure

As a proof script grows in size and complexity, it is convenient to use syntactical conventions that approximate its static structure to its dynamic behaviour. This allows the user to have a better understanding of the proof structure, improving the readability and maintenability of the proof script.

A proof script is normally a sequence of tactics that are applied, one at a time, to a stack of goals. Some tactics might just change the goal on top of the stack, remove one or more subgoals from the top of the stack or still add new goals to the top of the stack. The proof is finished when the stack is empty.

Thus, the proof script (implicitly) creates a proof tree, where one goal might be split in one or more new goals. In order to keep track of which tactics are being used to prove what goals, Vernacular provides *bullets* that can be used to focus on the current goal, ignoring the rest. Bullets in the same level (represented by the same symbol) are used to relate different goals that have originated from the same goal. The bullets used by Coq are $-$, $+$ and $\star$, and have no special meaning except as a structuring mechanism. They can be used in any order, however this order must be respected during the proof.

As an example, consider the following proof script (for an introduction to the semantics of the proof script, please refer to Section B.6):

```
Lemma example:
∀ a b c d: Prop,
(a → (a → b) → c → (c → d) → (a ∧ b) ∧ (c ∧ d)).
Proof.
intros a b c d H1 H2 H3 H4.
split.
split.
exact H1.
apply H2.
exact H1.
split.
exact H3.
apply H4.
exact H3.
```

The script above can be rewritten, using bullets, in such a way that is becomes clear that each `split` generates two new subgoals, and how each of these is proved:

```
Lemma example:
∀ a b c d: Prop,
(a → (a → b) → c → (c → d) → (a ∧ b) ∧ (c ∧ d)).
Proof.
intros a b c d H1 H2 H3 H4.
split.
- split.
  + exact H1.
  + apply H2.
    exact H1.
- split.
  + exact H3.
  + apply H4.
    exact H3.
Qed.
```

Indentation is free and optional, and of course can be used in order to further improve the readability of the script.

B.4.3 Commands

The following are the main commands used to define new objects in the formalization:

- `Definition`: used to associate an identifier to a term (including non recursive functions). Type information is optional, since it can be retrieved from the term if it is well-formed.

  ```
  Definition one:= S 0.
  Definition one: nat:= S 0.
  Definition square:= fun x: nat ⇒ x * x.
  ```

- `Variable`: used to introduce a new identifier with the corresponding type in the context (inside a section). Objects that refer to such an identifer inside the section are parametrized with respect to the identifier when used outside the section.

  ```
  Variable x: Prop.
  ```

- `Parameter`: used to introduce a new identifier with the corresponding type in the environment (outside a section).

  ```
  Parameter y: Prop → Prop.
  ```

- `Notation`: an abbreviation for a more complex expression; notations exist only inside sections and disappear when they are closed.

  ```
  Notation natlist2:= (list (list nat)).
  ```

- `Record`: a collection of values stored in different fields of the same structure with possibly different types and accessed by different names. May include predicates that control the relationship between the values of the fields.

  ```
  Record coordinates: Set := {
         p: nat;
         q: nat;
         r: nat }.
  Definition c: coordinates:= {| p:= 1; q:= 2; r:= 3 |}.
  ```

- `Inductive`: used to initiate the definition of an inductive type or an inductively defined proposition. Consists of a set of constructors that inductively define all the inhabitants of the type (see Section B.5.2 and Section B.5.3).

- `Fixpoint`: used for the declaration of recursive functions; contains a list of parameters and must terminate after a finite number of calls. The termination should be verified by ensuring that sucessive calls are made with “smaller” values of some parameter. In the following example, `fact` is called with successively smaller values of a natural number, and this guarantees that the function always terminates despite the value of the argument in the initial call.

```
Fixpoint fact (n: nat): nat:=
 match n with
 | O ⇒ 1
 | S m ⇒ n * fact m
 end.
```

- `Lemma`: used to introduce a new lemma statement and its proof; the lemma should have a name and be associated to a well-formed proposition. The proof starts after the keyword `Proof` and finishes before the keyword `Qed`.

```
Lemma example:
∀ a b c: Prop,
(a → (b → c)) → (b → (a → c)).
Proof.
intros a b c H1 H2 H3.
apply H1.
– exact H3.
– exact H2.
Qed.
```

- `Theorem`: same as `Lemma`; the selection of one or another reflects the importance of the proposition in the context of the formalization and is a purely personal choice.

```
Theorem example:
∀ a b c: Prop,
(a → (b → c)) → (b → (a → c)).
```

- `Proof`: starts the construction of the proof term for the corresponding proposition; it should be followed by a sequence of tactics that instruct the system on how to build the proof term.

- `Qed`: finishes the current proof in case there are no remaining subgoals; also, it checks that the constructed proof term is well-formed and stores it for future references.

The next set of commands is used to structure and organize the formalization inside a source file and among different files:

- `Require Import`: imports all definitions of the argument file (in its compiled version), thus making them available to the file where the command was issued.

```
Require Import Omega.
```

- `Section`: initiates a new block in the source file, where local definitions can be created and stored. The section is identified by the argument name, and must be closed with the command `End`.

  ```
  Section Simplification.
  ```

- `End`: closes the current section. The name used must be the same used in the corresponding `Section` command.

  ```
  End Simplification.
  ```

Finally, we present a set of commands that is used to retrieve information from objects of the formalization:

- `Check`: returns the type or sort of the argument term.

  ```
  Check (fun x: nat ⇒ x).
  fun x : nat ⇒ x : nat → nat
  ```

- `Print`: returns the term associated to the argument.

  ```
  Print list.
  Inductive list (A : Type) : Type :=
  nil : list A
  | cons : A → list A → list A
  ```

B.5 Gallina

Gallina is the formal language used by Coq to represent types, propositions, terms and proofs. It is a direct implementation of the Calculus of Inductive Constructions introduced in Section A.7. With it, one can create definitions (constants, functions, types, predicates etc) and prove lemmas and theorems about them, thus defining whole theories. It can also be used to construct programs that satisfy specifications, or simply to check that a program implements a specification. Gallina can be manipulated directly, when the user describes his objects, or indirectly, when the manipulation is done via the tactics language of Vernacular.

B.5.1 Types and Propositions

Every term in Coq, no matter if it is an expression or a type, has a type. Thus, proofs, programs, propositions and even types have types. The type of a type is called a *sort*.

The notation $a : A$ is used to express the fact that term a has type A. It can also be interpreted as the fact that a is a proof of proposition A. The base of the type system of Coq are the sorts `Set`, `Prop` and `Type`, as in `A:Type`.

The type inference rule *Var* (presented in Section A.7.3 and Section A.7.4) is used to introduce new identifiers in the context, along with their types. `Type` represents a family of sorts, namely all sorts $Type(i), i \geq 1$. Sorts `Prop` and `Set` are of type `Type(1)`, represented by Coq as simply `Type`:

```
Check Set.
Set: Type.

Check Prop.
Prop: Type.
```

Also, the type of `Type` is `Type`:

```
Check Type.
Type: Type.
```

In this case, however, the `Type` to the left of `:` is an abbreviation for `Type(i)`, for some `i`, and the `Type` to the right of `:` is an abbreviation for `Type(i+1)`, for the same value of `i`.

New types can be constructed through the use of different operators. The first one is the *arrow type*, used to represent simultaneously the type of functions and also the logic connective of implication.

As an example, lets consider:

```
A: Set
B: Set
f: A → B
a: A
```

Then, the term `(f a)`, which represents the application of term `f` to term `a`, is of type `B` and corresponds to the application of the type inference rule *App*.

Now, suppose that:

```
A: Prop
B: Prop
f: A → B
a: A
```

In this case, *a* is a proof of proposition *A* and `(f a)`, which is also an application of term `f` to term `a`, is a proof of proposition *B*.

In both cases, `A -> B` is the type of functions that map from *A* to *B*, and its interpretation depends on the sorts of *A* and *B*. The rule *Lam* is used to specify the type of such a function:

$$\underbrace{\texttt{fun a: A => b}}_{program} : \underbrace{\texttt{A -> B}}_{specification}$$

$$\underbrace{\texttt{fun a: A => b}}_{proof}:\underbrace{\texttt{A -> B}}_{proposition}$$

The keyword `fun` and the symbol `=>` represent, respectively, the symbols λ and $\cdot$ of the lambda-calculus. They are followed by the list of parameters of the function and the term that yields the result. Thus, `fun a:A=>b` corresponds to the lambda-term $\lambda a^A.b$.

Propositions are formulas that represent logical statements and can be parametrized with a number or arguments. A proposition is *well-formed* is it is formed in accordance to the typing rules of the underlying theory. Besides the implication connective, introduced in Section A.7.4, the inductively defined connectives presented in Section B.5.3 (conjunction, disjunction, negation and existential quantification), along with the primitive universal quantification, can also be used to form more complex propositions and thus richer types.

A proposition is an element of the sort `Prop`. A term whose type is a proposition is a proof of the proposition. In other words, a proposition is the type of all proofs of that proposition.

As an example, consider:

```
Parameter A B: Prop.
Definition and_comm: Prop:= A ∧ B → B ∧ A.
```

In this case, the constant `and_comm` is the name used to refer to the proposition `A/\B -> B/\A`. Suppose now that `H: and_comm`. Then, `H` is a proof of proposition `and_comm`. Since this is essentially an implication, the proof is a function that maps a proof of `A/\B` to a proof of `B/\A`:

```
fun H : A ∧ B ⇒
conj
  (( fun H0 : B ⇒ H0) match H with
                       | conj _ x0 ⇒ x0
                       end)
  (( fun H0 : A ⇒ H0) match H with
                       | conj x _ ⇒ x
                       end)
```

Note that the function `conj` (which represents a constructor of the inductively defined proposition `and`) is used in the construction of the proof. The `match` command (see Section B.5.5) is used to retrieve, respectively, the proof of `B` and the proof of `A` before applying the `conj` constructor to return the proof of `B/\A`. Naturally, the type of this function is `A/\B -> B/\A`.

B.5.2 Inductively Defined Types

New types can be defined inductively by providing a finite number of constructors (fresh identifiers) that can be used to create a finite or an infinite number of inhabitants for the type. A constructor can be a single inhabitant of the type, or a function that generates new inhabitants of the type according to some arguments.

As an example of an inductive definition that has a finite number of inhabitants, lets consider the bool type:

```
Inductive boolean: Type:=
 | false: boolean
 | true: boolean.

Variable x: boolean.
Definition f: boolean:= false.
```

In this case, false and true are two constructors that have no arguments and define the only two inhabitants of this type, respectively false and true. Then we can define a variable x of type bool and also a constant f of type bool and value false. This inductive definition represents the type bool with values false and true.

The next example is the inductive definition of a natural number (nat). This type is essentially different from bool, as it has an infinite number of inhabitants. In order to achieve this, we will still use two constuctors, however the second constructor will be defined as a function that takes an inhabitant of type nat and returns another inhabitant of the same type. Thus, successive applications of this function will generate all inhabitants of the type, in a direct analogy to the unary representation of natural numbers.

```
Inductive nat: Type:=
 | O: nat
 | S: nat → nat.

Variable y: nat.
Definition zero: nat:= O.
Definition one: nat:= S zero.
Definition two: nat:= S one.
```

Observe that O is the base element of the type and represents the number 0 (zero). Constructor S (*successor*) is a function that takes a natural number as argument and generates a new natural number. This way, variable y can be declared with type nat, the constant zero can be defined directly, the constant one can be defined as the successor of zero and two as the successor of one. This inductive definition represents the type nat with values { 0, 1, 2, 3, ...} respectively denoted by O, S O, S S O, S S S O etc.

Differently from the previous examples, a list is a type that is parametrized by another type (in this case, A). This means that the type list can be used to build inhabitants that are lists of inhabitants taken from another type, and this characterises an example of a polymorphic type:

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A.
```

Constructor `nil` represents the empty list. Constructor `cons` takes two arguments, respectively an inhabitant of type `A` and a list of inhabitants of type `A`), and returns a list of inhabitants of type `A`. This represents the inclusion of a new element in an existing list.

For example, the following definitions create a list of naturals (`list_of_nat`) and represent some of its elements (respectively the empty list, the list of the single element 1 and the list with elements 2 and 3):

```
Definition list_of_nat: Type:= list nat.

Definition l_empty: list_of_nat:= nil nat.
Definition l_1: list_of_nat:= cons nat (S O) l_empty.
Definition l_2_3: list_of_nat:=
cons nat (S (S O)) (cons nat (S (S (S O))) l_empty).
```

Other examples of inductive types that have been extensively used in this formalization are the polymorphic types such as the cartesian product, the disjoint sum and the option. They are polymorphic because the type being defined depends on another type (or types), as shown in the previous list example.

- Cartesian product:

  ```
  Inductive prod (A B: Type): Type:=
  | pair: A → B → A * B.
  ```

- Disjoint sum:

  ```
  Inductive sum (A B: Type): Type:=
  | inl: A → A + B
  | inr: B → A + B.
  ```

- Option:

  ```
  Inductive option (A: Type): Type:=
  | Some: A → option A
  | None: option A.
  ```

Symbol $\star$ is used as a notation for `prod` (that is, `A*B` is the same as `prod A B`) and, similarly, symbol + is a notation for `sum A B`. The `option` type is used to simulate partial functions, since Coq deals only with total functions. In this case, constructors `Some` and `None` are used to represent partial mappings.

B.5.3 Inductively Defined Propositions

An inductively defined proposition is a family of propositions that is indexed by some parameters and conveniently defined in an inductive way. A parametrized proposition is also called a *predicate*, and corresponds to the notion of a predicate in the programming language Prolog.

As an example, the following definition represents a famility of propositions that relates two natural numbers i and j in such that j occupies the ith position in a Fibonacci series $(0,1,1,2,3,5,...)$:

```
Inductive fibonacci: nat → nat → Prop:=
| fib_0:
    fibonacci 1 0
| fib_1:
    fibonacci 2 1
| fib_sum:
    ∀ i a b: nat,
    fibonacci i a →
    fibonacci (S i) b →
    fibonacci (S (S i)) (a + b).
```

According to this definition, `fibonacci 1 0`, `fibonacci 2 1` are valid propositions, since 0 and 1 are, respectively, the first and second elements of the series. This fact results directly from constructors `fib_0` and `fib_1`, since they have no dependencies. Constructor `fib_sum`, on the other hand, states that if `fibonacci i a` and `fibonacci (S i) b` are valid propositions, so is `fibonacci (S (S i)) (a + b)`. From this constructor we can then prove, for example, that `fibonacci 3 1`, `fibonacci 4 2` and `fibonacci 5 3` are also valid propositions. Thus, the inductive definition above represents an infinite family of propositions `fibonacci x y` relating sequence numbers to values in a Fibonacci series, such that `x` is the index and `y` is the corresponding value.

Except for the implication, all other logical connectives used in Coq are inductively defined propositions that depend only on the primitive notion of the the universal quantifier implemented in the core system. These connectives are used to build more complex propositions and specifications.

Conjunction is represented by the inductive definition `and` with the single constructor `conj` (∧ is a notation for `and`, thus `and a b` is the same as `a ∧ b`). This definition is an inductive implementation of the introduction rule for the conjunction (∧ I) from Natural Deduction (see Section A.5).

```
Inductive and (A B: Prop): Prop :=
| conj: A → B → A ∧ B
```

Disjunction is represented by the inductive definition `or` with constructors `or_introl` and `or_intror` (∨ is a notation for `or`, thus `or a b` is the same as `a ∨ b`). The two constructors relate directly to the two introduction rules for the disjunction ($\vee I_1$ and $\vee I_2$) from Natural Deduction.

```
Inductive or (A B: Prop): Prop :=
| or_introl : A → A ∨ B
| or_intror : B → A ∨ B
```

Negation, represented by symbol ~, is an abbreviation for implication to false:

```
~ A ≡ A -> false
```

Existential quantifier is represented by the inductive definition `ex` with constructor `ex_intro` (∃ is a notation for `exists`, thus `exists x,Px` is the same as `∃ x,Px`). It relates to the introduction rule for the existential quantifier ($\exists\ I$) from Natural Deduction, however using as hypothesis a universal quantification for the same proposition.

```
Inductive ex (A: Type) (P: A → Prop): Prop:=
| ex_intro: ∀ x: A, P x → ∃ x, Px
```

Finally, even `True` and `False` are not primitive terms, they are inductively defined as:

```
Inductive True: Prop:=
| I : True

Inductive False: Prop:=
```

Observe that `True` has only one constructor (`I`), while `False` has no constructors at all.

B.5.4 Programs and Proofs

Programs and proofs also share the same structure, as happens with types and propositions. Their definition is obtained through the use of the rules *Lam* and *App*, with some extensions such as the one used for case analysis (see Section B.5.5). The syntax used by Gallina to represent programs and proofs is presented in the next example.

Consider the following lemma:

```
Lemma example:
∀ a b c: Prop,
(a → (b → c)) → (b → (a → c)).
```

The proof of this lemma (or the program that complies to this specification) is the term:

```
fun (a b c : Prop) (H1 : a → b → c) (H2 : b) (H3 : a) ⇒ H1 H3 H2
: ∀ a b c : Prop, (a → b → c) → b → a → c
```

which can be understood, according to the Curry-Howard Isomorphism, as either:

- A term that takes as arguments a function $H1$ of type $a \to b \to c$, a value $H2$ of type b and a value $H3$ of type a, and maps to the application $(H1(H3H2))$ which is a value of type c, or

- A term that takes as arguments a proof $H1$ of the implication $a \Rightarrow b \Rightarrow c$, a proof $H2$ of b and a proof $H3$ of a, and maps to the application $(H1(H3H2))$ which is a proof of c.

The syntax used by Gallina to represent abstractions is as follows: the keyword `fun` is used to start a new abstraction declaration. The items between parentheses are its named parameters, along with the corresponding types. The symbol => introduces the body of the abstraction, that is, the term that the abstraction maps to, in this case the application `(H1 H3) H2` or simply `H1 H3 H2` since application is left associative. The type of this abtraction (a dependent product) comes in the second line, after the : symbol.

For recursive function definitions, the keyword `fix`, followed by the name of the function, should be used in the place of `fun`. The next example corresponds to a Gallina program that computes the factorial of a number:

```
fix fact (n: nat): nat:=
match n with
| 0 ⇒ 1
| S m ⇒ n ∗ fact m
end
: nat → nat
```

This program is constructed by Coq automatically from the corresponding definition in Vernacular, using the keyword `Fixpoint`.

Functions in Coq represent the notion of an algorithm, and not the mathematical concept of an abstract mapping between sets, represented by a subset of the cartesian product of these sets. This corresponds, respectively, to the difference between the *intensional* and *extensional* definitions of equality, of which the first is adopted by Coq. All functions are total, and partial functions can only be simulated using the `option` type constructor. Finally, all functions must provably terminate after a finite number of interactions.

In summary, Gallina is a functional language and its functions are extensions of the Typed Lambda Calculus.

B.5.5 Case Analysis

Case analysis can be used to inspect the structure of an inductive term, and considers all the constructors that could have been used for this term. For each constructor, a different branch is executed and yields the value of a different term. The keyword `match` is used to introduce the term that will be inspected.

As an example, consider the following function definition:

```
Definition negb (b: bool):=
match b with
| false ⇒ true
| true ⇒ false
end.
```

Function `negb` maps `b` of type `bool` to the negation of its value. This is done by first checking the current value of `b`. If it is `false`, then `negb` evaluates to `true`. If it is `true`, then `negb` evaluates to `false`.

The syntax of a `match` mechanism is as follows: the term to be inspected (which must be of an inductive type) comes between `match` and `with`. Every possible form of the term must be considered in a sequence that starts with the symbol `|`. To the left of the symbol => we find the particular constructor selected, with the corresponding arguments, and to the right we find the term associated. The keywork `end` must be used after the final branch.

A slightly more complex example uses type `nat`:

```
Definition succ (n: nat):=
match n with
| O ⇒ O
| S n ⇒ S (S n)
end.
```

Function `succ` has a parameter `n` of type `nat`, and evaluates to `O` (zero) if `n` is zero, otherwise `n+1` (or `S(S(n))`) if `n` is greater than zero (that is, if `n` is the successor of some value, as indicated by `S(n)`).

Case analysis is also an important proof technique. The tactic `destruct` (and its simpler version `case`) can be used to do case analysis of an inductively defined term in a proof. For example, in the lemma:

```
Lemma true_or_false:
∀ b: bool, b=true ∨ b=false.
Proof.
intros b.
destruct b.
– left. reflexivity.
– right. reflexivity.
Qed.
```

the tactic `destruct b` substitutes the current goal `b = true \/ b = false` by the two subgoals `true = true \/ true = false` and `false = true \/ false =`

`false`, each of which corresponds to a particular value (constructor) of `b`, so that they can then be easily proved.

For inductively defined propositions, the `inversion` tactic is used in order to generate and examine all the valid possibilities that could result in the term being inspected, and corresponds to a special kind of case analysis that is very useful in the construction of proof scripts in the presence of predicates.

B.5.6 Induction Principles

Every inductive definition (type or proposition) has associated an induction principle (composed by a term and its type) that can be used to prove propositions that depend on these definitions. Induction principles are constructed automatically by Coq and become available to the user as soon as the inductive definition is processed.

As an example, the type of the induction principle for the `bool` type is:

```
∀ P: bool → Prop, P true → P false → ∀ b: bool, P b
```

This induction principle states that, for any property that maps a boolean value to a proposition (represented by `P` above), the proofs that this proposition holds for `true` and `false` are necessary and sufficient conditions for the property to hold for any boolean value. Since `bool` is a finite type, this is not a surprise. However, infinite types also benefit from the use of induction principles, and indeed this is their main application.

Type `nat`, for example, consists of an infinite set of values. If `P` is now a property that maps from `nat` to `Prop`, the proof that this property holds for all elements of `nat` can be obtained by using the corresponding induction principle for `nat`:

```
∀ P : nat → Prop,
P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

In this case, the induction principle requires (i) a proof that `P` holds for the base element of the type (`0`) and also (ii) a proof that if `P` holds for some `n`, then `P` also holds for the sucessor of `n`. This clearly proves that `P` holds for all elements of `nat`.

Proofs by induction over types and propositions were extensively used in the formalization. In most situations, the default induction principle was used, however in a few cases a different and specialized induction principle had to be defined. This is also the case when mutual inductive definitions were used.

B.6 Reasoning

Let us now see, step by step, how a proof term can be constructed using the basic tactics of Vernacular. Consider the following script, which is used to build a proof of a simple statement from the Propositional Logic:

```
Lemma example:
∀ a b c: Prop,
(a → (b → c)) → (b → (a → (b ∧ c))).
Proof.
intros a b c H1 H2 H3.
split.
– exact H2.
– apply H1.
  + exact H3.
  + exact H2.
Qed.
```

The syntax of this example is straightforward: the keyword `Lemma` introduces the name of the proposition to be proved, `Proof` indicates the beginning of the proof script and `Qed` indicates the end of the proof. The name of the proposition to be proved is `example`, and `forall a b c:Prop, (a->(b->c))->(b->(a->(b/\c)))` is the proposition to be proved. When this example is placed in the editing area of CoqIDE, we have the result shown in (Figure B.4).

Observe, also in Figure B.4, that a portion of the script is marked in green, while the rest is not. This is because the first three lines have already been processed by Coq, and the result is that this area is now locked (that is, can not be changed by the user) and, in the right side, the proposition to be proved appears as the initial goal.

To move further (that is, to have Coq process the next line of the script), it is sufficient to click in the *down-arrow* button (the third from left to right) or use a configurable shortcut. To go back (that is, to undo the processing of the last line marked in green) one should click in the *up-arrow* button or use the corresponding shortcut.

The processing of tactic `intros a b c H1 H2 H3` removes premises from the goal and puts them in the context in the form of named hypotheses. It uses the $\forall I$ rule for backward reasoning: a proof of the implication will be constructed (using the $\forall I$ rule) if a proof term can be obtained for the conclusion, in the context of the premises of the implication. In this case, the names selected for the variables of the universal quantification (`a b c`) were the same used in the goal. The other three names (`H1 H2 H3`) were freely selected by the user (Figure B.5).

The `split` tactic is then used to create two new subgoals from the current goal that is a conjunction (Figure B.6). Observe that goal `b/\c` has been substitued by subgoals `b` and `c`, of which `b` is the first one to be considered. This corresponds to the application of rule $\wedge I$.

The subgoal `b` is easily proved by tactic `exact H2`, as `H2` is a proof of `b` already available in the context (Figure B.7 and Figure B.8). This concludes the proof of the first subgoal, and since the goal stack is not empty, the second subgoal shows up with the corresponding context.

The second subgoal `c` requires the application of `H1`, which in turn creates two new subgoals: one is the proof of `a` and the other is the proof of `b` (Figure B.9 and Figure B.10). This corresponds to the application of the $\forall E$ rule, which in turn demands proofs of `a` and `b`.

These two proofs can be found in the context, under the names `H3` and `H2` respectively (Figure B.11, Figure B.12, Figure B.13 and Figure B.14), and solving the subgoals can be done easily by invoking twice the tactic `exact` with the corresponding arguments.

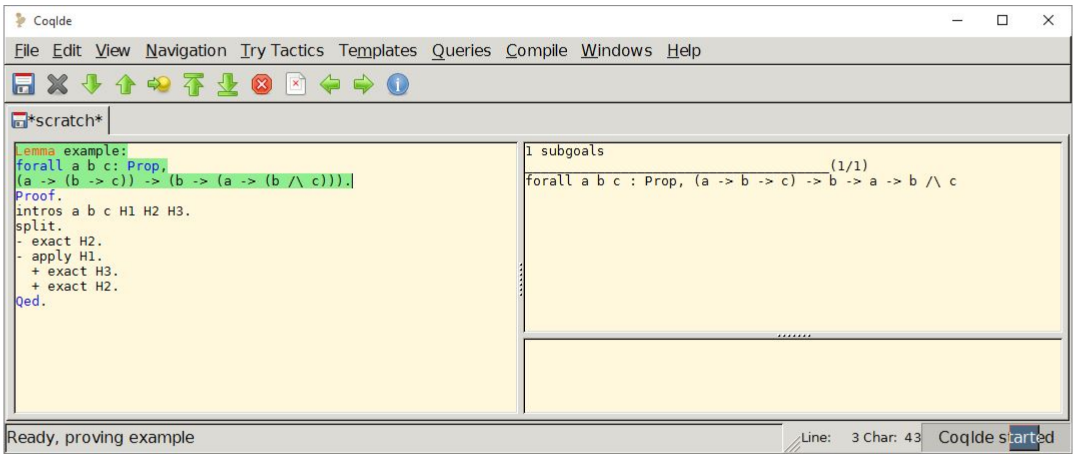
After this, the goal stack becomes empty and the proof reaches an end. The `Qed` tactic checks the proof script again, builds the proof term according to the sequence of tactics used, and stores it for future use (Figure B.15).

The proof term that was built can then be retrieved with the command `Print example`, which yields:

```
example =
fun (a b c : Prop) (H1 : a → b → c) (H2 : b) (H3 : a) ⇒ conj H2 (H1 H3 H2)
: ∀ a b c : Prop, (a → b → c) → b → a → b ∧ c
```

The proof term (the second line above) is a function with six parameters (of types respectively `Prop`, `Prop`, `Prop`, `a->b ->c`, `b` and `a`) that maps to the expression `conj H2 (H1 H3 H2)`, where `conj` is the constructor of the inductively defined proposition `and`, which corresponds to the logical connective $\wedge$. Observe that the proof term contains instances of the $\forall I$ and $\forall E$ rules, and also that the type of this function (in the third line) is the proposition that was proved.

Figure B.4: Start of the proof


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

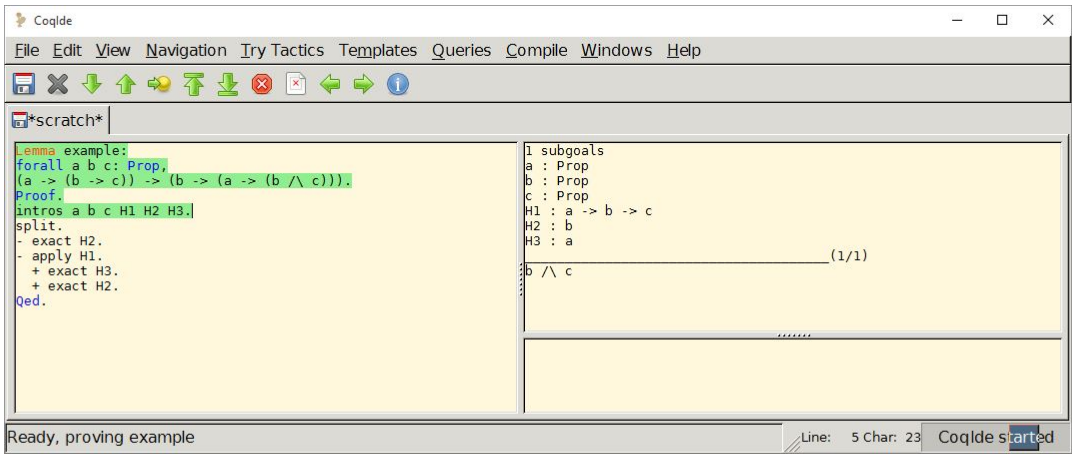
```
1 subgoals
______________________________________(1/1)
forall a b c : Prop, (a -> b -> c) -> b -> a -> b /\ c
```

```
Ready, proving example          Line: 3 Char: 43   CoqIde started
```

Source: the author

Proof by induction is a common proof technique in mathematics and was extensively used in this formalization. It requires the demonstration that some property, expressed as a proposition, holds for all elements of an inductively defined set (in this case, an inductively defined type or proposition). For this, Coq offers the high-level tactic `induction`. When invoked, this tactic identifies the property that must be proved in the current goal, applies the

Figure B.5: After `intros`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

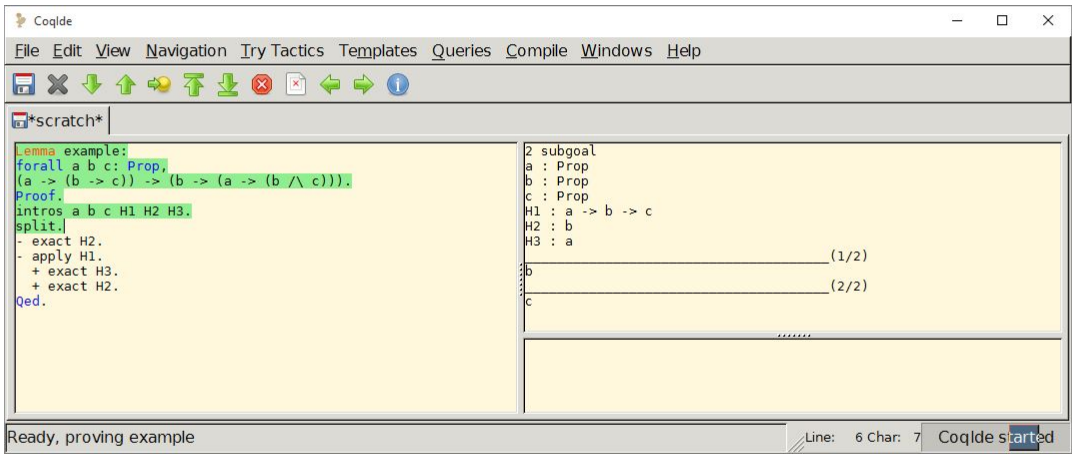
```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
1 subgoals
a : Prop
b : Prop
c : Prop
H1 : a -> b -> c
H2 : b
H3 : a
______________________________________(1/1)
b /\ c
```

```
Ready, proving example        Line:  5 Char:  23   CoqIde started
```

Source: the author

Figure B.6: After `split`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
2 subgoal
a : Prop
b : Prop
c : Prop
H1 : a -> b -> c
H2 : b
H3 : a
______________________________________(1/2)
b
______________________________________(2/2)
c
```

```
Ready, proving example        Line:  6 Char:  7   CoqIde started
```

Source: the author

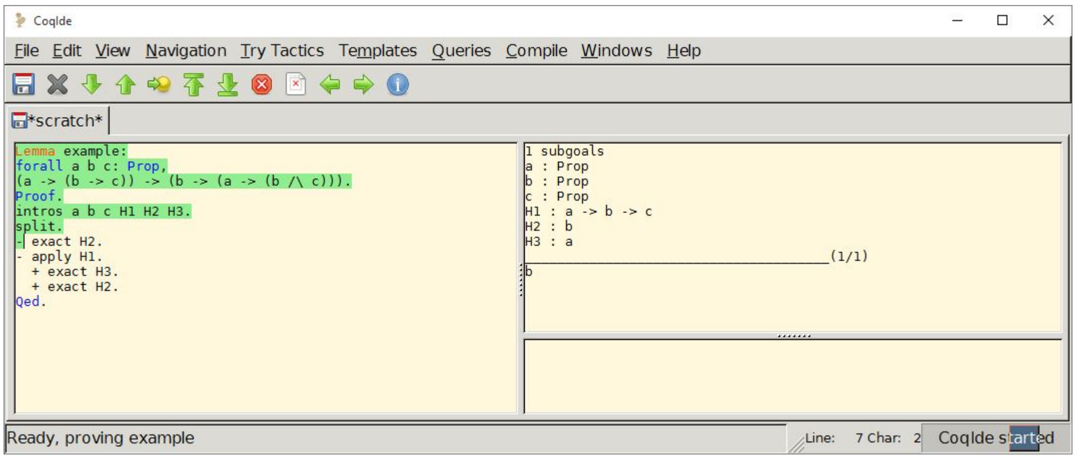
induction principle associated to corresponding type or proposition, creates one new subgoal for each constructor of the definition and, for each of these, introduces the new hypotheses in the context and requires that the user builds the proof for each subgoal.

In what follows, the symbol >= is a notation for the predicate `ge`, which in turn is defined in terms of the inductively defined predicate `le` with constructors `le_n` and `le_S`. The symbol <= is a notation for the predicate `le`:

```
Definition ge (n m: nat):= m ≤ n.
Inductive le (n : nat): nat → Prop :=
| le_n : n ≤ n
| le_S : ∀ m : nat, n ≤ m → n ≤ S m
```

The next example illustrates the use of the `induction` tactic in the proof of the property $\forall x, x * x \geq x$ over naturals (some simple lemmas not presented before have been used).

Figure B.7: Before `exact H2`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

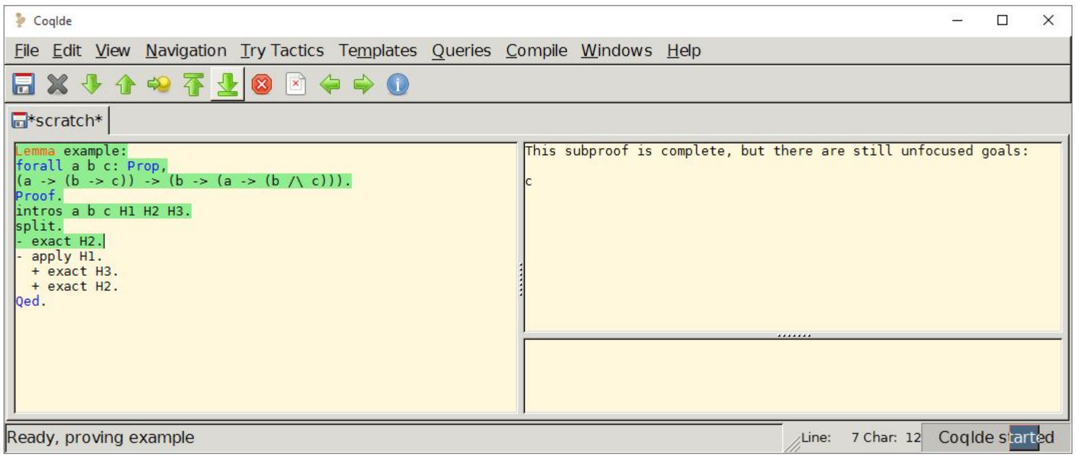
```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
1 subgoals
a : Prop
b : Prop
c : Prop
H1 : a -> b -> c
H2 : b
H3 : a
______________________________________(1/1)
b
```

```
Ready, proving example          Line:  7 Char:  2   CoqIde started
```

Source: the author

Figure B.8: After `exact H2`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
This subproof is complete, but there are still unfocused goals:

c
```

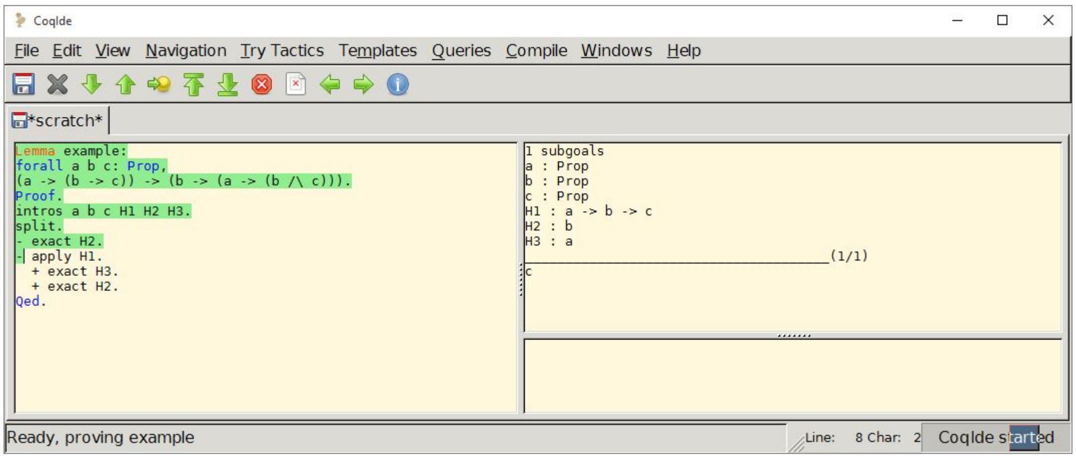
```
Ready, proving example          Line:  7 Char:  12   CoqIde started
```

Source: the author

Subsequently, new versions of this proof will be constructed with the `elim` and the `apply` tactics. The induction principle for `nat` (see Section B.5.6) is implicitely used in the first two versions of the proof, and explicitly used in the third version.

```
Lemma ex_induction:
∀ x: nat, x ∗ x ≥ x.
Proof.
intros x.
induction x.
− simpl.
  apply le_n.
− replace (S x ∗ S x) with (x ∗ x + x + x + 1).
  + apply le_plus_trans with (p:= x) in IHx.
    apply le_plus_trans with (p:= x) in IHx.
```

Figure B.9: Before `apply H1`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

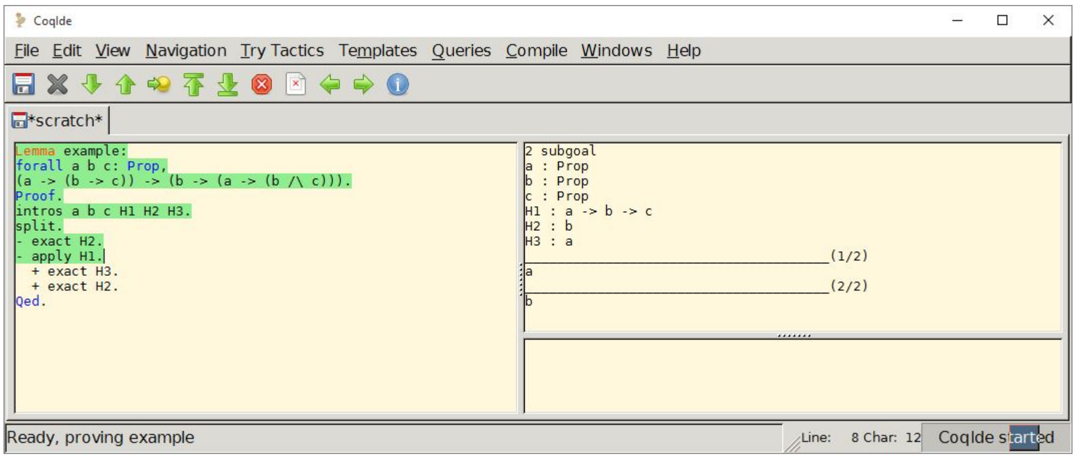
```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
1 subgoals
a : Prop
b : Prop
c : Prop
H1 : a -> b -> c
H2 : b
H3 : a
______________________________________(1/1)
c
```

```
Ready, proving example        Line:   8 Char:  2   CoqIde started
```

Source: the author

Figure B.10: After `apply H1`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
2 subgoal
a : Prop
b : Prop
c : Prop
H1 : a -> b -> c
H2 : b
H3 : a
______________________________________(1/2)
a
______________________________________(2/2)
b
```

```
Ready, proving example        Line:   8 Char:  12   CoqIde started
```

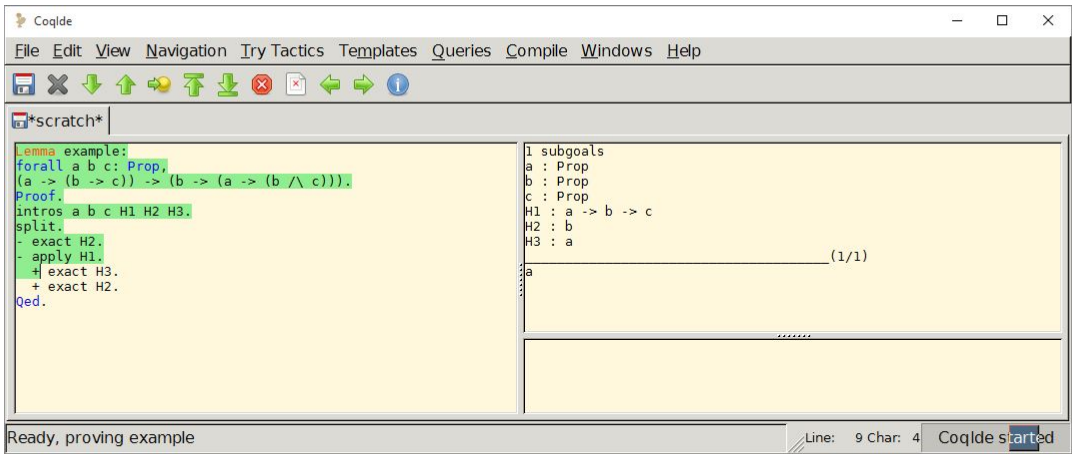
Source: the author

```
    apply plus_le_compat_r with (p:= 1) in IHx.
    replace (S x) with (x + 1).
    * exact IHx.
    * omega.
  + replace (S x) with (x + 1).
    * rewrite mult_plus_distr_r.
      rewrite mult_plus_distr_l.
      omega.
    * omega.
Qed.
```

The tactic `elim` is similar to `induction`, with the difference that the assumptions of the goal are not automatically brought into the context. For this, we have to use a `intros` tactic:

```
Lemma ex_elim:
```

Figure B.11: Before `exact H3`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

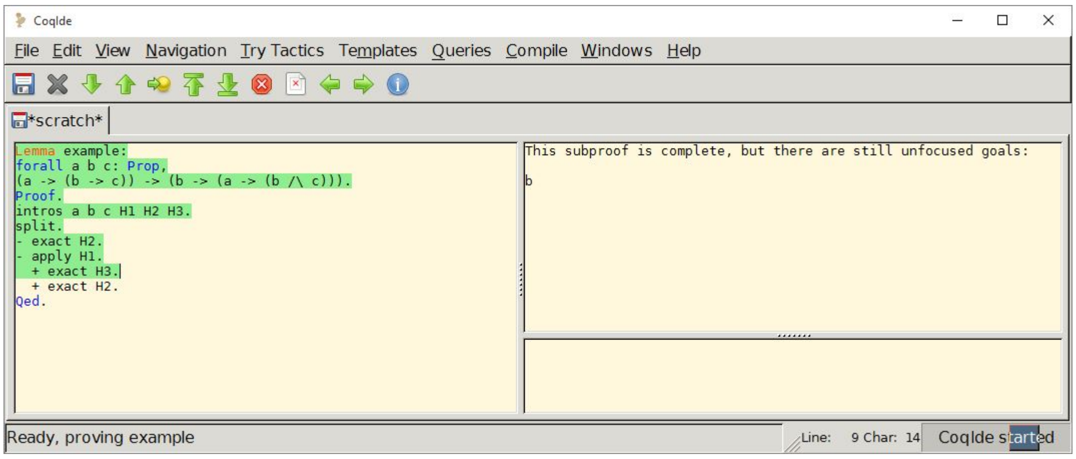
```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
1 subgoals
a : Prop
b : Prop
c : Prop
H1 : a -> b -> c
H2 : b
H3 : a
______________________________________(1/1)
a
```

```
Ready, proving example        Line:   9 Char:   4   CoqIde started
```

Source: the author

Figure B.12: After `exact H3`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

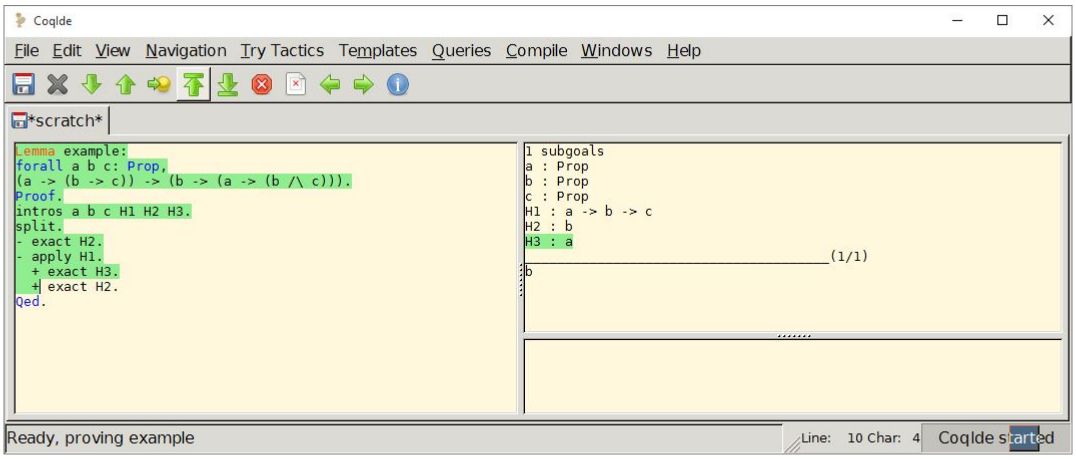
```
This subproof is complete, but there are still unfocused goals:

b
```

```
Ready, proving example        Line:   9 Char:  14   CoqIde started
```

Source: the author

```
∀ x: nat, x ∗ x ≥ x.
Proof.
intros x.
elim x.
− simpl.
  apply le_n.
− (∗ NEW ∗) intros n IHn.
  replace (S n ∗ S n) with (n ∗ n + n + n + 1).
  + apply le_plus_trans with (p:= n) in IHn.
    apply le_plus_trans with (p:= n) in IHn.
    apply plus_le_compat_r with (p:= 1) in IHn.
    replace (S n) with (n + 1).
    ∗ exact IHn.
```

Figure B.13: Before `exact H2`

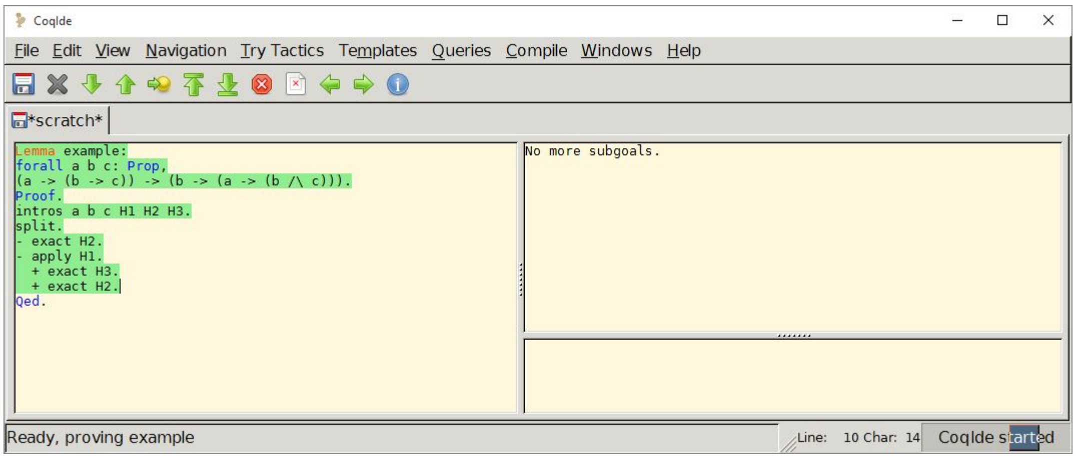

CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch
Ready, proving example
Line: 10 Char: 4
CoqIde started

```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
1 subgoals
a : Prop
b : Prop
c : Prop
H1 : a -> b -> c
H2 : b
H3 : a
______________________________________(1/1)
b
```

Source: the author

Figure B.14: After `exact H2`


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch
Ready, proving example
Line: 10 Char: 14
CoqIde started

```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
No more subgoals.
```

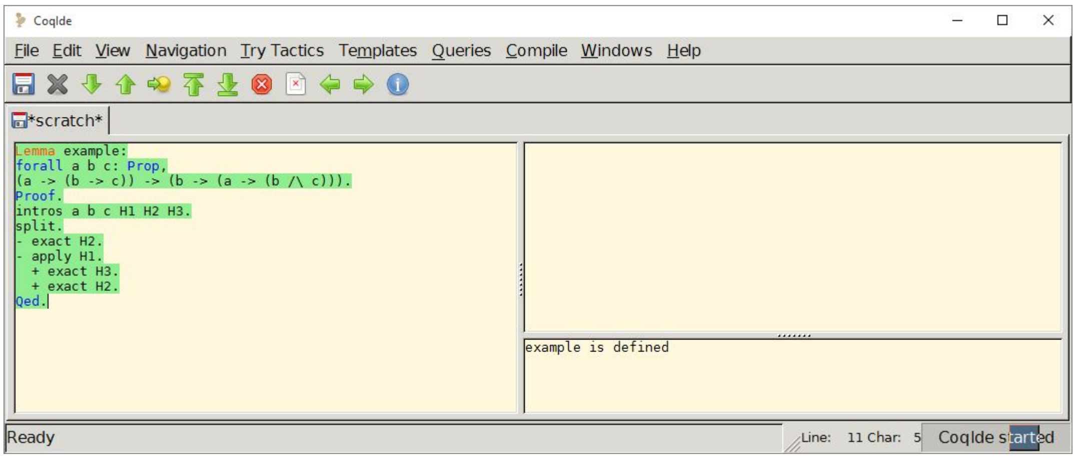
Source: the author

```
    * omega.
  + replace (S n) with (n + 1).
    * rewrite mult_plus_distr_r.
      rewrite mult_plus_distr_l.
      omega.
    * omega.
Qed.
```

Finally, the proof could have been done without using both `induction` and `elim`. Instead, the `apply` tactic with a reference to the name of the correct induction principle (which in this case is `nat_ind`) would result in a similar proof:

```
Lemma ex_apply:
∀ x: nat, x ∗ x ≥ x.
Proof.
```

Figure B.15: End of the proof


CoqIde
File Edit View Navigation Try Tactics Templates Queries Compile Windows Help
scratch

```
Lemma example:
forall a b c: Prop,
(a -> (b -> c)) -> (b -> (a -> (b /\ c))).
Proof.
intros a b c H1 H2 H3.
split.
- exact H2.
- apply H1.
  + exact H3.
  + exact H2.
Qed.
```

```
example is defined
```

```
Ready                    Line: 11 Char: 5   CoqIde started
```

Source: the author

```
intros x.
(* NEW *) generalize x.
(* NEW *) apply nat_ind.
- simpl.
  apply le_n.
- (* NEW *) intros n IHn.
  replace (S n * S n) with (n * n + n + n + 1).
  + apply le_plus_trans with (p:= n) in IHn.
    apply le_plus_trans with (p:= n) in IHn.
    Search (_ ≤ _).
    apply plus_le_compat_r with (p:= 1) in IHn.
    replace (S n) with (n + 1).
    * exact IHn.
    * omega.
  + replace (S n) with (n + 1).
    * rewrite mult_plus_distr_r.
      rewrite mult_plus_distr_l.
      omega.
    * omega.
Qed.
```

Observe, in this case, the necessity of generalizing the goal on x, and also invoking explicitly the induction principle required for he proof.

B.7 Computing

Gallina can be used to represent programs (terms) that can be computed (by means of mechanisms similar to the substitutions of the lambda calculus). Thus, it is suitable also as a functional programming language. Although computation is not the main objective of Coq, it is important as an auxiliary tool in reasoning because some proofs can benefit from some kind of computation. Also, computing in Coq enables the user to experiment and check function defintions before advancing with the formalization. *Gallina* does not offer special features for data input or output, as in plain functional programming.

Computation is done via a series of reductions of four different kinds, depending on the type of substitution that is considered. Also, different strategies can be selected by the users (call-by-value, lazy etc). Coq has the *strong normalization* property, which means that all computations terminate, leading to the normal form of the original term. A normal form is a term that can not be further simplified and represents the final value of the computation. This characteristic is important in order to ensure the consistency of the system, however represents a limitation in its expressive power, in the sense that not all computations can be carried out in Coq (and this refers to the computations that do not reach an end). This is in opposition to computation that can be defined in most programming languages. Still, it is expressive enough in order to cope with different needs and uses.

The basic interaction command for doing computation in Vernacular is the `Eval compute in`, for a call-by-value term evaluation. It takes an expression as argument and reduces it to its normal form. As an example, consider the factorial function defined in Section B.5.4. The factorial of 5 can be computed by typing:

```
Eval compute in fact 5.
```

The result, `120`, is printed as an immediate consequence.

Computation in proof scripts (that is, the simplification of terms in the current goal or any of the hypotheses) can be done by invoking tactics such as `simpl`, `red`, `compute`, `vm_compute`, `cbv` and `lazy`, depending on the kind of reduction rules to be used and the desired strategy.

It should be stressed, however, that the main objective of Coq is not to provide for the efficient execution of Gallina programs. Instead, it offers an extraction mechanism that can be used to convert these to other (real) programs written in other programming languages (such as OCAML or Haskell), which can then be compiled or interpreted to meet various performance criteria.

C Definitions, Lemmas and Theorems

In the following sections we present a list of the main definitions, lemmas and theorems of the main libraries included in the formalization. Each section corresponds to a different library, and has two subsections in most of the cases: the first contains the key definitions of the library, and the second the main lemmas that reason upon these definitions.

In the first subsection of each section, the definitions reproduce exactly the contents of the corresponding source file. A brief paragraph explains what is being defined in each case.

In the second subsection of each section, each lemma entry includes the lemma or theorem name, a brief description and its statement. The name is the same used in the formalization. The description is a short paragraph about the purpose and use of the lemma/theorem. The statement has been simplified from the version included in the formalization in order to ease its understanding. This simplification consists mostly of dropping the notations used for lists and disjoint unions, type annotations and mappings in general. For the details and also for the corresponding proof scripts, please refer to the repository where the source code is available (<https://github.com/mvmramos/v1>).

The libraries are presented in the same order as they are discussed in Chapter 6. For each library, the lemmas and theorems are presented in the same order as they appear in the source files. The auxiliary libraries "misc_arith", "misc_list", "cfg", "cfl", "trees" and "pigeon" are not included here.

C.1 Library "union"

Closure of context-free languages under the union operation. The objective of this library is to prove that the union of any two context-free languages is also a context-free language. This result is expressed in the lemmas `l_uni_is_cfl`, `l_uni_correct` and `l_uni_correct_inv`.

C.1.1 Definitions

- Type of the non-terminal symbols of the union grammar. It uses all the symbols of both grammars, plus a new one (`Start_uni`):

```
Inductive g_uni_nt: Type:=
| Start_uni
| Transf1_uni_nt: non_terminal_1 → g_uni_nt
| Transf2_uni_nt: non_terminal_2 → g_uni_nt.
```

- Rules of the union grammar:

```
Inductive g_uni_rules
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal): g_uni_nt → sfu → Prop :=
| Start1_uni: g_uni_rules g1 g2
          Start_uni [inl (Transf1_uni_nt (start_symbol g1))]
| Start2_uni: g_uni_rules g1 g2
          Start_uni [inl (Transf2_uni_nt (start_symbol g2))]
| Lift1_uni: ∀ nt: non_terminal_1,
          ∀ s: list (non_terminal_1 + terminal),
          rules g1 nt s →
          g_uni_rules g1 g2 (Transf1_uni_nt nt)
                      (map g_uni_sf_lift1 s)
| Lift2_uni: ∀ nt: non_terminal_2,
          ∀ s: list (non_terminal_2 + terminal),
          rules g2 nt s →
          g_uni_rules g1 g2 (Transf2_uni_nt nt)
                    (map g_uni_sf_lift2 s).
```

- Union grammar:

```
Definition g_uni
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal): (cfg g_uni_nt terminal):= {|
start_symbol:= Start_uni;
rules:= g_uni_rules g1 g2;
rules_finite:= g_uni_finite g1 g2
|}.
```

C.1.2 Lemmas

- `g_uni_correct_left`:
  Every sentential form generated by grammar g_1 is also generated by grammar

$(g\_uni\ g_1\ g_2)$.
$\forall\ g_1, g_2, s,$
$generates\ g_1\ s \to$
$generates\ (g\_uni\ g_1\ g_2)\ s$

- `g_uni_correct_right`:
  Every sentential form generated by grammar g_2 is also generated by grammar $(g\_uni\ g_1\ g_2)$.
  $\forall\ g_1, g_2, s,$
  $generates\ g_2\ s \to generates\ (g\_uni\ g_1\ g_2)\ s$

- `g_uni_correct`:
  Unification of `g_uni_correct_left` and `g_uni_correct_right`.
  $\forall\ g_1, g_2, s_1, s_2,$
  $(generates\ g_1\ s_1 \to generates\ (g\_uni\ g_1\ g_2)\ s_1 \wedge$
  $(generates\ g_2\ s_2 \to generates\ (g\_uni\ g_1\ g_2)\ s_2)$

- `g_uni_correct_inv`:
  If $(g\_uni\ g_1\ g_2)$ generates sentential form s, then s is either the start symbol of $(g\_uni\ g_1\ g_2)$ or a sentential form of g_1 or a sentential form of g_2.
  $\forall\ g_1, g_2, s,$
  $generates\ (g\_uni\ g_1\ g_2)\ s \to$
  $(s = (start\_symbol\ (g\_uni\ g_1\ g_2))) \vee$
  $(\exists\ s_1, (s = s_1) \wedge generates\ g_1\ s_1) \vee$
  $(\exists\ s_2, (s = s_2) \wedge generates\ g_2\ s_2)$

- `l_uni_is_cfl`:
  The union of two context-free languages is a context-free language.
  $\forall\ l_1, l_2,$
  $cfl\ l_1 \to$
  $cfl\ l_2 \to$
  $cfl\ (l\_uni\ l_1\ l_2)$

- `l_uni_correct`:
  If s belongs to l_1 or to l_2, then it must also belong to $(l\_uni\ l_1\ l_2)$.
  $\forall\ l_1, l_2, s,$
  $(s \in l_1 \vee s \in l_2) \to s \in (l\_uni\ l_1\ l_2)$

- `l_uni_correct_inv`:
  If s belongs $(l\_uni\ l_1\ l_2)$ then it must belong to either l_1 or l_2.
  $\forall\ l_1, l_2, s,$

$s \in (l\_uni\ l_1\ l_2) \rightarrow$
$s \in l_1 \vee s \in l_2$

C.2 Library "concatenation"

Closure of context-free languages under the concatenation operation. The objective of this library is to prove that the concatenation of any two context-free languages is also a context-free language. This result is expressed in the lemmas `l_cat_is_cfl`, `l_cat_correct` and `l_cat_correct_inv`.

C.2.1 Definitions

- Type of the non-terminal symbols of the concatenation grammar. It uses all the symbols of both grammars, plus a new one (`Start_cat`):

```
Inductive g_cat_nt: Type:=
| Start_cat
| Transf1_cat_nt: non_terminal_1 → g_cat_nt
| Transf2_cat_nt: non_terminal_2 → g_cat_nt.
```

- Rules of the concatenation grammar:

```
Inductive g_cat_rules
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal): g_cat_nt → sfc → Prop :=
| New_cat: g_cat_rules g1 g2 Start_cat
          ([ inl (Transf1_cat_nt (start_symbol g1))]++
       [inl (Transf2_cat_nt (start_symbol g2))])
| Lift1_cat: ∀ nt s,
             rules g1 nt s →
             g_cat_rules g1 g2 (Transf1_cat_nt nt)
                          (map g_cat_sf_lift1 s)
| Lift2_cat: ∀ nt s,
             rules g2 nt s →
             g_cat_rules g1 g2 (Transf2_cat_nt nt)
                          (map g_cat_sf_lift2 s).
```

- Concatenation grammar:

```
Definition g_cat
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal): (cfg g_cat_nt terminal):= {|
start_symbol:= Start_cat;
```

```
    rules:= g_cat_rules g1 g2;
    rules_finite:= g_cat_finite g1 g2
    |}.
```

C.2.2 Lemmas

- `derives_add_cat_left`:

  Derivations in grammar g_1 are preserved in grammar (g_cat g_1 g_2).

  $\forall\, g_1, g_2, s, s'$

  $derives\ g_1\ s\ s' \rightarrow derives\ (g\_cat\ g_1\ g_2)\ s\ s'$

- `derives_add_cat_right`:

  Derivations in grammar g_2 are preserved in grammar (g_cat g_1 g_2).

  $\forall\, g_1, g_2, s, s',$

  $derives\ g_2\ s\ s' \rightarrow derives\ (g\_cat\ g_1\ g_2)\ s\ s'$

- `g_cat_correct_aux`:

  Derivations combine in the concatenation grammar.

  $\forall\, g_1, g_2, s_{11}, s_{12}, s_{21}, s_{22},$

  $derives\ g_1\ s_{11}\ s_{12} \wedge derives\ g_2\ s_{21}\ s_{22} \rightarrow$

  $derives\ (g\_cat\ g_1\ g_2)\ (s_{11} \cdot s_{21})\ (s_{12} \cdot s_{22})$

- `g_cat_correct`:

  Sentential forms concatenate in the concatenation grammar.

  $\forall g_1, g_2, s_1, s_2,$

  $generates\ g_1\ s_1 \wedge generates\ g_2\ s_2 \rightarrow$

  $generates\ (g\_cat\ g_1\ g_2)\ (s_1 \cdot s_2)$

- `g_cat_correct_inv`:

  A sentential form of (g_cat g_1 g_2) must either be the the start symbol of the grammar or a concatenation of a sentential form of g_1 and a sentential form of g_2.

  $\forall\, g_1, g_2, s,$

  $generates\ (g\_cat\ g_1\ g_2)\ s \rightarrow$

  $s = (start\_symbol\ (g\_cat\ g_1\ g_2)) \vee$

  $\exists\, s_1, s_2,$

  $s = s_1 \cdot s_2 \wedge$

  $generates\ g_1\ s_1 \wedge$

  $generates\ g_2\ s_2$

- `l_cat_is_cfl`:

  The concatenation of two context-free languages is also a context-free language.

  $\forall\, l_1, l_2,$

$cfl\ l_1 \rightarrow$
$cfl\ l_2 \rightarrow$
$cfl\ (l\_cat\ l_1\ l_2)$

- `l_cat_correct`:
  If s_1 and s_2 are, respectively, sentences of l_1 and l_2, then $s_1 \cdot s_2$ is a sentence of $(l\_cat\ l_1\ l_2)$.
  $\forall\ l_1, l_2, s_1, s_2,$
  $(s_1 \in l_1 \wedge s_2 \in l_2) \rightarrow (l\_cat\ l_1\ l_2)\ (s_1 \cdot s_2)$

- `l_cat_correct_inv`:
  Every sentence that belongs to $(l\_cat\ l_1\ l_2)$ is the concatenation of a sentence of l_1 with a sentence of l_2.
  $\forall\ l_1, l_2, s,$
  $(l\_cat\ l_1\ l_2)\ s \rightarrow$
  $\exists\ s_1\ s_2,$
  $s = s_1 \cdot s_2 \wedge s_1 \in l_1 \wedge s_2 \in l_2$

C.3 Library “closure”

Closure of context-free languages under the Kleene star operation. The objective of this library is to prove that the Kleene star of any context-free language is also a context-free language. This result is expressed in the lemmas `l_clo_is_cfl`, `l_clo_correct` and `l_clo_correct_inv`.

C.3.1 Definitions

- Type of the non-terminal symbols of the closure grammar. It uses all the symbols of the original grammar, plus a new one (`Start_clo`):

```
Inductive g_clo_nt: Type :=
| Start_clo : g_clo_nt
| Transf_clo_nt : non_terminal → g_clo_nt.
```

- Rules of the closure grammar:

```
Inductive g_clo_rules
(g: cfg non_terminal terminal): g_clo_nt → sfc → Prop :=
| New1_clo: g_clo_rules g Start_clo ([inl Start_clo] ++
              [inl (Transf_clo_nt (start_symbol g))])
| New2_clo: g_clo_rules g Start_clo []
| Lift_clo: ∀ nt: non_terminal,
∀ s: sf,
rules g nt s →
g_clo_rules g (Transf_clo_nt nt)
          (map g_clo_sf_lift s).
```

- Closure grammar:

```
 Definition g_clo
(g: cfg non_terminal terminal): (cfg g_clo_nt terminal):= {|
 start_symbol:= Start_clo;
 rules:= g_clo_rules g;
 rules_finite:= g_clo_finite g
 |}.
```

C.3.2 Lemmas

- `derives_add_clo`:
  Every sentential form derived in g can also be derived in (l_clo g).
  $\forall\, g, s_1, s_2,$
  $derives\ g\ s_1\ s_2 \rightarrow derives\ (g\_clo\ g)\ s_1\ s_2$

- `g_clo_correct`:
  Correctness of (g_clo g).
  $\forall\, g, s, s',$
  $generates\ (g\_clo\ g)\ \varepsilon\ \wedge$
  $(generates\ (g\_clo\ g)\ s' \wedge generates\ g\ s \rightarrow generates\ (g\_clo\ g)\ (s' \cdot s))$

- `g_clo_correct_inv`:
  Completeness of (g_clo g).
  $\forall\, g, s,$
  $generates\ (g\_clo\ g)\ s \rightarrow$
  $(s = \varepsilon) \vee$
  $(s = start\_symbol\ (g\_clo\ g)) \vee$
  $(\exists\, s', s'', generates\ (g\_clo\ g)\ s' \wedge generates\ g\ s'' \wedge s = s' \cdot s'')$

- `produces_split`:
  Qualifies the sentences produced by (g_clo g).
  $\forall\, g, w,$
  $produces\ (g\_clo\ g)\ w \rightarrow$
  $w = \varepsilon \vee$
  $\exists\, w_1, w_2,$
  $produces\ (g\_clo\ g)\ w_1 \wedge$

$produces\ g\ w_2 \wedge$
$w = w_1 \cdot w_2$

- `derives_g_clo_derives_g`:
  If s is a sentential form of g and ($g\_clo\ g$) derives w from s, then s derives w also in g.
  $\forall\ g, s, w,$
  $derives\ (g\_clo\ g)\ s\ w \rightarrow$
  $\forall sg,$
  $s = sg \rightarrow$
  $derives\ g\ sg\ w$

- `g_clo_correct_inv_v2`:
  Sentences produced by ($g\_clo\ g$) are the concatenation of sentences individually derived by g.
  $\forall\ g, w,$
  $produces\ (g\_clo\ g)\ w \rightarrow$
  $\exists\ w',$
  $(\forall\ s, In\ s\ w' \rightarrow produces\ g\ s) \wedge w = flat\ w'$

- `l_clo_is_cfl`:
  The Kleene star (closure) of a context-free language is also a context-free language.
  $\forall\ l,$
  $cfl\ l \rightarrow$
  $cfl\ (l\_clo\ l)$

- `l_clo_correct`:
  Correctness of ($l\_clo\ l$).
  $\forall\ l,$
  $\varepsilon \in (l\_clo\ l) \wedge$
  $\forall\ s_1, s_2,$
  $s_1 \in (l\_clo\ l) \rightarrow$
  $s_2 \in l \rightarrow$
  $(s_1 \cdot s_2) \in (l\_clo\ l)$

- `l_clo_correct_inv`:
  Completeness of ($l\_clo\ l$).
  $\forall\ l, s,$
  $(l\_clo\ l)\ s \rightarrow$
  $s = \varepsilon \vee$
  $\exists\ s_1, s_2,$
  $s = s_1 \cdot s_2 \wedge$

$s_1 \in (l\_clo\ l) \wedge$
$s_2 \in l$

C.4 Library “emptyrules”

Elimination of empty rules in context-free grammars, while preserving the language generated. The objective of this library is to prove that every context-free grammar generates a language that can also be generated by an equivalent grammar which is (i) completely free of empty rules if the language does not contain the empty string, or (ii) has only one empty rule (with the start symbol in the left-hand side) if the language contains the empty string. Also, the start symbol of the new grammar does not appear in the right-hand side of any rule. This result is expressed in lemma `g_emp'_correct`.

C.4.1 Definitions

- Type of the non-terminal symbols of the new grammar, free of empty rules. It uses all the symbols of the original grammar, plus a new one (`New_ss`):

```
Inductive non_terminal': Type:=
| Lift_nt: non_terminal → non_terminal'
| New_ss.
```

- Rules of the new grammar, totally free of empty rules:

```
Inductive g_emp_rules
(g: cfg _ _): non_terminal' → sf' → Prop :=
| Lift_direct :
      ∀ left: non_terminal,
      ∀ right: sf,
      right ≠ [] → rules g left right →
      g_emp_rules g (Lift_nt left) (map symbol_lift right)
| Lift_indirect:
      ∀ left: non_terminal,
      ∀ right: sf,
      g_emp_rules g (Lift_nt left) (map symbol_lift right)→
      ∀ s1 s2: sf,
      ∀ s: non_terminal,
      right = s1 ++(inl s) :: s2 →
      empty g (inl s) →
      s1 ++s2 ≠ [] →
      g_emp_rules g (Lift_nt left) (map symbol_lift (s1 ++s2))
| Lift_start_emp:
```

```
            g_emp_rules g New_ss [inl (Lift_nt (start_symbol g))].
```

- New grammar, totally free of empty rules:

```
  Definition g_emp (g: cfg non_terminal terminal):
 cfg non_terminal' terminal := {|
  start_symbol:= New_ss;
  rules:= g_emp_rules g;
  rules_finite:= g_emp_finite g
  |}.
```

- Rules of the final grammar, may contain a single empty rule. Based on the rules defined in g_emp_rules:

```
  Inductive g_emp'_rules (g: cfg _ _):
 non_terminal' non_terminal → sf' → Prop :=
 | Lift_all:
        ∀ left: non_terminal' _,
        ∀ right: sf',
        rules (g_emp g) left right →
        g_emp'_rules g left right
 | Lift_empty:
        empty g (inl (start_symbol g)) →
     g_emp'_rules g (start_symbol (g_emp g)) [].
```

- Final grammar, may contain a single empty rule:

```
  Definition g_emp' (g: cfg non_terminal terminal):
 cfg (non_terminal' _) terminal := {|
  start_symbol:= New_ss _;
  rules:= g_emp'_rules g;
  rules_finite:= g_emp'_finite g
  |}.
```

C.4.2 Lemmas

- `start_symbol_not_in_rhs_g_emp`:
  The start symbol does not appear in the right-hand side of any rule in (*g_emp g*).
  $\forall\ g,$
  start_symbol_not_in_rhs (*g_emp g*)

- `g_emp_not_derives_empty`:
  The empty string can not be derived in grammar (*g_emp g*).

$\forall\, g, n,$
$\neg\, derives\ (g\_emp\ g)\ n\ \varepsilon$

- `g_emp_has_no_empty_rules`:
  Grammar (*g_emp g*) contains no empty rules.
  $\forall\, g,$
  $has\_no\_empty\_rules\ (g\_emp\ g)$

- `in_left_not_empty`:
  The non-terminal symbol in the left-hand side of any rule of (*g_emp g*) does not generate the empty string.
  $\forall\, g, x, right,$
  $rules\ (g\_emp\ g)\ x\ right \rightarrow \neg\, empty\ (g\_emp\ g)\ x$

- `in_right_not_empty`:
  No symbol that appears in the right-hand side of any rule of (*g_emp g*) generates the empty string.
  $\forall\, g, x, n, right,$
  $rules\ (g\_emp\ g)\ x\ right \rightarrow In\ n\ right \rightarrow \neg\, empty\ (g\_emp\ g)\ n$

- `rules_g_emp_g`:
  A rule of (*g_emp g*) is either a rule of *g* or its right-hand side can be derived from the left-hand side in *g*.
  $\forall\, g, left, right,$
  $rules\ (g\_emp\ g)\ left\ right \rightarrow$
  $rules\ g\ left\ right \vee derives\ g\ left\ right$

- `derives_g_emp_g`:
  Every sentential form that can be derived in (*g_emp g*) can also be derived in *g*.
  $\forall\, g, n, s,$
  $derives\ (g\_emp\ g)\ n\ s \rightarrow derives\ g\ n\ s$

- `rules_g_g_emp`:
  Every non-empty rule of *g* is also a rule of (*g_emp g*).
  $\forall\, g, left, right,$
  $right \neq \varepsilon \rightarrow$
  $rules\ g\ left\ right \rightarrow$
  $rules\ (g\_emp\ g)\ left\ right$

- `derives_g_g_emp`:
  Every non-empty string derivable in *g* can also be derived in (*g_emp g*).
  $\forall\, g, n, s,$

$s \neq \varepsilon \rightarrow$
derives g n s $\rightarrow$ *derives* (*g_emp g*) *n s*

- `g_emp_correct`:
  Every context-free grammar is in relation to another one such that they generate the same language (except for the empty string) and the latter has no empty rules and its start symbol does not appear in the right-hand side of any rule.
  $\forall\, g,$
  g_equiv_without_empty (*g_emp g*) *g* $\wedge$
  has_no_empty_rules (*g_emp g*) $\wedge$
  start_symbol_not_in_rhs (*g_emp g*)

- `start_symbol_not_in_rhs_g_emp'`:
  The start symbol does not appear in the right-hand side of any rule in (*g_emp′ g*).
  $\forall\, g,$
  start_symbol_not_in_rhs (*g_emp′ g*)

- `derives_g_emp'_or`:
  A sentence that is derived from the start symbol of (*g_emp′ g*) is either empty or can also be derived in the same grammar from the start symbol of (*g_emp g*).
  $\forall\, g, s,$
  derives (*g_emp′ g*) (*start_symbol* (*g_emp′ g*)) *s* $\rightarrow$
  $s = \varepsilon \vee$
  derives (*g_emp′ g*) (*start_symbol* (*g_emp g*)) *s*

- `derives_g_emp'_empty`:
  If (*g_emp′ g*) derives the empty string, then (*g_emp′ g*) has the empty rule.
  $\forall\, g,$
  derives (*g_emp′ g*) (*start_symbol* (*g_emp′ g*)) $\varepsilon \rightarrow$
  rules (*g_emp′ g*) (*start_symbol* (*g_emp′ g*)) ε

- `g_emp_equiv_g_emp'`:
  Every non-empty sentence that can be derived in (*g_emp′ g*) can also be derived in (*g_emp g*). Every sentence that can be derived in (*g_emp g*) can also be derived in (*g_emp′ g*).
  $\forall\, g, s,$
  ($s \neq \varepsilon \rightarrow$
  derives (*g_emp′ g*) (*start_symbol* (*g_emp′ g*)) *s* $\rightarrow$
  derives (*g_emp g*) (*start_symbol* (*g_emp g*)) *s*)
  $\wedge$
  (*derives* (*g_emp g*) (*start_symbol* (*g_emp g*)) *s* $\rightarrow$
  derives (*g_emp′ g*) (*start_symbol* (*g_emp′ g*)) *s*)

- `g_emp'_has_one_empty_rule`:
  If g generates the empty string, then (g_emp' g) has one empty rule.
  $\forall\, g,$
  $generates\_empty\ g \rightarrow has\_one\_empty\_rule\ (g\_emp'\ g)$

- `g_emp'_has_no_empty_rules`:
  If g does not generate the empty string, then (g_emp' g) has no empty rules.
  $\forall\, g,$
  $\neg\, generates\_empty\ g \rightarrow has\_no\_empty\_rules\ (g\_emp'\ g)$

- `g_emp'_correct`:
  Every context-free grammar is in relation to another one such that they generate the same language and the latter has at most one empty rule and its start symbol does not appear in the right-hand side of any rule.
  $\forall g,$
  $g\_equiv\ (g\_emp'\ g)\ g \wedge$
  $(produces\_empty\ g \rightarrow has\_one\_empty\_rule\ (g\_emp'\ g)) \wedge$
  $(\neg produces\_empty\ g \rightarrow has\_no\_empty\_rules\ (g\_emp'\ g)) \wedge$
  $start\_symbol\_not\_in\_rhs\ (g\_emp'\ g)$

- `no_empty_rules_no_empty`:
  A grammar without empty rules cannot derive the empty string.
  $\forall\, g,$
  $has\_no\_empty\_rules\ g \rightarrow$
  $\neg\, derives\ g\ (start\_symbol\ g)\ \varepsilon$

C.5 Library “unitrules”

Elimination of unit rules in context-free grammars, while preserving the language generated. The objective of this library is to prove that every context-free grammar generates a language that can also be generated by an equivalent grammar which is completely free of unit rules. This result is expressed in lemma `g_unit_correct`.

C.5.1 Definitions

- Two non-terminal symbols a and b are in a *unit* relation in grammar g if $a \Rightarrow_g^+ b$:

```
 Inductive unit
(g: cfg non_terminal terminal)
(a: non_terminal): non_terminal → Prop:=
 | unit_rule: ∀ (b: non_terminal),
              rules g a [inl b] → unit g a b
```

```
| unit_trans: ∀ b c: non_terminal,
            unit g a b →
            unit g b c →
            unit g a c.
```

- Rules of the new grammar, free of unit rules:

```
 Inductive g_unit_rules (g: cfg _ _):
non_terminal → sf → Prop :=
| Lift_direct':
       ∀ left: non_terminal,
       ∀ right: sf,
       (∀ r: non_terminal,
       right ≠ [inl r]) → rules g left right →
       g_unit_rules g left right
| Lift_indirect':
       ∀ a b: non_terminal,
       unit g a b →
       ∀ right: sf,
       rules g b right →
       (∀ c: non_terminal,
       right ≠ [inl c]) →
       g_unit_rules g a right.
```

- New grammar, free of unit rules:

```
Definition g_unit (g: cfg _ _): cfg _ _ := {|
start_symbol:= start_symbol g;
rules:= g_unit_rules g;
rules_finite:= g_unit_finite g
|}.
```

C.5.2 Lemmas

- `unit_exists`:
  If non-terminal symbols a and b have a *unit* relation in g (that is, if $a \Rightarrow_g^+ b$), then a is the left-hand side of some rule in g.
  $\forall\, g, a, b,$
  $unit\ g\ a\ b \rightarrow$
  $\exists\, c, rules\ g\ a\ c$

- `unit_not_unit`:
  There are no non-terminal symbols that have a *unit* relation in ($g\_unit\ g$).

$\forall g, a, b,$
$\neg\ unit\ (g\_unit\ g)\ a\ b$

- `unit_derives`:
  If two non-terminal symbols *a* and *b* have a *unit* relation in *g*, then *a* derives *b* in *g*.
  $\forall\ g, a, b,$
  $unit\ g\ a\ b \rightarrow$
  $derives\ g\ a\ b$

- `rules_g_unit_g`:
  A rule of (*g_unit g*) is either a rule of *g* or its left-hand side derives its right-hand side in *g*.
  $\forall\ g, left, right,$
  $rules\ (g\_unit\ g)\ left\ right \rightarrow$
  $rules\ g\ left\ right \vee derives\ g\ left\ right$

- `rules_g_unit_g'`:
  A variant of `rules_g_unit_g`. A rule of (*g_unit g*) is either a rule of *g* or its left-hand side has a *unit* relation with another non-terminal symbol, which in turn is the left-hand side of a rule of *g* whose right-hand side is the right-hand side of the rule in (*g_unit g*).
  $\forall\ g, left, right,$
  $rules\ (g\_unit\ g)\ left\ right \rightarrow$
  $rules\ g\ left\ right \vee$
  $\exists\ left', unit\ g\ left\ left' \wedge rules\ g\ left'\ right$

- `rules_g_unit_not_unit`:
  Grammar (*g_unit g*) is free of unit rules.
  $\forall\ g, left, right,$
  $(rules\ (g\_unit\ g)\ left\ right) \rightarrow$
  $(\neg\ \exists\ n, right = n)$

- `generates_g_unit_g`:
  Every sentential form derived by (*g_unit g*) can also be derived by *g*.
  $\forall\ g, s,$
  $generates\ (g\_unit\ g)\ s \rightarrow generates\ g\ s$

- `rules_g_g_unit`:
  Every non-unit rule of *g* is also a rule of (*g_unit g*).
  $\forall\ g, left, right,$
  $rules\ g\ left\ right \rightarrow$

$(\forall\, n, right \neq n) \rightarrow$
$rules\ (g\_unit\ g)\ left\ right$

- `derives_g_g_unit`:
  Every sentence that is derived in g can also be derived in (g_unit g).
  $\forall\, g, s, n,$
  $derives\ g\ n\ s \rightarrow derives\ (g\_unit\ g)\ n\ s$

- `g_equiv_unit`:
  Grammars g and (g_unit g) generate exactly the same set of sentences.
  $\forall\, g,$
  $g\_equiv\ (g\_unit\ g)\ g$

- `g_unit_has_no_unit_rules`:
  A variant of `rules_g_unit_not_unit`. Grammar (g_unit g) is free of unit rules.
  $\forall\, g,$
  $has\_no\_unit\_rules\ (g\_unit\ g)$

- `g_unit_correct`:
  Every context-free grammar g is in relation with another grammar (g_unit g) which generates the same language and is free of unit rules.
  $\forall\, g,$
  $g\_equiv\ (g\_unit\ g)\ g \wedge$
  $has\_no\_unit\_rules\ (g\_unit\ g)$

C.6 Library “useless”

Elimination of useless symbols in context-free grammars, while preserving the language generated. The objective of this library is to prove that every context-free grammar generates a language that can also be generated by an equivalent grammar which is completely free of useless symbols. This result is expressed in lemma `g_use_correct`.

C.6.1 Definitions

- A symbol s (terminal ou non-terminal) is *useful* in grammar g if $s \Rightarrow_g^* w$, where w consists only of terminal symbols:

```
 Definition useful
(g: cfg _ _) (s: non_terminal + terminal): Prop:=
 match s with
 | inr t ⇒ True
| inl n ⇒ ∃ s: sentence, derives g [inl n] (map term_lift s)
end.
```

- Rules of the new grammar, free of useless symbols:

```
 Inductive g_use_rules (g: cfg _ _):
non_terminal → sf → Prop :=
 | Lift_use : ∀ left: non_terminal,
              ∀ right: sf,
              rules g left right →
              useful g (inl left) →
              useful_sf g right →
              g_use_rules g left right.
```

- New grammar, free of useless symbols:

```
Definition g_use (g: cfg _ _): cfg _ _:= {|
start_symbol:= start_symbol g;
rules:= g_use_rules g;
rules_finite:= g_use_finite g
|}.
```

C.6.2 Lemmas

- `useful_exists`:
  If a symbol is *useful* in g (that is, if a word consisting only of terminal symbols can be derived from this symbol in g), then there must exist a rule in g with this symbol in either the left- or right-hand side.
  $\forall\, g, n,$
  $useful\ g\ n \rightarrow$
  $\exists\, left, right,$
  $rules\ g\ left\ right \wedge (n = left \vee In\ n\ right)$

- `use_step`:
  If every symbol in the right-hand side of a rule is useful, so is the non-terminal symbol in the left-hand side of the same rule.
  $\forall\, g, left, right,$
  $rules\ g\ left\ right \rightarrow$
  $(\forall\, s, In\ s\ right \rightarrow useful\ g\ s) \rightarrow$
  $useful\ g\ left$

- `useful_g_g_use`:
  If a symbol is useful in g it is also useful in (g_use g).

$\forall\, g, n,$
$useful\ g\ n \rightarrow useful\ (g\_use\ g)\ n$

- `useful_g_use`:
  Every symbol of (*g_use g*) is useful.
  $\forall\, g, n,$
  $appears\ (g\_use\ g)\ n \rightarrow useful\ (g\_use\ g)\ n$

- `produces_g_use_g`:
  Every sentence derived by (*g_use g*) can also be derived in *g*.
  $\forall\, g, s,$
  $produces\ (g\_use\ g)\ s \rightarrow produces\ g\ s$

- `produces_g_g_use`:
  Every sentence derived by *g* can also be derived in (*g_use g*).
  $\forall\, g, s,$
  $produces\ g\ s \rightarrow produces\ (g\_use\ g)\ s$

- `g_equiv_use`:
  If the language generated by grammar *g* contains at least one sentence, then (*g_use g*) and *g* generate exactly the same language.
  $\forall\, g,$
  $non\_empty\ g \rightarrow g\_equiv\ (g\_use\ g)\ g$

- `g_use_correct`:
  If the language generated by grammar *g* contains at least one sentence, then (*g_use g*) and *g* generate exactly the same language and (*g_use g*) is free of useless symbols.
  $\forall\, g,$
  $non\_empty\ g \rightarrow$
  $g\_equiv\ (g\_use\ g)\ g \wedge$
  $has\_no\_useless\_symbols\ (g\_use\ g)$

C.7 Library "inaccessible"

Elimination of inaccessible symbols in context-free grammars, while preserving the language generated. The objective of this library is to prove that every context-free grammar generates a language that can also be generated by an equivalent grammar which is completely free of inaccessible. This result is expressed in lemma `g_acc_correct`.

C.7.1 Definitions

- A symbol s (terminal or non-terminal) is *accessible* in grammar g if $S \Rightarrow \alpha s \beta$, where S is the start symbol of g and α and β are any strings of terminal and non-terminal symbols.

```
Definition accessible
(g: cfg _ _) (s: non_terminal + terminal): Prop:=
∃ s1 s2: sf, derives g [inl (start_symbol g)] (s1++s::s2).
```

- Rules of the new grammar, free of inaccessible symbols:

```
Inductive g_acc_rules (g: cfg _ _):
non_terminal → sf → Prop :=
| Lift_acc : ∀ left: non_terminal,
             ∀ right: sf,
             rules g left right → accessible g (inl left) →
                                  g_acc_rules g left right.
```

- New grammar, free of inaccessible symbols:

```
Definition g_acc (g: cfg _ _): cfg _ _ := {|
start_symbol:= start_symbol g;
rules:= g_acc_rules g;
rules_finite:= g_acc_finite g
|}.
```

C.7.2 Lemmas

Elimination of inaccessible symbols in context-free grammars, while preserving the language generated. The objective of this library is to prove that every context-free grammar generates a language that can also be generated by an equivalent grammar which is completely free of inaccessible symbols. This result is expressed in lemma `g_acc_correct`.

- `accessible_g_g_acc`:
  If a symbol is *accessible* in g (that is, it is part of a sentential form derived from the start synbol of g), then it is also accessible in ($g\_acc\ g$).
  $\forall\, g, s,$
  $accessible\ g\ s \rightarrow accessible\ (g\_acc\ g)\ s$

- `produces_g_acc_g`:
  Every sentence produced by ($g\_acc\ g$) is also produced by g.
  $\forall\, g, s,$
  $produces\ (g\_acc\ g)\ s \rightarrow produces\ g\ s$

- `acc_step`:
  If the left-hand side of a rule of g is accessible, the all the symbols in the right-hand side of the same rule are also accessible.
  $\forall\, g, n, right,$
  $accessible\ g\ n \rightarrow rules\ g\ n\ right \rightarrow$
  $\forall\, s,$
  $In\ s\ right \rightarrow accessible\ g\ s$

- `produces_g_g_acc`:
  Every sentence produced by g is also produced by $(g\_acc\ g)$.
  $\forall\, g, s,$
  $produces\ g\ s \rightarrow produces\ (g\_acc\ g)\ s$

- `g_equiv_acc`:
  Grammars g and $(g\_acc\ g)$ generate exactly the same language.
  $\forall\, g,$
  $g\_equiv\ (g\_acc\ g)\ g$

- `g_acc_has_no_inaccessible_symbols`:
  Grammar $(g\_acc\ g)$ is free of inaccessible symbols.
  $\forall\, g,$
  $has\_no\_inaccessible\_symbols\ (g\_acc\ g)$

- `g_acc_correct`:
  Every context-free grammar is in relation to another grammar that generates the same language and is free of inaccessible symbols.
  $\forall\, g,$
  $g\_equiv\ (g\_acc\ g)\ g \wedge$
  $has\_no\_inaccessible\_symbols\ (g\_acc\ g)$

C.8 Library “simplification”

Unification of the individual grammar simplification strategies of the libraries “emptyrules”, “unitrules”, “useless” and “inaccessible” presented before. The final objective is to prove that every context-free grammar that generates a non-empty language is also generated by an equivalent grammar which is simultaneously free of empty rules, unit rules, useless symbols and inaccessible symbols. In the case that the original grammar generates the empty string, then a single empty rule is allowed in the new grammar. Also, the start symbol of the new grammar does not appear in the right-hand side of any rule. This final result is expressed in lemma `g_simpl_ex_v1`.

C.8.1 Lemmas

- `no_useless_symbols`:

  Every context-free grammar that generates a non-empty language is also generated by a grammar which is free of useless symbols.

  $\forall\, g,$

  $non\_empty\ g \rightarrow$

  $\exists\, g',$

  $g\_equiv\ g'\ g \wedge$

  $has\_no\_useless\_symbols\ g'$

- `no_inaccessible_symbols`:

  Every context-free grammar generates a language that is also generated by a grammar which is free of useless symbols.

  $\forall\, g,$

  $\exists\, g',$

  $g\_equiv\ g'\ g \wedge$

  $has\_no\_inaccessible\_symbols\ g'$

- `no_unit_rules`:

  Every context-free grammar generates a language that is also generated by a grammar which is free of unit rules.

  $\forall\, g,$

  $\exists g',$

  $g\_equiv\ g'\ g \wedge$

  $has\_no\_unit\_rules\ g'$

- `no_empty_rules`:

  Every context-free grammar generates a language that is also generated by a grammar which is free of empty rules. If the language contains the empty string, then one single empty rule is allowed in the new grammar. Also, the start symbol of the new grammar does not appear in the right-hand side of any rule.

  $\forall\, g,$

  $\exists\, g',$

  $g\_equiv\ g'\ g \wedge$

  $(generates\_empty\ g \rightarrow has\_one\_empty\_rule\ g') \wedge$

  $(\neg\ generates\_empty\ g \rightarrow has\_no\_empty\_rules\ g') \wedge$

  $start\_symbol\_not\_in\_rhs\ g'$

- `g_acc_preserves_use`:

  If a symbol is useful and accessible in a grammar g, then it is also useful in grammar *g_acc g* (the equivalent grammar which is free of inaccessible symbols).

$\forall\, g, s,$
$useful\ g\ s \rightarrow$
$accessible\ g\ s \rightarrow$
$useful\ (g\_acc\ g)\ s$

- `acc_appears`:
  In a grammar *g* that generates a non-empty language, an accessible symbol must appear either in the left- or right-hand side of some rule.
  $\forall\, g, n,$
  $useful\ g\ (start\_symbol\ g) \rightarrow$
  $accessible\ g\ n \rightarrow$
  $appears\ g\ n$

- `in_g_use_acc_is_use`:
  In a grammar *g* that generates a non-empty language, if a symbol remains accesible after all useless symbols have have eliminated, then this symbol is also useful.
  $\forall\, g, n,$
  $useful\ g\ (start\_symbol\ g) \rightarrow$
  $accessible\ (g\_use\ g)\ n \rightarrow$
  $useful\ (g\_use\ g)\ n$

- `no_useless_no_inaccessible_symbols_v1`:
  In a grammar *g* that generates a non-empty language, the elimination of useless symbols followed by the elimination of inaccessible symbols leads toa grammar that generates the same language and is free of inaccessible and useless symbols.
  $\forall\, g,$
  $non\_empty\ g \rightarrow$
  $g\_equiv\ (g\_acc\ (g\_use\ g))\ g \wedge$
  $has\_no\_inaccessible\_symbols\ (g\_acc\ (g\_use\ g)) \wedge$
  $has\_no\_useless\_symbols\ (g\_acc\ (g\_use\ g))$

- `no_useless_no_inaccessible_symbols_v2`:
  A variant of `no_useless_no_inaccessible_symbols_v1`. Every context-free grammar generates a language that can also be generated by a gramar which is free of inaccessible and useless symbols.
  $\forall\, g,$
  $non\_empty\ g \rightarrow$
  $\exists\, g',$
  $g\_equiv\ g'\ g \wedge$

$has\_no\_inaccessible\_symbols\ g' \land$
$has\_no\_useless\_symbols\ g'$

- `New_ss_not_in_unit_v1`:
  In a grammar where all empty rules have been eliminated, no symbol is in unit relation with the start symbol of the grammar.
  $\forall\ g, n,$
  $\neg\ unit (g\_emp'\ g)\ n\ (start\_symbol\ (g\_emp'\ g))$

- `New_ss_not_in_unit_v2`:
  A variant of `New_ss_not_in_unit_v1`. If a symbol n_1 is in a unit relation with symbol n_2 in a grammar which is free of empty rules, then n_2 cannot be the start symbol of the grammar.
  $\forall\ g, n_1, n_2,$
  $unit\ (g\_emp'\ g)\ n_1\ n_2 \to n_2 \neq (start\_symbol\ (g\_emp'\ g))$

- `g_unit_preserves_one_empty_rule`:
  If grammar g has a single empty rule, so does the grammar which is free of unit rules ($g\_unit\ g$).
  $\forall\ g, n_1, n_2,$
  $unit\ g\ n_1\ n_2 \to n_2 \neq (start\_symbol\ g) \to$
  $has\_one\_empty\_rule\ g \to$
  $has\_one\_empty\_rule\ (g\_unit\ g)$

- `g_unit_preserves_no_empty_rules`:
  If grammar g has no empty rules, so does the grammar which is free of unit rules ($g\_unit\ g$).
  $\forall\ g,$
  $has\_no\_empty\_rules\ g \to$
  $has\_no\_empty\_rules\ (g\_unit\ g)$

- `no_empty_no_unit_rules_v1`:
  The elimination of empty rules, followed by the elimination of unit rules leads to a new grammar which is equivalent to the original one. Also, the new grammar will have a single empty rule or no empty rules at all, depending on whether the language contains the empty string or not. The new grammar is free of unit rules as well.
  $\forall\ g,$
  $g\_equiv\ (g\_unit\ (g\_emp'\ g))\ g \land$
  $(generates\_empty\ g \to has\_one\_empty\_rule\ (g\_unit\ (g\_emp'\ g))) \land$
  $(\neg\ generates\_empty\ g \to has\_no\_empty\_rules\ (g\_unit\ (g\_emp'\ g))) \land$
  $has\_no\_unit\_rules\ (g\_unit\ (g\_emp'\ g))$

- `no_empty_no_unit_rules_v2`:
  A variant of `no_empty_no_unit_rules_v1`. The elimination of empty rules (despite the fact that the language might contain the empty string) followed by the elimination of unit rules leads to a new grammar which generates the same language as the original grammar (except for the empty string) and is free of empty and unit rules.
  $\forall\, g,$
  $g\_equiv\_without\_empty\ (g\_unit\ (g\_emp\ g))\ g \wedge$
  $has\_no\_empty\_rules\ (g\_unit\ (g\_emp\ g)) \wedge$
  $has\_no\_unit\_rules\ (g\_unit\ (g\_emp\ g))$

- `no_empty_no_unit_rules_v3`:
  A variant of `no_empty_no_unit_rules_v2`. Every context-free grammar generates a languaga that can be generated by a grammar which is free of empty and unit rules. In the case that the language contains the empty string, the new grammar will have a single empty rule.
  $\forall\, g,$
  $\exists\, g',$
  $g\_equiv\ g'\ g \wedge$
  $(generates\_empty\ g \rightarrow has\_one\_empty\_rule\ g') \wedge$
  $(\neg\ generates\_empty\ g \rightarrow has\_no\_empty\_rules\ g') \wedge$
  $has\_no\_unit\_rules\ g'$

- `g_acc_g_use_preserves_rules`:
  In a grammar where the useless and inaccessible symbols have been eliminated, the remaining rules area also rules of the original grammar.
  $\forall\, g, left, right,$
  $rules\ (g\_acc\ (g\_use\ g))\ left\ right \rightarrow rules\ g\ left\ right$

- `g_acc_g_use_preserves_empty_rule`:
  If a grammar has an empty rule with the start symbol in the left-hand side, so does the equivalent grammar which is free of useless and inaccessible symbols.
  $\forall\, g,$
  $rules\ g\ (start\_symbol\ g)\ \varepsilon \rightarrow$
  $rules\ (g\_acc\ (g\_use\ g))\ (start\_symbol\ (g\_acc\ (g\_use\ g)))\ \varepsilon$

- `g_emp_preserves_non_empty`:
  If a grammar generates a language that contains a sentence other than the empty string, then the language generated by ($g\_emp\ g$) is non-empty.
  $\forall\, g,$

$(\exists\, s, produces\ g\ s \land s \neq \varepsilon) \rightarrow$
$non\_empty\ (g\_emp\ g)$

- `g_emp'_preserves_non_empty`:
  If a grammar generates a language that contains at least one sentence, then the language generated by $(g\_emp'\ g)$ also contains at least one sentence.
  $\forall\, g,$
  $non\_empty\ g \rightarrow$
  $non\_empty\ (g\_emp'\ g)$

- `g_unit_preserves_non_empty`:
  If a grammar generates a non-empty language, so does the grammar that is equivalent to this one and is free of unit rules.
  $\forall\, g,$
  $non\_empty\ g \rightarrow$
  $non\_empty\ (g\_unit\ g)$

- `g_unit_preserves_start`:
  If a grammar generates does not have the start symbol in the right-hand side of any rule, so does the grammar that is equivalent to this one and is free of unit rules.
  $\forall\, g,$
  $start\_symbol\_not\_in\_rhs\ g \rightarrow$
  $start\_symbol\_not\_in\_rhs\ (g\_unit\ g)$

- `g_use_preserves_start`:
  If a grammar generates does not have the start symbol in the right-hand side of any rule, so does the grammar that is equivalent to this one and is free of useless symbols.
  $\forall\, g,$
  $start\_symbol\_not\_in\_rhs\ g \rightarrow$
  $start\_symbol\_not\_in\_rhs\ (g\_use\ g)$

- `g_acc_preserves_start`:
  If a grammar generates does not have the start symbol in the right-hand side of any rule, so does the grammar that is equivalent to this one and is free of inaccessible symbols.
  $\forall\, g,$
  $start\_symbol\_not\_in\_rhs\ g \rightarrow$
  $start\_symbol\_not\_in\_rhs\ (g\_acc\ g)$

- `g_simpl_ex_v1`:
  Every context-free grammar that generates a non-empty language can also be generated by an equivalent grammar that is simultaneously free of useless symbols,

inaccessible symbols, empty rules and unit rules. In the case that the language contains the empty string, then a single empty rule is admitted. Finally, the start symbol of the new grammar does appear not in the right-hand side of any rule.
$\forall\, g,$
$non\_empty\ g \rightarrow$
$\exists\, g',$
$g\_equiv\ g'\ g \wedge$
$has\_no\_inaccessible\_symbols\ g' \wedge$
$has\_no\_useless\_symbols\ g' \wedge$
$(produces\_empty\ g \rightarrow$
$has\_one\_empty\_rule\ g') \wedge$
$(\neg\ produces\_empty\ g \rightarrow has\_no\_empty\_rules\ g') \wedge$
$has\_no\_unit\_rules\ g' \wedge$
$start\_symbol\_not\_in\_rhs\ g'$

- `g_simpl_ex_v2`:
  A variant of `g_simpl_ex_v1`. If a grammar generates a language that contains at least one non-empty string, then this language (except for the empty string) can also be generated by a new grammar which is simultaneously free of inaccessible symbols, useless symbols, empty rules and unit rules. Also, the start symbol of the new grammar does not appear in the right-hand side of any rule.
  $\forall\, g,$
  $(\exists\, s, produces\ g\ s \wedge s \neq \varepsilon) \rightarrow$
  $\exists\, g',$
  $g\_equiv\_without\_empty\ g'\ g \wedge$
  $has\_no\_inaccessible\_symbols\ g' \wedge$
  $has\_no\_useless\_symbols\ g' \wedge$
  $has\_no\_empty\_rules\ g' \wedge$
  $has\_no\_unit\_rules\ g' \wedge$
  $start\_symbol\_not\_in\_rhs\ g'$

C.9 Library “chomsky”

Normalization of context-free grammars with respect to the Chomsky Normal Form. The final objective of this library is to prove that every context-free grammar generates a language that can also be generated by an equivalente grammar which is in the Chomsky Normal Form (with a special empty rule in the case that the language contains the empty string). Also, the start symbol of the CNF grammar does not appear in the right-hand side of any rule. This result is expressed in lemma `g_cnf_ex`.

C.9.1 Definitions

- A rule $N \rightarrow \beta$ such that β consists of a single terminal symbol or two non-terminal symbols:

```
Definition is_cnf_rule (left: non_terminal) (right: sf): Prop:=
(∃ s1 s2: non_terminal, right = [inl s1; inl s2]) ∨
(∃ t: terminal, right = [inr t]).
```

- A CNF grammar is a grammar whose rules satisfy `is_cnf_rule`:

```
Definition is_cnf (g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right → is_cnf_rule left right.
```

- A CNF grammar with a single empty rule:

```
Definition is_cnf_with_empty_rule
(g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right →
(left = (start_symbol g) ∧ right = []) ∨
is_cnf_rule left right.
```

- The type of the non-terminal symbols of a CNF grammar. It states that any string of terminal and/or non-terminal symbols can be used to characterize a new non-terminal symbol:

```
Inductive non_terminal': Type:=
| Lift_r: sf → non_terminal'.
```

- The rules of a CNF grammar (without the empty rule):

```
Inductive g_cnf_rules (g: cfg non_terminal terminal):
non_terminal' → sf' → Prop:=
| Lift_cnf_t: ∀ t: terminal,
              ∀ left: non_terminal,
              ∀ s1 s2: sf,
              rules g left (s1++[inr t]++s2) →
              g_cnf_rules g (Lift_r [inr t]) [inr t]
| Lift_cnf_1: ∀ left: non_terminal,
              ∀ t: terminal,
              rules g left [inr t] →
```

```
                  g_cnf_rules g (Lift_r [inl left]) [inr t]
| Lift_cnf_2: ∀ left: non_terminal,
                  ∀ s1 s2: symbol,
                  ∀ beta: sf,
                  rules g left (s1 :: s2 :: beta) →
                  g_cnf_rules g (Lift_r [inl left])
                            [inl (Lift_r [s1]);
                            inl (Lift_r (s2 :: beta))]
| Lift_cnf_3: ∀ left: sf,
                  ∀ s1 s2 s3: symbol,
                  g_cnf_rules g (Lift_r left) [inl (Lift_r [s1]);
                            inl (Lift_r (s2 :: s3 :: beta))] →
                  g_cnf_rules g (Lift_r (s2 :: s3 :: beta))
              [inl (Lift_r [s2]); inl (Lift_r (s3 :: beta))].
```

- New CNF grammar (without the empty rule):

```
 Definition g_cnf (g: cfg non_terminal terminal):
cfg non_terminal' terminal := {|
 start_symbol:= Lift_r [inl (start_symbol g)];
 rules:= g_cnf_rules g;
 rules_finite:= g_cnf_finite g
 |}.
```

- The rules of a CNF grammar (with a single empty rule and based in `g_cnf_rules`):

```
 Inductive g_cnf'_rules
(g: cfg non_terminal terminal): non_terminal' → sf' → Prop:=
| Lift_cnf'_all:
       ∀ left: non_terminal',
       ∀ right: sf',
       g_cnf_rules g left right →
       g_cnf'_rules g left right
| Lift_cnf'_new:
       g_cnf'_rules g (start_symbol (g_cnf g)) [].
```

- New CNF grammar (with a single empty rule):

```
 Definition g_cnf'
 (g: cfg non_terminal terminal):
cfg non_terminal' terminal:= {|
 start_symbol:= start_symbol (g_cnf g);
 rules:= g_cnf'_rules g;
 rules_finite:= g_cnf'_finite g
```

|}.

C.9.2 Lemmas

- `rules_g_cnf_to_rules_g_right`:
  From the right-hand side of a rule of ($g\_cnf\ g$) it is possible to assert the existence of a rule of g.
  $\forall\, g, right_2, n_1, n_2, s_1, s_2,$
  $g\_cnf\_rules\ g\ n_1\ (n_2 \cdot (Lift\_r\ (s_1 \cdot s_2 \cdot right_2))) \to$
  $\exists\, left, right_1,$
  $rules\ g\ left\ (right_1 \cdot s_1 \cdot s_2 \cdot right_2)$

- `rules_g_cnf_to_rules_g_left`:
  From the left-hand side of a rule of ($g\_cnf\ g$) it is possible to assert the existence of a rule of g.
  $\forall\, g, s_1, s_2,$
  $g\_cnf\_rules\ g\ (Lift\_r\ s_1)\ s_2 \to$
  $(\exists\, t, s_1 = t \wedge \exists\, left, s_1, s_2, rules\ g\ left\ (s_1 \cdot t \cdot s_2)) \vee$
  $(\exists\, n, s_1 = n \wedge \exists\, right, rules\ g\ n\ right) \vee$
  $(\exists\, left, right, s_3, s_4, rules\ g\ left\ right \wedge right = s_3 \cdot s_1 \cdot s_4)$

- `derives_g_cnf_g`:
  A sentence derived by ($g\_cnf\ g$) is also derived by g.
  $\forall\, g, s,$
  $derives\ (g\_cnf\ g)\ (start\_symbol\ (g\_cnf\ g))\ s \to$
  $derives\ g\ (start\_symbol\ g)\ s$

- `rules_g_derives_g_cnf`:
  The right-hand side of a rule of a grammar g is derivable from the corresponding left-hand side in ($g\_cnf\ g$).
  $\forall\, g, left, right,$
  $has\_no\_unit\_rules\ g \to$
  $right \neq \varepsilon \to$
  $rules\ g\ left\ right \to$
  $derives\ (g\_cnf\ g)\ (Lift\_r\ left)\ right$

- `derives_g_g_cnf_v1`:
  If a grammar g derives sentence s, then s must be derivable in ($g\_cnf\ g$).
  $\forall\, g, s,$
  $has\_no\_unit\_rules\ g \to$
  $has\_no\_empty\_rules\ g \to$

$derives\ g\ (start\_symbol\ g)\ s \rightarrow$
$derives\ (g\_cnf\ g)\ (start\_symbol\ (g\_cnf\ g))\ s$

- `derives_g_g_cnf_v2`:
  A variant of `derives_g_g_cnf_v1` in the case that *g* contains empty rules.
  $\forall\ g, s,$
  $has\_no\_unit\_rules\ g \rightarrow$
  $has\_one\_empty\_rule\ g \rightarrow$
  $s \neq \varepsilon \rightarrow$
  $start\_symbol\_not\_in\_rhs\ g \rightarrow$
  $derives\ g\ (start\_symbol\ g)\ s \rightarrow$
  $derives\ (g\_cnf\ g)\ (start\_symbol\ (g\_cnf\ g))\ s$

- `g_cnf_preserves_start`:
  If the start symbol of *g* does not appear in the right-hand side of any rule of *g*, then it also does not appear in the right-hand side of any rule of (*g_cnf g*).
  $\forall g,$
  $start\_symbol\_not\_in\_rhs\ g \rightarrow$
  $start\_symbol\_not\_in\_rhs\ (g\_cnf\ g)$

- `g_cnf'_preserves_start`:
  If the start symbol of *g* does not appear in the right-hand side of any rule of *g*, then it also does not appear in the right-hand side of any rule of (*g_cnf′ g*).
  $\forall g,$
  $start\_symbol\_not\_in\_rhs\ g \rightarrow$
  $start\_symbol\_not\_in\_rhs\ (g\_cnf'\ g)$

- `g_cnf_correct_v1`:
  Correctness of (*g_cnf g*) in the case that *g* does not contain empty rules.
  $\forall\ g,$
  $has\_no\_unit\_rules\ g \wedge$
  $has\_no\_empty\_rules\ g \wedge$
  $start\_symbol\_not\_in\_rhs\ g \rightarrow$
  $is\_cnf\ (g\_cnf\ g) \wedge$
  $g\_equiv\_without\_empty\ (g\_cnf\ g)\ g \wedge$
  $start\_symbol\_not\_in\_rhs\ (g\_cnf\ g)$

- `g_cnf_correct_v2`:
  Correctness of (*g_cnf g*) in the case that *g* has an empty rule.
  $\forall\ g,$
  $has\_no\_unit\_rules\ g \wedge$

$has\_one\_empty\_rule\ g \wedge$
$start\_symbol\_not\_in\_rhs\ g \rightarrow$
$is\_cnf\ (g\_cnf\ g) \wedge$
$g\_equiv\_without\_empty\ (g\_cnf\ g)\ g \wedge$
$start\_symbol\_not\_in\_rhs\ (g\_cnf\ g)$

- `g_cnf'_correct`:
  Correctness of g_cnf'.
  $\forall\ g,$
  $has\_no\_unit\_rules\ g \wedge$
  $has\_no\_empty\_rules\ g \wedge$
  $start\_symbol\_not\_in\_rhs\ g \rightarrow$
  $is\_cnf\_with\_empty\_rule\ (g\_cnf'\ g) \wedge$
  $g\_equiv\_without\_empty\ (g\_cnf'\ g) g \wedge$
  $start\_symbol\_not\_in\_rhs\ (g\_cnf'\ g)$

- `g_cnf_ex`:
  Asserts the existence of a CNF gramar that is equivalent to the original grammar. Also, that the CNF does not contain the start symbol in the right-hand side of any rule.
  $\forall\ g, \exists\ g',$
  $g\_equiv\ g'\ g \wedge (is\_cnf\ g' \vee is\_cnf\_with\_empty\_rule\ g') \wedge start\_symbol\_not\_in\_rhs\ g'$

C.10 Library "pumping"

Pumping Lemma for context-free languages. The objective of this library is to prove the Pumping Lemma for context-free languages, a result that is expressed in `pumping_lemma`.

C.10.1 Lemmas

- `pumping_aux`:
  If a binary tree that represents a derivation in a CNF grammar has a subtree with the same root as the main tree, then the pumping of an internal subtree generates a corresponding pumped frontier and internal path. This lemma is auxiliary in the proof of the `pumping_lemma` presented next.
  $\forall\ g, t_1, t_2, n, c_1, c_2, v, x,$
  $btree\_decompose\ t_1\ c_1 = Some\ (v, t_2, x) \rightarrow$
  $btree\_cnf\ g\ t_1 \rightarrow$
  $broot\ t_1 = n \rightarrow$
  $bcode\ t_1\ (c_1 \cdot c_2) \rightarrow$

$c_1 \neq \varepsilon \rightarrow$
$broot\ t_2 = n \rightarrow$
$bcode\ t_2\ c_2 \rightarrow$
$(\forall\ i, \exists\ t',$
$btree\_cnf\ g\ t' \wedge$
$broot\ t' = n\ \wedge$
$btree\_decompose\ t'\ c_1^i = Some\ (v^i, t_2, x^i) \wedge$
$bcode\ t'\ (c_1^i \cdot c_2) \wedge$
$get\_nt\_btree\ c_1^i\ t' = Some\ n)$

- `pumping_lemma`:
  Pumping Lemma for Context-Free Languages (uses $n = 2^k$).
  $\forall\ l,$
  $cfl\ l \rightarrow$
  $\exists\ n, \forall\ s,$
  $s \in l \rightarrow$
  $|s| \geq n \rightarrow$
  $\exists\ uvwxy,$
  $s = u \cdot v \cdot w \cdot x \cdot y \wedge$
  $|v \cdot x| \geq 1 \wedge$
  $|u \cdot y| \geq 1 \wedge$
  $|v \cdot w \cdot x| \leq n \wedge$
  $\forall\ i, u \cdot v^i \cdot w \cdot x^i \cdot y \in l$

- `pumping_lemma_v2`:
  A variant of `pumping_lemma` that uses $n = 2^{k-1} + 1$ instead of $n = 2^k$.
  $\forall\ l,$
  $cfl\ l \rightarrow$
  $\exists\ n, \forall\ s,$
  $s \in l \rightarrow$
  $|s| \geq n \rightarrow$
  $\exists\ uvwxy,$
  $s = u \cdot v \cdot w \cdot x \cdot y \wedge$
  $|v \cdot x| \geq 1 \wedge$
  $|v \cdot w \cdot x| \leq (n-1) * 2 \wedge$
  $\forall\ i, u \cdot v^i \cdot w \cdot x^i \cdot y \in l$