Développement du système MathNat pour la formalisation automatique des textes mathématiques

Humayoun Muhammad

To cite this version:

Humayoun Muhammad. Développement du système MathNat pour la formalisation automatique des textes mathématiques. Mathématiques générales [math.GM]. Université de Grenoble, 2012. Français. NNT : 2012GRENM001 . tel-00680095

HAL Id: tel-00680095
https://theses.hal.science/tel-00680095

Submitted on 17 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR

de l'Université de Grenoble

Spécialité: Mathématiques et Informatiques

Présentée et soutenue publiquement par

MUHAMMAD HUMAYOUN

Le 18 janvier 2012

TITRE:

Developing the System MathNat
for Automatic Formalization of Mathematical texts

Thèse dirigée par **Christophe Raffalli** et **Aarne Ranta**

Préparée au sein du **LAMA, Université de Savoie** et de **l'École Doctorale MSTII**

JURY

Civilité/Nom/Prénom	**Fonction et lieu de la fonction**	**Rôle**
M Loïc POTTIER	HDR, Chargé de recherche, INRIA Sophia Antipolis	Rapporteur
M Zhaohui LUO	Professeur, University of London	Rapporteur
M Laurent VUILLON	Professeur, LAMA, Université de Savoie	Examinateur
Mme Laurence DANLOS	Professeur, Université de Paris 7	Présidente du jury
M Christophe RAFFALLI	Maître de Conférences, LAMA, Université de Savoie	Directeur de thèse
M Aarne RANTA	Professeur, University of Gothenburg	Directeur de thèse

UNIVERSITÉ DE GRENOBLE

UNIVERSITY OF GRENOBLE
PHD THESIS

to obtain the title of

Doctor of Science

of the University of Grenoble

Specialization: Mathematics and Computer Science

Defended by

MUHAMMAD HUMAYOUN

18 January 2012

TITLE:

Developing the System MathNat
for Automatic Formalization of Mathematical texts

PhD Thesis under the direction of **Christophe Raffalli** and **Aarne Ranta**

Prepared at **LAMA, University of Savoie** and the **Doctoral School MSTII**

JURY

Mr. Loïc Pottier	HDR, Research Associate, INRIA Sophia Antipolis	Reviewer
Mr. Zhaohui Luo	Professor, University of London	Reviewer
Mr. Laurent Vuillon	Professor, LAMA, University of Savoie	Examiner
Ms. Laurence Danlos	Professor, Université de Paris 7	President
Mr. Christophe Raffalli	Associate Professor, LAMA, University of Savoie	Thesis Director
Mr. Aarne Ranta	Professor, University of Gothenburg	Thesis Director

Acknowledgments

All Praise be to Allah, Lord of the Worlds.
May Allah exalt and bring peace upon Muhammad.

First and foremost, I would like to express my deepest gratitude, appreciation and sincere thanks to my PhD supervisors Dr. Christophe Raffalli and Prof. Aarne Ranta, for their excellent guidance, constant help and dedicated support. While I remain the only responsible of omissions and mistakes, this thesis literally could not be finished without their amazing supervision, continuous encouragement and a lot of proofreading.

I could not have had a better supervisor than Dr. Christophe Raffalli. He has been a constant inspiration in all these years. He was always very patient with my mistakes, ramblings and slow progress. His pleasant and spirited personality, cheerful attitude and friendliness always made me comfortable to admit my shortcomings and mistakes, which in return, helped me to improve this work under his resourceful guidance.

Other than our daily scientific discussions and meetings, I enjoyed his company during coffee breaks, lunches, and several barbecue events and dinners which were hosted by him at his home. Above all, I cannot forget that memorable week when he hosted me in his home for a week on my arrival to France. I feel greatly obliged for the hospitality of his wife Rebecca, and for the company of their charming children. Because of Dr. Raffalli and his family, France became a second home to me.

I am also greatly inspired by Prof. Aarne Ranta, especially by his wisdom and approach towards research and life. It was his course on 'natural language technology' at Chalmers University of Technology, which introduced me to this language technology business. Later, he accepted to become supervisor for my Master thesis, and warmly involved me in the activities of the language technology group at Chalmers. It was my first encounter with research and Prof. Ranta is a perfect role model to follow. His friendly attitude and constant guidance shaped my decision of doing a PhD. I am very lucky to have Prof. Ranta as my second supervisor in this PhD. His support, guidance and stimulating ideas proved to be a very valuable asset for me. During this PhD, I visited Chalmers for about six months, where he helped me a lot to improve my work. I am also thankful to my colleagues at the language technology group in department of computer science, Chalmers University of Technology.

I am especially in-debt to my dear friend Dr. Harald Hammarström who spent much time and effort proofreading my thesis draft. He was also involved in my Master thesis as a mentor. I have learned a lot from his very friendly yet "no-nonsense" attitude for studies and research.

I am grateful to Prof. Zhaohui Luo and Prof. Loïc pottier, the referees of this dissertation, for their careful reading, and insightful comments and suggestions. Any mistake left is surely my fault. I also would like to pay my gratitude to the whole jury for kindly accepting the invitation.

I would like to pay my gratitude to my mentors at XEROX research centre France, Mr. Victor Ciriza and Dr. Jean-Marc Andreoli for their kindness and very good supervision during my internship. I'v learned a lot about the software development and

data mining through the numerous meetings and discussions with Mr. Ciriza. He is an excellent mentor.

Many thanks go to my colleagues at the department of mathematics (LAMA), Université de Savoie, for the pleasant and creative atmosphere. I also want to thank all the secretaries (especially Mme Nadine Mari), for their help in administrative issues.

I would like to thank my office mate Florian Hatat for his sincere friendship and nice discussions. I also would like to thank my office mate and *partner in crimes* Mohamad Ziadeh, for his brotherly friendship, delicious food and constant harassment. I am blessed by having a lot of very good and sincere friends. Instead of naming them one by one, I express my sincere gratitude for there memorable friendship.

I would like to pay my deepest gratitude and love to my parents. I dedicate this thesis to them. They choose a hard life, so that we the children, may have a comfortable life and good education. All my success is due to their sacrifices and continuous prayers. Thank you both, for loving me, teaching me the ABC of life and encouraging me always. I am also grateful to my brother and sister, for their prayers and love.

I am greatly in-debt to the parents, and the brothers and sisters of my wife, for their love, support and prayers. They, especially, my mother in-law took care of my children time after time. Also many thanks to my sister in-law Rumaisa, who proofread an early version of my dissertation.

My dear wife Adeela, I couldn't have managed this without you. Thanks a lot for your love, comfort, support, encouragement and patience over the last few years. Thanks for being a wonderful companion. I love you with all my heart!

Ali and Warda, my dear children, you are the most precious gift from the heavens. You and your mother's presence keeps me happy and content.

Muhammad Humayoun
Le Bourget du Lac, France.
13 January 2012

This work is funded by "Informatique, Signal, Logiciel Embarqué" (ISLE), Rhone-Alpes, France.
http://cluster-isle.grenoble-inp.fr.

Abstract

There is a wide gap between the language of mathematics and its formalized versions. The term "language of mathematics" or "mathematical language" refers to prose that the mathematician uses in authoring textbooks and publications. It mainly consists of natural language, symbolic expressions and notations. It is flexible, structured and semantically well-understood by mathematicians.

However, it is very difficult to formalize it automatically. Some of the main reasons are: complex and rich linguistic features of natural language and its inherent ambiguity; intermixing of natural language with symbolic mathematics causing problems which are unique of its kind, and therefore, posing more ambiguity; and the possibility of containing reasoning gaps, which are hard to fill using the current state of art theorem provers (both automated and interactive).

One way to work around this problem is to abandon the use of the language of mathematics. Therefore in current state of art of theorem proving, mathematics is formalized manually in very precise, specific and well-defined logical systems. The languages supported by these systems impose strong restrictions. For instance, these languages have non-ambiguous syntax with a limited number of possible syntactic constructions.

This enterprise divides the world of mathematics in two groups. The first group consists of a vast majority of mathematicians whose rely on the language of mathematics only. In contrast, the second group consists of a minority of mathematicians. They use formal systems such as theorem provers (interactive ones mostly) in addition to the language of mathematics.

To bridge the gap between the language of mathematics and its formalized versions, we may ask the following gigantic question:

> Can we build a program that understands the language of mathematics used by mathematicians and can we mechanically verify its correctness?

This problem can naturally be divided in two sub-problems, both very hard:

1. Parsing mathematical texts (mainly proofs) and translating those parse trees to a formal language after resolving linguistic issues.

2. Validation of this formal version of mathematics.

The project MathNat (**Math**ematics in controlled **Nat**ural language) aims at being the first step towards solving this problem, focusing mainly on the first question.

For that, first, we develop a **C**ontrolled **L**anguage for **M**athematics (CLM) which is a precisely defined subset of English with restricted grammar and lexicon. To make CLM natural and expressive, we support important linguistic features such as anaphoric pronouns and references, rephrasing of a sentence in multiple ways, the proper handling of distributive and collective readings and so on. The coverage of CLM at the moment is yet rather small and to be improved as the project keeps evolving in future.

Second, we develop MathAbs (**Math**ematical **Abs**tract language). It is a prover independent formal language to represent the semantics of CLM texts preserving its logical and reasoning structure. MathAbs is designed as an intermediate language between CLM and the formal languages of theorem provers, allowing proof checking.

Third, we propose a system that can automatically translate CLM to MathAbs, giving a precise semantics to CLM. We consider that formalizing mathematics automatically in such a formal language that has a precise semantics is an important progress even if it can't always be proof-checked.

This brings us to the second question for which we report a very limited work. We only translate MathAbs to the first-order formulas. If we feed these formulas to the automated theorem provers (ATPs), then fundamentally the ATPs should be able to validate them sometimes. In other words, the resulting MathAbs document is not completely verifiable for the moment, but it represents an opportunity for the mathematician to write mathematical text (mainly proofs) without becoming expert of any theorem prover.

Keywords: Computational linguistics, Language technology, Controlled languages, The language of mathematics, Formalization, Formal systems, Validation, Proof checking.

List of Figures

2.1 A typical theorem and its proof 6
2.2 MathNat Architecture in a Nutshell 8
2.3 Proof tree of theorem and proof given in figure 2.1. 13
2.4 Three CLM processable versions of figure 2.1. 14
2.5 The First theorem and its proof of figure 2.4 as verbatim. 15
2.6 Context-free grammar for simple propositions 18
2.7 Abstract Syntax for simple propositions 19
2.8 Concrete syntax for abstract syntax given in figure 2.7 20

3.1 Some axioms from mathematical texts 32
3.2 Some definitions from mathematical texts 33
3.3 Some theorems and their proofs from mathematical texts 34
3.4 Mathematical texts showing proposition, lemma, corollary and their proofs (when given) 35

4.1 A typical text from elementary number theory and its MathAbs 61
4.2 Semantics of the proof in figure 4.1 as a proof tree. 62
4.3 The rules of Gentzen's natural deduction 69
4.4 A proof by case and its MathAbs explaining existential quantifier. . . . 77

5.1 Context-free grammar for simple propositions 94
5.2 Abstract Syntax for simple propositions 94
5.3 Concrete Syntax for simple propositions 95
5.4 Abstract syntax trees (AST) of two simple propositions 96

6.1 ASCII Symbol List for Symbolic Mathematics. 106
6.2 Labelled BNF grammar for symbolic mathematics. 109

8.1 CLM grammar to MathAbs 218

9.1 Concrete syntax for French for the abstract syntax given in figure 2.7 . . 238

Contents

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Contributions and Outline . . . 3
1.2.1 The Controlled language of Mathematics (CLM) . . . 3
1.2.2 MathAbs - The Mathematical Abstract Language . . . 3
1.2.3 The Host System MathNat . . . 4
1.2.4 Proof Checking . . . 4

2 The MathNat Framework **5**
2.1 Introduction . . . 5
2.2 Analysis of the Language of Mathematics . . . 5
2.3 Semantics (Mathematical Abstract language: MathAbs) . . . 8
2.4 Syntax (The Controlled Language of Mathematics) . . . 12
2.5 Defining CLM . . . 17
2.5.1 Sentence Level Grammar . . . 17
2.5.2 Restoring Discourse . . . 19
2.6 Related Work . . . 23

3 The Language of Mathematics **29**
3.1 Introduction . . . 29
3.2 Mathematical Discourse . . . 31
3.2.1 Axiom . . . 31
3.2.2 Definition . . . 31
3.2.3 Theorem . . . 33
3.2.4 Proof . . . 35
3.2.5 Remark, Example, Exercise . . . 39
3.3 Micro Structure of the Language of Mathematics . . . 40
3.3.1 Intermixing of Text and Symbols . . . 40
3.3.2 Ambiguity . . . 42
3.3.3 Statements and Logical Operators . . . 44
3.3.4 Anaphoric Pronouns . . . 49
3.3.5 Anaphoric References . . . 52
3.3.6 Miscellaneous Features . . . 54
3.4 List of Mathematical texts used for analysis . . . 56

4 MathAbs - The Abstract Mathematical Language **57**
4.1 Introduction . . . 57
4.2 Why MathAbs . . . 58
4.2.1 Origin of MathAbs . . . 59
4.3 Syntax of MathAbs . . . 59
4.3.1 MathAbs and Extensibility . . . 64
4.4 The MathAbs Semantics . . . 64
4.5 Completeness . . . 68
4.6 Translating Textual Quantifiers to MathAbs . . . 72

4.6.1 Simplifying MathAbs 76
4.7 Proof checking 79
4.7.1 Proof checking using Automated Theorem Provers 79
4.7.2 Proof Checking Limitations and Future Directions 84
4.8 Conclusion 85
4.9 Appendix 85
4.9.1 An Example: 85

5 The Controlled Language for Mathematics 91
5.1 Introduction 91
5.2 Grammatical Framework 92
5.2.1 Getting Started 93
5.3 Overview of CLM 96
5.3.1 Macro Level 96
5.3.2 Micro Level 99

6 Micro Level CLM Grammar 105
6.1 Introduction 105
6.2 Symbolic Mathematics 105
6.2.1 Symbolic Mathematics in GF 110
6.3 Grammar Implementation for Textual Mathematics 113
6.3.1 A Few Low Level Constructs 113
6.3.2 Propositions 124
6.3.3 Existential Statements 136
6.3.4 Relational Statements 140
6.3.5 Equation with a Reference 142
6.3.6 Statements 143
6.3.7 Conditional Statement 146
6.3.8 Take Statement 147
6.3.9 Justifications 147

7 Macro Level CLM Grammar 155
7.1 Introduction 155
7.2 Theorem and its Proof 155
7.2.1 Proof and its Statements 156
7.2.2 Theorem and its Statements 170
7.3 Axiom 173
7.4 Definition 174
7.5 Conventions 176
7.5.1 Ambiguity 177

8 The Host System MathNat 179
8.1 Introduction 179
8.1.1 The Overall Picture 180
8.1.2 The CLM Grammar in Haskell 181
8.2 Semantic Checks 182
8.2.1 List of expressions and Quantity 184
8.2.2 Pronoun and Quantity 185

8.2.3 Collective properties . . . 185
8.2.4 List of expressions or Pronouns with Noun Adjunct . . . 187
8.2.5 Statements with Property Arbitrary . . . 188
8.2.6 Quantity and List of Expressions . . . 188
8.2.7 Expression, Equation and Symbolic Formula . . . 189
8.3 Discourse, Linguistic Features and Others . . . 189
8.3.1 Propositions . . . 191
8.3.2 Equation with Reference . . . 200
8.3.3 Existential Statements . . . 202
8.3.4 Statement and its lists . . . 207
8.3.5 Conditional Statement . . . 208
8.3.6 Justifications . . . 210
8.3.7 The Macro Level Grammar . . . 213
8.3.8 Theorem . . . 213
8.3.9 Proof . . . 217
8.3.10 Theorem and its Proof . . . 220
8.3.11 Case Analysis for Proof by Cases . . . 221
8.3.12 The rest of blocks, categories, statements and rules . . . 232

9 Concluding Remarks 233
9.1 Results . . . 233
9.2 Limitations and Future Directions . . . 234
9.2.1 Ambiguity . . . 234
9.2.2 Proof Checking . . . 237
9.2.3 Multilingual CLM . . . 237
9.2.4 Generation from MathAbs . . . 238

A A Few Selected Examples 239
A.1 Irrationality of $\sqrt{2}$ . . . 239
A.1.1 First Version: . . . 239
A.1.2 Second Version: . . . 242
A.1.3 Third Version: . . . 244
A.1.4 Fourth Version: . . . 246
A.2 Second Example . . . 247
A.3 Third Example . . . 248
A.4 Fourth Example . . . 251
A.4.1 First Version: . . . 251
A.4.2 Second Version: . . . 251
A.5 Fifth Example . . . 252
A.6 Sixth Example . . . 253
A.7 Seventh Example . . . 254
A.8 Eighth Example . . . 254

Bibliography 255

Chapter 1

Introduction

Contents

1.1 Problem Statement **1**

1.2 Contributions and Outline **3**

1.2.1 The Controlled language of Mathematics (CLM) 3

1.2.2 MathAbs - The Mathematical Abstract Language 3

1.2.3 The Host System MathNat 4

1.2.4 Proof Checking 4

1.1 Problem Statement

Natural languages are ambiguous, complex and rich. These characteristics make their formalization a very difficult task. In contrast, mathematics is written in a specific scientific register which is evolved in centuries. However, we are concerned with its most modern version and refer it as the "language of mathematics". It is in fact the prose that mathematicians use to author textbooks and publications. It mainly consists of natural language, symbolic expressions and notations. It is flexible, structured and semantically well-understood by mathematicians.

In principle, formalization of mathematics is possible; for instance in first-order logic, in higher-order logic, or in set theory; using Hilberts' system, natural deduction or sequent calculus for proofs. However, it is very difficult to formalize mathematical texts automatically. Some of the main reasons for this enterprise are:

- Although the natural language used in mathematical texts is simpler and restricted to this particular domain, it still contains complex and rich linguistic features such as the use of anaphoric pronouns and references, rephrasing and paraphrasing, distributive and collective readings which requires proper handling, etc. Furthermore, the mathematical texts could be inherently ambiguous if not written carefully.

- The language of mathematics being a mixture of words and symbols is unique. It poses many unique challenges such as: discovering the kind ($\forall, \exists$) and ordering of quantification of variables, discovering the type of mathematical variables and objects, and solving the precedence of mathematical equations and formulas. Failing to address them would lead to the serious issue of ambiguity.

- To make the mathematical texts comprehensive and aesthetically elegant, mathematicians tend to omit obvious details. Such reasoning gaps may be quite easy for a human to figure out but definitely hard to be filled by the current state of art theorem provers.

- In a similar way, more details given in mathematical texts for the purpose of explanation might be useful for the reader but definitely pose problems for computer assisted theorem provers. For instance, the automated theorem provers are normally too sensitive to such details. It is mainly because of their inability to, first, include what is necessary for proof checking, and second, exclude what is not necessary (cf. §4.7 on page 79 for more details).

These problems convince a multitude of logicians and scientists that if not impossible, automatic formalization of the mathematics in its textual form is far more difficult and problematic. For instance, on the use of the language of mathematics, Frege expressed his dissatisfaction in following words:

> "...I found the inadequacy of language to be an obstacle; no matter how unwieldy the expressions was ready to accept, I was less and less able, as the relations became more and more complex, to attain the precision that my purpose required." ([Frege 1879], Preface).

This leads to the current state of the art in theorem proving, in which the language of mathematics is given up and mathematical texts are sometimes manually formalized in very precise and accurate systems using specific formalisms normally based on some particular calculus or logic. It normally contains a lot of technical details of the underlying formal system, making it not suitable for the human comprehension. Among others, a classic example of such a work is *Principia Mathematica* [Whitehead & Russell 1962]. Although it is one of the first comprehensive works on formal mathematics, it leads to the view that complete formalization is too extensive and perhaps uninteresting for a mathematician. For instance, one of the authors, Russell expressed his experience in the following words:

> [...] my intellect never quite recovered from the strain [of doing this work]. I have been ever since definitely less capable of dealing with difficult abstractions than I was before.
> ([Russell 1998, p. 155]; first published in 1967)

This leads to a bipolar enterprise in which the mathematicians rely upon the language of mathematics; while the logicians use formal systems such as proof assistants and theorem provers (both automated and interactive). This wide gap plagues both fields and reduces the usefulness of computer assisted theorem proving in learning, teaching and formalizing mathematics. In the light of the above discussion, we may ask the following question:

> Can we build a program that understands the language of mathematics used by mathematicians in their published work and can we mechanically verify its correctness?

The project MathNat (**Math**ematics in controlled **Nat**ural language) aims at being a first step towards answering this gigantic question. It requires the handling of two very hard problems:

1. **Automatic Formalization:** Parsing mathematical texts mainly proofs and translating their parse trees to a formal language after resolving linguistic issues. This

formal language should be able to represent mathematics as close as possible to the intentions expressed by the author and must be independent of any logic and prover.

2. **Validation** of this formal version of mathematics.

This thesis attempts to answer the first question, which is related to the formalization of the mathematical texts. The second question, which addresses validation, is answered partially.

1.2 Contributions and Outline

The system MathNat in a nutshell is given in chapter 2. In this thesis, to treat the first problem we develop the following components:

1.2.1 The Controlled language of Mathematics (CLM)

We would like to develop a grammar for the language of mathematics having reasonable coverage but with some rich linguistic features described above. With it, we would like to solve the problems of ambiguity, redundancy, scalability and complexity.

However, before doing anything we need to understand the language of mathematics and its discourse really well. For that we allocate Chapter 3, in which we give a comprehensive survey of the language of mathematics with a focus on elementary mathematics.

After this survey, we develop a **C**ontrolled **L**anguage for **M**athematics (CLM) with the look and feel of textbook mathematics. It is a precisely defined subset of English with restricted grammar, dictionary, style and predefined conventions. By doing this we reduce the natural language side effects.

At the same time, to make CLM natural and expressive, we support some important linguistic features such as anaphoric pronouns and references, rephrasing of a sentence in multiple ways and the proper handling of distributive and collective readings. CLM also support two patterns for 'proof by case' method. We describe the synopsis of CLM grammar in Chapter 5. Whereas, the development of CLM is described in Chapters 6 and 7.

1.2.2 MathAbs - The Mathematical Abstract Language

We need a formal language which can faithfully represent the mathematical texts by preserving their logical and reasoning structure. Such a formal language must be independent of any logic, theory or prover because different logics[1] and theories[2] have their merits and demerits for the formalization of mathematics. Since we intend to represent the language of mathematics "as it is", therefore it is better to postpone such decisions till the phase of proof checking.

Having this in mind, we adapt new_command, a formal language which was originally developed in the DemoNat project [Thévenon 2005, Thévenon 2006] as an intermediate formalism between natural language proofs and the proof assistant PhoX [Raffalli 2005]. For instance, we remove the PhoX dependencies and simplify the proof

[1] For instance, first-order, higher-order, predicative, impredicative, etc.

[2] For instance, different versions of set theories and type theories; and category theory.

language by reducing the constructs to minimal. Further, we add the language of definition and theorem. We also provide the semantics of MathAbs for axioms, definitions, theorems and their proofs, which was not done before. We name it MathAbs (**Math**ematical **Abs**tract language).

MathAbs is intended only for machine manipulation and used as an intermediary between the natural language of the mathematician and the formal language of the logician. We give a detailed account of MathAbs including its motivation, syntax, formal definition, semantics and completeness in Chapter 4.

1.2.3 The Host System MathNat

The CLM (**C**ontrolled **L**anguage for **M**athematics) is an attribute grammar [Knuth 1968, Deransart *et al.* 1988], which we implement in Grammatical Framework (GF) [Ranta 2004, Ranta *et al.* 2010, Ranta 2011a]. This GF grammar of CLM only deals will sentences having no discourse (we call it sentence level grammar). It means that linguistic features such as anaphoric resolution are not yet applied. Similarly, logical blocks such as axiom, definition, theorem, proof are simply the list of sentences in CLM.

It is the host system MathNat, which automatically builds the context from CLM discourse and provides the above mentioned linguistic features. Also the host system MathNat automatically translate CLM to MathAbs. This way the semantics of logical blocks is restored. In this step the CLM is completely formalized. The procedure of doing these various tasks such as context building, anaphoric resolution, discourse building and providing semantics are described in Chapter 8.

We argue that formalizing mathematics in such a formal language that has a precise semantics is an important progress even if it can't always be checked by a prover[3]. Furthermore, we claim that the narrative style of MathAbs makes the translation task easier.

1.2.4 Proof Checking

The second hard problem is validation for which we report work with a very limited scope. For instance, in the current implementation, we translate MathAbs to the first-order formulas which could be given to the automated theorem provers for validation.

As described in the beginning of this section, proof checking of informal mathematical text is problematic because of at least two reasons; first, the reasoning gaps; and second, the relevance of reasoning justifications to the proof checking system.

We do not solve these problems. Therefore, validation of the first-order formulas produced from the MathAbs will hardly be possible. It is because, currently, there is no prover available which understands MathAbs. We describe these initial results in §4.7 of Chapter 4.

[3]By this we mean that there is currently no prover available which understands MathAbs.

Chapter 2

The MathNat Framework

Contents

2.1 Introduction 5
2.2 Analysis of the Language of Mathematics 5
2.3 Semantics (Mathematical Abstract language: MathAbs) 8
2.4 Syntax (The Controlled Language of Mathematics) 12
2.5 Defining CLM 17
2.5.1 Sentence Level Grammar 17
2.5.2 Restoring Discourse 19
2.6 Related Work 23

2.1 Introduction

As noted in Chapter 1, MathNat is a long term project which aims at being the first step towards automatic formalization and verification of mathematical texts. It is also obvious from Chapter 1 that there are many components and concepts we are going to build in this thesis, which need to be explained. Worse, they are intermixed in such a way that it is easy to get lost in details. So, the purpose of this chapter is to describe MathNat in a nutshell. Each section described here will become a chapter subsequently. However this chapter is not meant to be an exhaustive list of concepts and features we currently support in MathNat. The MathNat architecture is given in figure 2.2 on page 8. An overview of the system MathNat has been published in [Humayoun & Raffalli 2010b].

2.2 Analysis of the Language of Mathematics

The term "language of mathematics" or "mathematical language" refers to prose that the mathematician uses in authoring textbooks and published material. Definitely, the first step towards such work is to understand the language of mathematics and its discourse well enough (cf. Chapter 3 for a detailed account). To get started, let us consider a famous theorem (irrationality of $\sqrt{2}$) and its proof given in figure 2.1 as a running example. In the process of explaining this example, we introduce many of the basic concepts of the mathematical language.

We start with the macro view: the text in figure 2.1 is structured in theorem and proof blocks. An unacquainted reader might need supporting axioms, definitions, lemmas, examples, etc, to understand it properly. These structured blocks are referred to as "mathematical discourse"; a detailed account is given in §3.2.

Theorem 43 (PYTHAGORAS' THEOREM) $\sqrt{2}$ is irrational.
Proof. If $\sqrt{2}$ is rational, then the equation

$$a^2 = 2b^2 \qquad (4.3.1)$$

is soluble in positive integers a, b with $(a, b) = 1$. Hence a^2 is even, and therefore a is even. If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$ and b is also even, contrary to the hypothesis that $(a, b) = 1$. □

[Hardy & Wright 1975, pp.39–40]

Figure 2.1: A typical theorem and its proof.

The theorem block in figure 2.1, consists of a statement expressing a proposition. In contrast, the proof is a collection of arguments which establishes the truth of this theorem. Both are usually written in a narrative style.

There are various proof methods which can be used to prove a theorem, lemma, or proposition. For instance, the use of a contradiction to prove the opposite to the primary hypothesis in figure 2.1, suggests that the method "proof by contradiction" is used. A detailed account on various proof methods is given in §3.2.4.1.

On a sentence level, the most prominent feature is the intermixing of text, symbols and notations (cf. §3.3.1). For instance, $\sqrt{2}$, a^2, etc, are symbols and (a, b) is a notation which stands for "the greatest common divisor of a and b" in figure 2.1. Furthermore, regarding the textual as well as symbolic mathematics[1], [Ganesalingam 2009, page 10] notes that the textual mathematics resembles the words of ordinary natural language but has many differences; and similarly, symbolic mathematics resembles the artificial languages but it behaves in a more complex way. Such intermixing of text and symbols is unique and full of rich features. Some of them are following:

1. It is mostly written in an ASCII[2] based typographic language called LaTeX[Lamport 1986]. For instance $\sqrt{2}$ is encoded as $\sqrt{2}$.

2. Dependence of text on symbols and symbols on text; for instance, variables a and b are first used in equation 4.3.1 but introduced in the later part of the same sentence. So, first, we have to interpret the second part and then the first part.

3. A label given to an equation (4.3.1). It is not used in this proof but it could be used in a sentence such as "by substituting the value of b in equation 4.3.1, we get ...", etc.

4. Implicit references; for instance, the use of 'hence' and 'therefore'. If we look at the second sentence of the proof: "Hence a^2 is even, and therefore a is even.", the clue word "hence" suggests that the fact of "a^2 being even" should be deduced from all hypotheses which are introduced before. In contrast, the clue word "therefore" only refers to the last fact. (also noted by [Zinn 2004, p. 79])

5. Similarly, anaphoric pronouns could also be found. For example, in statement "if a and b are even then they have a common factor", the use of "they" refers to variables a and b.

[1] Like [Ganesalingam 2009], we use the term "textual mathematics" for the natural language part and "symbolic mathematics" for the symbolic expressions and notations used in the mathematical texts.

[2] American Standard Code for Information Interchange.

6. Distributive readings; for instance, the variables a and b being even. "even" is a 1–arity predicate which is applied on a and b in a distributive manner: "even(a) $\wedge$ even(b)". Similarly, we might have encountered a collective reading if the equation 4.3.1 would be given in a textual form: "a^2 and $2b^2$ are equal". "equal" is an n–arity predicate, which is applied collectively: "equal($a^2, 2b^2$)".

7. Long sentences with complicated structure. For instance, only three sentences prove this theorem.

8. Keeping the same arguments and line of reasoning, this text could easily be rephrased in many ways. Such rephrasing is normally influenced by the writing taste of an author and the level of details (s)he wants to present.

9. Sometimes variables are not introduced at all. For instance, c is used directly without any introduction in the third sentence. However, variables similar to c are sometimes defined globally at the beginning the textbook with the sentence such as: "small letters such as $a, b, c, \ldots, x, y, z$ represent integers, except otherwise explicitly mentioned", etc.

10. Ambiguity in textual parts of mathematics, such as coordination ambiguity, word sense ambiguity, attachment ambiguity, quantifier scope ambiguity, etc.

11. However, in terms of the use of natural language and its meaning (semantics), mathematical language is somewhat simpler. It uses certain key phrases which have definite meanings. For instance, statements "We assume that P", "We conclude that P", etc. Similarly it uses certain patterns which have definite meaning. For instance, "If P then Q", "For all P such that Q, R holds", etc.

12. Symbolic conventions such as an optional use of times ($\times$) operator for multiplication (for instance, $2b^2$ means $2 \times b^2$).

13. Another major symbolic convention is "precedence" in expressions. For instance, $a = 2c$ in figure 2.1 is interpreted as "equal(a, times$(2, c)$)". It is so because it is an expression of elementary number theory where multiplication has higher precedence than, let us say, equality. Precedence is context dependent. For instance, as noted by [Sacerdoti Coen 2009], "equality on propositions (denoted by $=$, a notational abuse for co-implication) has precedence higher than conjunction (denoted by $\wedge$), which is higher than equality on set elements (also denoted by $=$), which is higher than meet for lattice elements (also denoted by $\wedge$). Thus $A = B \wedge P$ can be parsed either as $(A = B) \wedge P$ (a conjunction of propositions) or as $A = (B \wedge P)$ (equality of lattice elements)".

14. Similar to textual mathematics, symbolic mathematics is also highly ambiguous. With conventions (cf. bullet 12 and 13), it tries to use the same few notations over and over again for concepts that share similar properties or intuitive meaning. For instance, symbol "$-$" in context of number theory subtracts two numbers, but in set theory it sometimes gives the difference of two sets[3] (which is intuitively similar to subtraction).

A detailed analysis is given in Chapter 3. But how do we solve these issues? We answer this in §2.4.

[3]Sometimes another symbol "$\setminus$" is used.

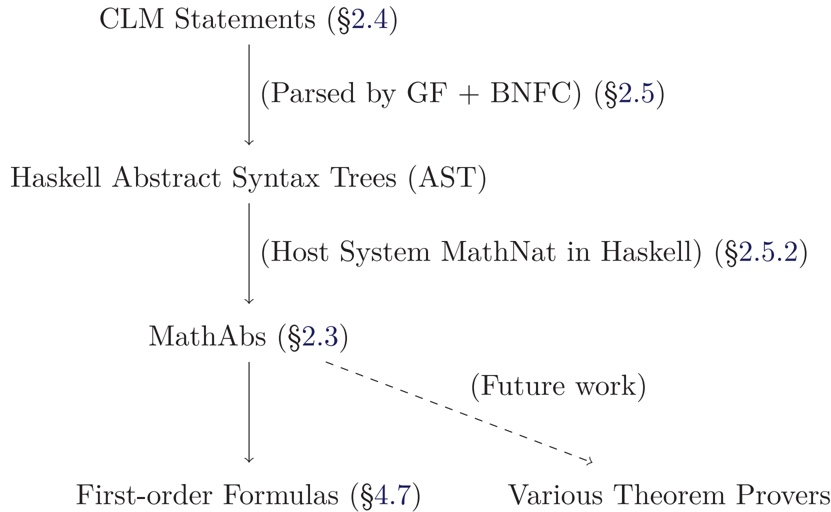


Figure 2.2: MathNat Architecture in a Nutshell

2.3 Semantics (Mathematical Abstract language: MathAbs)

Along with some of the linguistic difficulties we discuss so far in §2.2, there is another problem associated with this proof; it contains reasoning gaps. In textual proofs[4], mathematicians tend to exempt obvious parts by performing many reasoning steps in fewer steps (as it is done in proof given in figure 2.1). In contrast, they also tend to perform fewer steps in many detailed steps for the purpose of explanation.

Logical systems such as natural deduction, sequent calculus, Hilbert systems, etc, are not so natural for textual proofs because they do not allow such freedom. However, mathematicians use the rules of these deduction systems[5] freely and informally (but always implicitly) in their textual proofs.

MathAbs (**Math**ematical **Abs**tract language) on the other hand, is not based on any specific logic, theory or proof assistant. Mathematical text in it is simply the same pieces of arguments but without linguistic features and problems. Furthermore, it allows to describe proofs which contain reasoning gaps.

MathAbs can represent theorems and their proofs along with supporting axioms and definitions. However, analogous to mathematical texts, the language of proof is the most significant part.

Similar to proof theory, we can divide mathematical arguments presented in a textual theorem and its proof as follows: we assume facts (called assumptions or hypotheses), we have goals which have to be proved, and we have deductions (also called as deduced hypotheses or conclusions) which are first proved in one branch of the proof and then used as hypotheses in the other branches (implicitly or explicitly).

We keep this section a bit informal by describing how our running example in figure 2.1 can be formalized in MathAbs. We defer it's formal description to Chapter 4. To get started, consider the theorem given in figure 2.1 and its equivalent MathAbs:

Theorem 43 (PYTHAGORAS' THEOREM) $\sqrt{2}$ is irrational.

[4]This term refers to the proofs found in mathematical texts and hence may contain text, symbols and notations.

[5]And the other rules which can be derived from them.

Theorem_43_(PYTHAGORAS'_THEOREM). show $\sqrt{2} \notin \mathbb{Q}$ •;

The MathAbs' semantics can be very well described as a tree:

$$\dfrac{\dots}{\Gamma_0 \vdash \sqrt{2} \notin \mathbb{Q} \text{ (show rule)}}$$

In MathAbs' semantics, a theorem and its proof are described as a tree of logical rules. Theorem forms the initial sequent with some hypotheses[6] and a goal. Related axioms and definitions could have added as hypotheses in Γ_0. Rule show adds or change a goal in this tree and full-stop (•) marks the end of a sentence which is shown in the proof tree with double line. Furthermore, the proposition "$\sqrt{2}$ is irrational" is translated as "$\sqrt{2} \notin \mathbb{Q}$". Now consider the first sentence of proof from our running example.

Proof. If $\sqrt{2}$ is rational, then the equation

$$a^2 = 2b^2 \qquad (4.3.1)$$

is soluble in positive integers a, b with $(a, b) = 1$.

We divide it in two parts for future references:

Proof. $\underbrace{\text{If } \sqrt{2} \text{ is rational, then the equation } a^2 = 2b^2 \quad (4.3.1) \text{ is soluble}}_{\text{First part}}$ $\underbrace{\text{in positive integers } a, b \text{ with } (a, b) = 1}_{\text{Second part}}$.

First, we have to analyze the second part. Because it introduces a and b (with two conditions: being positive and having no common factor) which are required to analyze the first part. In MathAbs, we introduce variables with rule "let"[7] and add hypothesis with "assume", as shown below. This translation is compositional, i.e. the meaning of the whole is a function of its parts.

Proof. let $a, b \in \mathbb{Z}$
assume $(a > 0) \wedge (b > 0) \wedge (\gcd(a, b) = 1)$

These formulas are treated as predicates. Therefore, "$(\gcd(a, b) = 1)$" is syntactic sugar for "equal(gcd$(a, b), 1)$" and "$a > 0$" is an equivalent expression to "positive(a)". This MathAbs adds these variables and assumptions as hypotheses in the tree:

$$\dfrac{\dfrac{\dots}{\Gamma_1 \equiv (\Gamma_0, (a, b \in \mathbb{Z}), (a > 0) \wedge (b > 0) \wedge (\gcd(a, b) = 1)) \vdash \sqrt{2} \notin \mathbb{Q}}}{\Gamma_0 \vdash \sqrt{2} \notin \mathbb{Q} \text{(show)}} \text{let, assume}$$

For the purpose of readability, proof steps could be expanded or collapsed. (Every rule equally matters). For instance the above tree could be expanded as follows:

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma_2 \equiv (\Gamma_1, (a > 0) \wedge (b > 0) \wedge (\gcd(a, b) = 1)) \vdash \sqrt{2} \notin \mathbb{Q}}}{\Gamma_1 \equiv (\Gamma_0, (a, b \in \mathbb{Z})) \vdash \sqrt{2} \notin \mathbb{Q}} \text{assume}}{\Gamma_0 \vdash \sqrt{2} \notin \mathbb{Q} \text{ (show)}} \text{let}$$

[6]It is Γ_0 which is currently empty. But it may contain necessary axioms, definitions, propositions, theorems, etc, that are needed to support this proof.

[7]We use "let to introduce universally quantified variables. To see how we decide the quantification of a variable, see §4.6.

The first part above, is of the form "if A then B. Of course it is the textual form of "$A \Rightarrow B$". Intuitively[8], if we assume A then the statement to be deduced will be B. It corresponds to "assume A deduce B" in MathAbs as shown below:

assume $\sqrt{2} \in \mathbb{Q}$
deduce $a^2 = 2b^2$ (4.3.1)

The rule deduce B is a short hand for:

{
show B trivial;
assume B ...
}

It is a *proof by case* with precisely two cases, where "trivial" marks the end of the proof for the active sequent. Intuitively, for each deduction, first we have to prove it as a goal (the first case) and then we use it as an hypothesis for the rest of the proof (the second case), as shown below:

$$\dfrac{\dfrac{}{\Gamma_0 \vdash B \text{ (as a goal, show)}}\text{trivial} \qquad \dfrac{\dots}{\Gamma_1 \equiv (\Gamma_0, B) \vdash \text{main goal of the proof (assume)}}}{\Gamma_0 \vdash \text{main goal of the proof}}\text{deduce}$$

Now combining both parts of the sentence results in the following MathAbs and proof tree:

Proof. let $a, b \in \mathbb{Z}$
assume $(a > 0) \wedge (b > 0) \wedge (\gcd(a, b) = 1)$
assume* $\sqrt{2} \in \mathbb{Q}$
deduce $a^2 = 2b^2$ (4.3.1) •

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash (a^2 = 2b^2)}\text{trivial} \qquad \dfrac{\dots}{\Gamma_4 \equiv (\Gamma_3, (a^2 = 2b^2)) \vdash \sqrt{2} \notin \mathbb{Q}}}{\Gamma_3 \equiv (\Gamma_2, \sqrt{2} \in \mathbb{Q}) \vdash \sqrt{2} \notin \mathbb{Q}}\text{deduce}}{\Gamma_2 \equiv (\Gamma_1, (a > 0) \wedge (b > 0) \wedge (\gcd(a, b) = 1)) \vdash \sqrt{2} \notin \mathbb{Q}}\text{assume*}}{\Gamma_1 \equiv (\Gamma_0, (a, b \in \mathbb{Z})) \vdash \sqrt{2} \notin \mathbb{Q}}\text{assume}}{\Gamma_0 \vdash \sqrt{2} \notin \mathbb{Q} \text{ (show)}}\text{let}$$

Note the rule assume marked with asterisk * in the above MathAbs and in the proof tree. It is uncommon to separate hypothesis $((a > 0) \wedge (b > 0) \wedge (\gcd(a, b) = 1))$ from the assumption that $\sqrt{2} \in \mathbb{Q}$. However it does not result into any logical fallacy. See §4.5 on page 68 for further discussion.

There is another point worth mentioning. The procedure we have used to translate the first sentence to its equivalent MathAbs also works for the arbitrary sentence, as shown below:

1. Filter all the let variables and place them before any other rules occurring in that sentence.

[8] More precisely, it uses ($\Rightarrow$ intro) rule of natural deduction; cf. 4.3 for a complete list of rules.

2. Then place the conditions associated with these variables. For instance adjuncts, "where" clauses, etc.

3. Then translate the rest of the sentence as it occurs. i.e. the things which appears first should be translated (and placed in MathAbs) first.

Now consider the second sentence of the proof:

> Hence a^2 is even, and therefore a is even.

The clue word "hence" suggests that the fact of "a^2 being even" should be deduced from all hypotheses which are introduced before. Because each proof step is a logical consequence of the previous steps, we simply translate it as "deduce even(a^2)".

In contrast, the clue word "therefore" suggests that "a being even" is solely based on the fact that "a^2 is even" (cf. §3.2.4 on page 37). Such justification could be given as a Hint (for instance, by form even(a^2) below) to MathAbs rules. The MathAbs of this sentence would be as followed:

> deduce even(a^2)
> deduce even(a) by form even(a^2) •

It updates the proof tree as followed, where "even(a^2)" in "deduce (even(a^2))" represent a hint:

$$\dfrac{\dfrac{}{\Gamma_4 \vdash \text{even}(a^2)}\ \textsf{trivial} \qquad \dfrac{\dfrac{}{\Gamma_5 \vdash \text{even}(a)}\ \textsf{trivial} \qquad \dfrac{}{\Gamma_6 \equiv (\Gamma_5,\ \text{even}(a)) \vdash \sqrt{2} \notin \mathbb{Q}}}{\Gamma_5 \equiv (\Gamma_4,\ \text{even}(a^2)) \vdash \sqrt{2} \notin \mathbb{Q}}\ \textsf{deduce}(\text{even}(a^2))}{\Gamma_4 \equiv (\Gamma_3, (a^2 = 2b^2)) \vdash \sqrt{2} \notin \mathbb{Q}}\ \textsf{deduce}$$

The third sentence is too ambiguous to be understood of its own. So we rephrase it as followed:

> If $a = 2c$, then $4c^2 = 2b^2$. If $4c^2 = 2b^2$, then $2c^2 = b^2$. Consequently, b is even. If a and b are even then they have a common factor. It is a contradiction to the hypothesis that $(a, b) = 1$. □

These are translated as followed, proceeded by some explanation.

> let $c \in$ `TypeUnknown` assume $a = 2c$ deduce $4c^2 = 2b^2$ •
> assume $4c^2 = 2b^2$ deduce $2c^2 = b^2$ •
> deduce even(b) •
> assume even$(a) \wedge$ even(b) deduce one_common_factor(a, b) •
> show $\perp$ by form gcd$(a, b) = 1$ • trivial •

First, we introduce c with a dummy type "`TypeUnknown`[9]" because it is not introduced in the proof. Given the information about the other variables in proof, it may seem appropriate to deduce the type of variable c. However such deduction is not always trivial. Furthermore, our account for semantics is yet to be complete as we only rely on surface level information presented in the textual proofs. It means that we do not deduce any information from types at this point. It should be connected to some

[9] In the rest of the thesis, we call it `NoType`.

proof assistant to have complete semantics and reject interpretations which are wrong (cf. §4.7).

Another point worth noting is: if a user gives an underspecified text, instead of deducing that information, it might be a good idea to report it back to the user (specially if it is an application used to teach mathematics to the students).

In the second line, first we assume the fact that $4c^2 = 2b^2$. As it is already a deduced fact and exists in the context, "assume $4c^2 = 2b^2$" is simply ignored, and therefore, instead of the above MathAbs, following is the MathAbs of above sentences:

> let $c \in$ `TypeUnknown` assume $a = 2c$ deduce $4c^2 = 2b^2$ •
> deduce $2c^2 = b^2$ •
> deduce $\text{even}(b)$ •
> assume $\text{even}(a) \wedge \text{even}(b)$ deduce $\text{one_common_factor}(a, b)$ •
> show $\perp$ by form $\gcd(a, b) = 1$ • trivial •

In the third sentence, we translate "a common factor" as a predicate "one_common_factor(...)", stating that there exists a common factor for variables a and b. In the fourth line, instead of using "deduce", we use "show" as the proof is about to be finished, and therefore, we do not need this fact as an hypothesis anymore. Finally, square box (□) marks an end of proof and translated as "trivial" and "•". A complete proof tree is given in figure 2.3. But how do we verify this MathAbs? Different possibilities are discussed in §4.7.

2.4 Syntax (The Controlled Language of Mathematics)

As we have seen in §2.2, linguistically rich features of mathematical language put forth many obstacles and problems. So if not solved properly, they surely result into massive ambiguity (leading to the problem of complexity) and wrong interpretations for our grammar. One way to deal with these problems is to avoid them. But we cannot do it always, specially when naturalness and expressiveness of the language is on stake. Therefore, an alternate which is also a standard way, would be to define some conventions that somehow resolve these problems.

We also need to restrict the grammar and dictionary, allowing fewer constructions. It would help us in the following ways:

1. A language such as mathematics should be concise and strictly defined, because a carefully defined grammar is easier to interpret.

2. From an engineering point of view, the grammar should be easier to manage and update. This way, it would also be easy to avoid the issue of computational complexity which may arrive for a very large grammar (mainly due to ambiguity).

So we develop the **C**ontrolled **L**anguage for **M**athematics (CLM) as the syntactic component. It is a computer processable subset of the language of mathematics having restricted grammar, dictionary, style and predefined conventions. However, CLM is still natural and expressive. We achieve this by supporting some rich linguistic features such as anaphoric pronouns and references, rephrasing of a sentence in multiple ways and the proper handling of distributive and collective readings. To give a taste, we rephrase the text given in figure 2.1 in three ways as shown in figure 2.4. These versions are processable in CLM. Furthermore, other rephrased versions are also possible.

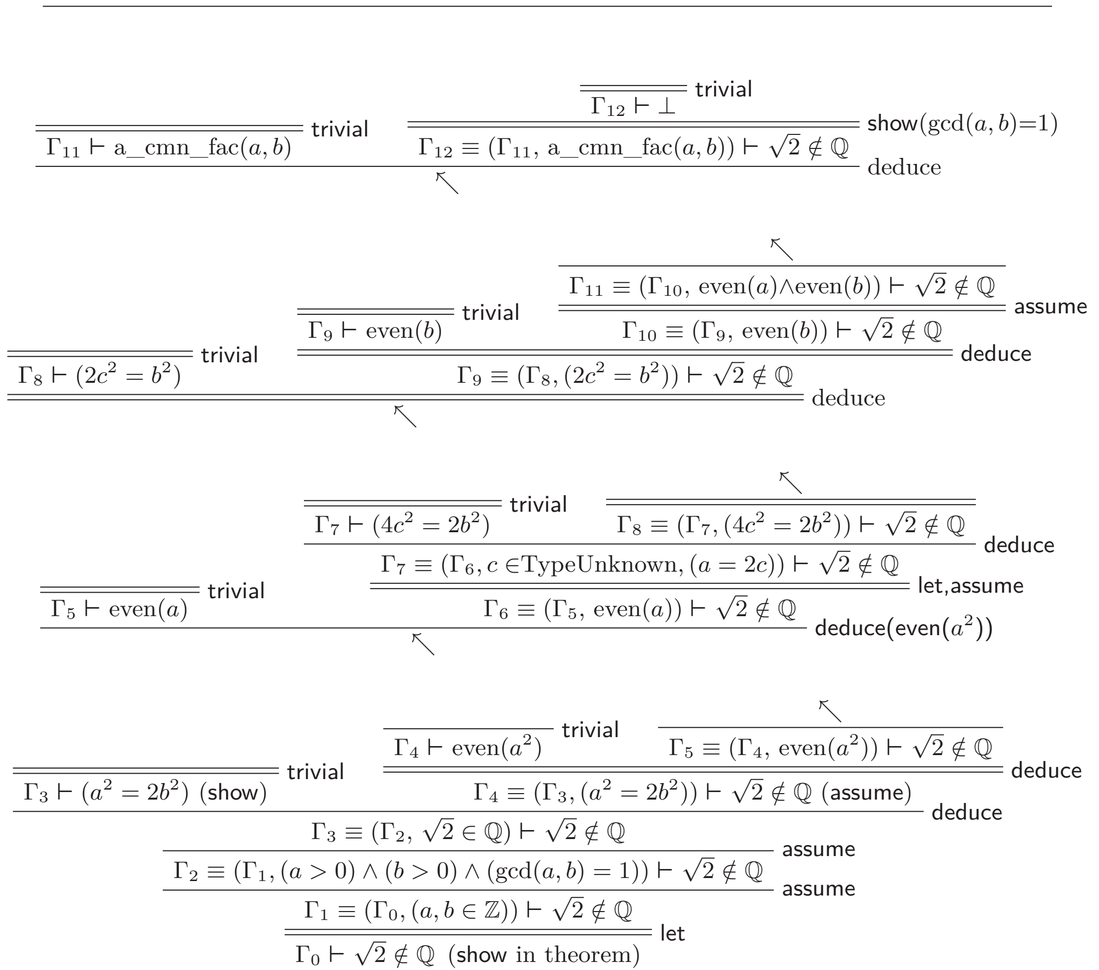


Figure 2.3: Proof tree of theorem and proof given in figure 2.1. We rename 'one_common_factor(a,b)' as 'a_cmn_fac(a,b)' to fit the proof tree on the page.

1. **Theorem 43 (PYTHAGORAS' THEOREM)** $\sqrt{2}$ is irrational.
2. **First Proof.** If $\sqrt{2}$ is rational, then $a^2 = 2 * b^2 \;-\;$ (4.3.1), where a and b are positive integers with $\gcd(a, b) = 1$.
3. Hence a^2 is even; and therefore a is even.
4. If $a = 2 * c$, then $4 * c^2 = 2 * b^2$.
5. If $4 * c^2 = 2 * b^2$ then $2 * c^2 = b^2$.
6. Consequently, b is even.
7. It is contrary to our hypothesis. □

8. **Second Proof.** Suppose that $\sqrt{2}$ is rational. Let a and b be positive integers with $\gcd(a, b) = 1$. Then $a^2 = 2 * b^2 \quad$ (4.3.1).
9. By the last equation a^2 is even; and therefore, a is even.
10. So, we can assume that $a = 2 * c$.
11. Substituting the value of a in equation (4.3.1) returns $4 * c^2 = 2 * b^2$.
12. Dividing both sides by 2 yields $2 * c^2 = b^2$.
13. The last equation implies that b^2 is even; and therefore, b is even.
14. It is contrary to the hypothesis that $\gcd(a, b) = 1$. □

15. **Third Proof.** Assume that $\sqrt{2}$ is a rational number.
16. By the definition of rational numbers, we can assume that $\sqrt{2} = a/b$ where a and b are non-zero integers with no common factor.
17. Thus, $b * \sqrt{2} = a$.
18. Squaring both sides yields $2 * b^2 = a^2 \;-\;$ (1).
19. It is clear that a^2 is even because it is a multiple of 2.
20. So we can write $a = 2 * c$, where c is an integer.
21. We get $2 * b^2 = (2 * c)^2 = 4 * c^2$, by substituting the value of a into equation (1).
22. Dividing both sides by 2, yields $b^2 = 2 * c^2$.
23. Thus b is even because 2 is a factor of b^2.
24. If a and b are even then they have a common factor.
25. It is a contradiction.
26. Therefore, we conclude that $\sqrt{2}$ is an irrational number.
27. This concludes the proof.

Figure 2.4: Three CLM processable versions of figure 2.1.

```
1   "Theorem 43 (PYTHAGORAS' THEOREM)" `sqrt(2)` is irrational.
2   "Proof" If `sqrt(2)` is rational, then `a^2 = 2 * b^2` -- (4.3.1)
3           where `a` and `b` are positive integers with `gcd(a,b)=1`.
4           Hence `a^2` is even; and therefore, `a` is even.
5           If `a = 2*c`, then `4*c^2 = 2*b^2`.
6           If `4*c^2 = 2*b^2` then `2*c^2 = b^2`.
7           Consequently, `b` is even.
8           It is contrary to our hypothesis. QED.
```

Figure 2.5: The First theorem and its proof of figure 2.4 as verbatim.

We give below a non exhaustive list of solutions against the issues we raised in §2.2. We start with the document structure in CLM. It currently supports various blocks such as "axiom", "definition", "theorem" and "proof". A structural block is a non-empty list (or sequence) of sentences. However, if these sentences in a block are not semantically well-formed, suitable error messages are reported to the user.

As regards the typography, it is written in ASCII format. Instead of LaTeX, we use ASCIIMath[10] to display symbolic expressions. It is quite similar to LaTeX but allows to render expressions on web pages. To illustrate it, we give the ASCII version of the first example of figure 2.4 in current implementation, in figure 2.5.

Here "Theorem 43 (PYTHAGORAS' THEOREM)" and "Proof" mark the start of a theorem block and proof block respectively. The grammar could recognize the symbolic parts without quotation marks (`...`). We only need them to allow ASCIIMath script to render symbolic parts properly in web browsers.

As regards the anaphoric references, they are supported in CLM. For instance, on lines 2, 8 and 18 of figure 2.4, equations are referenced. These references are then made available in the rest of the text to refer. Similarly, CLM also support implicit references. For instance, "dividing both sides by 2" on line 12 of figure 2.4 implies that there is an equation in some previous sentence (same hold for "squaring both sides" on line 18). Furthermore, CLM supports reference to equations, hypotheses, deductions and statements (such as "by the last equation", "by the first hypothesis", "substituting the value of a in equation 4.3.1", etc).

As we'll see in Chapter 3, anaphoric pronouns are also quite common in math texts. Therefore, pronoun "it" and "they" are supported. However, to resolve them we follow a naïve convention. i.e. they are always replaced by the latest singular or plural objects in the discourse. For instance, in the statement on line 24, "if a and b are even then they have a common factor", the pronoun "they" refers to a and b.

In a similar vein, CLM supports rephrasing of a sentence in multiple ways. For instance, instead of a conditional: "If A then B", we may rephrase it as "Suppose that A. Then B". See bullet 2 of figure 2.4, which we rephrase in second proof (cf. bullet 8) and in third proof (cf. bullets 2 and 3).

In bullet 3 and 13 of figure 2.4, we can see the following pattern of a *mathematical sentence* (brackets show how we interpret it to solve the coordination ambiguity):

$$\text{Pattern 1. } \underbrace{\text{Statement}_1;\ \text{and}\ \underbrace{\text{Statement}_2;\ \text{and}\ \underbrace{\ldots;\ \text{and Statement}_n.}}}$$

(For referral purpose, we call this sequence "mathematical sentence".)

[10] ASCIIMath homepage: http://www1.chapman.edu/~jipsen/mathml/asciimath.html

In contrast, each statement (i.e. Statement$_i$) in the above pattern may have the following structure.

$$\text{Pattern 2. } \textit{Optional key phrase } \underbrace{P_1, \underbrace{P_2, \ldots, \underbrace{\underbrace{P_{n-1}(\textsf{, and} \mid \textsf{, or})\ P_n}}}}$$

(For referral purpose, we call this sequence "mathematical statement".)

So, a mathematical statement is formed by a sequence of propositions P_i logically connected with each other by "$\textsf{, (and|or)}$", which is then concatenated to an optional key phrase. An example of pattern 2 would be following, where "we assume that" is the *key phrase* making it an assumption or hypothesis:

We assume that x is positive, y is negative , and z is even.

Note that each P_i of pattern 2, may also contain "(and|or)". An example could be:

We assume that $\underbrace{x \text{ is positive or negative}}_{P_1}$, $\underbrace{y \text{ is negative}}_{P_2}$, and $\underbrace{z \text{ is even and positive}}_{P_3}$.

If we substitute the pattern 2 in pattern 1, we get the following detailed pattern for the *mathematical sentence*:

Optional key phrase $P_1, P_2, \ldots, P_{n-1}(\textsf{, and} \mid \textsf{, or})\ P_n$ **; and**
Optional key phrase $P_{1'}, P_{2'}, \ldots, P_{n'-1'}(\textsf{, and} \mid \textsf{, or})\ P_{n'}$ **; and**
.... **; and**
Optional key phrase $P_{1''}, P_{2''}, \ldots, P_{n''-1''}(\textsf{, and} \mid \textsf{, or})\ P_{n''}$.

As we can see, the semi-colons (;) are used to disambiguate between the *mathematical sentence* and the *mathematical statements*[11]. Whereas the '$\textsf{, (and|or)}$ ' are used to disambiguate between the *mathematical statements* and *propositions* P_i. Two example mathematical sentences would be following, where the second one is rather superficial:

If $a = 2c$, then $4c^2 = 2b^2$ **; and** if $4c^2 = 2b^2$ then $2c^2 = b^2$ **; and** therefore b is even.

We assume that x_1 is positive, x_2 is negative, and x_3 is even **; and** we assume that y_1 is positive, and y_2 is negative **; and** therefore, we conclude that $x_1 * y_1 * x_2 * y_2$ is positive.

Note that we say nothing (yet) about the other kinds of ambiguity and notational collision of symbolic math (cf. bullet 10 and 13 of §2.2). These topics are discussed in §9.2.1.

[11] According to the pattern given here, a statement is itself a mathematical sentence. So, the distinction we made between them is superficial and only for the purpose of addressing them in unambiguous manner.

2.5 Defining CLM

We develop CLM in two concrete steps. The first step consists of the sentence level grammar developed in Grammatical Framework (GF) [Ranta 2004, Ranta *et al.* 2010, Ranta 2011a]. Instead of context-free, it is an attribute grammar [Knuth 1968, Deransart *et al.* 1988]. "Theoretically, GF is equivalent to **PMCFG (Parallel Multiple Context-Free Grammars,** [Seki *et al.* 1991]), which lies between mildly context-sensitive and fully context-sensitive grammars" as stated by [Ranta 2011a, page 10]. It is specifically designed to describe domain-specific or controlled grammars [Angelov & Ranta 2010].

However, the grammar designed in GF for CLM, only deals with syntax. It parses individual sentences in the usual manner; therefore, we call it sentence level grammar. For instance, a theorem or proof in GF grammar is only a list of sentences having no connection. There are also many semantically motivated constraints which we do not enforce in GF (i.e. at syntax level). As an example, consider the following sentences:

x and y are three positive numbers.
x has a common factor.

Linguistically they are well-formed statements; but logically ill-formed. For instance, in the first sentence we should have written "two" instead of "three"; and in second sentence, we should have mentioned more than one identifiers (we have only given one: "x").

Because GF is a limited programming language, it is not easy to enforce all such constraints in it[12]. Therefore, as a second concrete step, we enforce them in the host system MathNat (cf. §8.2).

The host system MathNat that is written in Haskell programming language [Marlow 2010], also builds the CLM discourse, provides the miscellaneous linguistic features (cf. §2.5.2 and §8.3) and gives semantics (cf. §2.3 and Chapter 4). We discuss these steps in the following subsections.

2.5.1 Sentence Level Grammar

GF is a type-theoretical grammar formalism. Inspired by Curry's distinction between *tectogrammatical* and *phenogrammatical* structure [Curry 1961], every GF grammar has two components: *abstract syntax* and *concrete syntax*. An **abstract syntax** defines an ontology, or in other words, semantic conditions to form *abstract syntax trees* of a language. In contrast, the **concrete syntax** is a set of linguistic objects (strings, inflection tables, records) associated to *abstract syntax trees*, providing rendering and parsing. In other words, it defines a mapping from abstract syntax to a language (and back, by reversibility).

In order to introduce GF and CLM (in a short and quick manner), an example grammar is presented for the following propositions taken from figure 2.5:

"$\sqrt{2}$ is irrational.", "x is irrational.", "It is irrational."

[12]We can enforce some of such semantically motivated constraints using dependent types. But it would make the abstract syntax quite complicated. For more reasons see §5.2 on page 93.

```
1  Proposition ::= Subject "is" Type
2
3  Subject     ::= Pron | SymbMath
4  Pron        ::= "it"
5  SymbMath    ::= String
6
7  Type        ::= Irrational | Integer | Number | ...
8
9  Irrational  ::= "irrational"
10 Integer     ::= "integer"
11 ...
```

Figure 2.6: Context-free grammar for simple propositions

We first define it in context-free notation as shown in figure 5.1. For the sake of readability, we keep these propositions very simple by not considering the plural subject (i.e, the pronoun "they" and more than one variables: "x, y and z", are not considered). Therefore, we do not need number agreement.

The context-free grammar shown in figure 5.1 becomes a pair of abstract and concrete syntax rules in GF. We give the rules for abstract syntax in figure 2.7. First, we need to define these categories with keyword `cat`, as shown on lines 1. The keyword `fun` stands for function declaration. In line 3, `fun` declares the function `MkProp` of type "`Subject -> Type -> Proposition`"; meaning it takes two parameters (a subject and a type[13]) and forms a proposition.

The arrow (`->`) is the usual function type arrow of programming languages. A subject is formed by pronoun (as shown on line 6) and math symbol (as shown on line 7).

Next, we define the function (`It`) which belongs to the category pronoun (on line 8). We define the function `Irrational` of the category `Type`. In full CLM grammar, we add types (e.g. integer, natural number, rational, set, group, proposition, etc) in a similar fashion.

Finally, as shown on line 9, we treat symbolic mathematics as an uninterpreted string (cf. `MkSymb`) in GF. An example of an abstract syntax tree in this grammar is:

```
MkProp (MkSymbSubj (MkSymb "sqrt(2)")) Irrational
```

Defining a formal grammar in GF is definitely possible. However, it comes with a penalty in efficiency for parsing[14]. It is because GF is general enough to cover both natural language grammars and formal language grammars. Of course this generality has a price in terms of efficiency.

Therefore, we define the symbolic expressions for symbolic mathematics outside GF as Labelled BNF grammar (LBNF). The host system MathNat, input this grammar to BNF Converter tool (BNFC) [Forsberg & Ranta 2004, Forsberg & Ranta 2005]. BNFC is a specialized tool for generating compiler front-ends[15] from LBNF grammars. The string in the above example (`"sqrt(2)"`) is interpreted as the following abstract syntax tree:

[13] It is a linguistic type which is not exactly similar to the notion of type in type theory.

[14] This penalty in efficiency is not huge now. It is because this observation is taken in 2008 and since then GF is improved a lot. See §6.2 for some other reasons.

[15] More precisely, a lexer, parser, abstract syntax definitions and pretty-printer.

```
1  cat Proposition; Subject; Type; Pron; SymbMath;
2
3  fun MkProp: Subject -> Type -> Proposition;
4
5  fun MkPronSubj : Pron        -> Subject;
6  fun MkSymbSubj : SymbMath -> Subject;
7
8  fun It: Pron;
9  fun MkSymb: String -> SymbMath ;
10
11 fun Irrational : Type;
```

Figure 2.7: Abstract Syntax for simple propositions

`EFun (Ident "sqrt") (EInt 2)`

The host system MathNat separates category `SymbMath` into expressions (including variables), equations, notations and formulas with semantic checks. It then seamlessly integrates the parse trees of textual mathematics given by GF and parse trees of symbolic mathematics given by BNFC tool. So the final abstract syntax is:

To map this abstract syntax into its concrete syntax, we define the linearization type (lincat) of each abstract syntax category (cat), and the linearization (lin) of each abstract syntax function (fun), as shown in figure 2.8. In line 1, we say that all categories are simply string records. To form the linearization of a proposition, in the function (`MkProp`) on line 3, we select the string values of subject and type (with `.s`), and concatenate them with the string "is" (using `++`). In lines 5–6, we up-cast the pronoun and the math symbols as subject. Then, we define the linearization of the functions `It` and `Irrational` as string records on line 8 and 11 respectively. Finally, we define the linearization of the function (`MkSymb`). It takes a string value (`symb.s`) and put around two single quotes (`‘...‘`).

In this concrete syntax, the abstract syntax tree given on page 19 is linearized to:

{s = "`‘sqrt(2)‘` is irrational"}

In this simple example, we avoided inflection, agreement or other complex morpho-syntactic features. However, as we'll see in Chapters 5, 6 and 7, GF allows more sophisticated linearization rules (with records, finite functions and algebraic data types).

2.5.2 Restoring Discourse

It is possible to compile GF grammars (and Labelled BNF grammars) into code usable in general purpose programming languages. For that GF provides an API which makes it possible to access functionality such as *linearize* and *parse* as ordinary functions. When the math text is parsed by GF, a list of sentence level abstract syntax trees (AST) is produced. We recognize each AST by pattern matching and build the *context* from

```
1  lincat Proposition, Subject, Type, SymbMath, Pron = {s:Str};
2
3  lin MkProp subj type = {s: subj.s ++ "is" ++ type.s};
4
5  lin MkPronSubj pron = pron ;
6  lin MkSymbSubj symb = symb ;
7
8  lin It = {s="it"};
9  lin MkSymb symb = {s= "‘" ++ symb.s ++ "‘"};
10
11 lin Irrational = {s= "irrational"};
```

Figure 2.8: Concrete syntax for abstract syntax given in figure 2.7

CLM discourse. The *context* is a collection of three tables containing information that occurs in the mathematical texts as shown in the subsequent subsections.

2.5.2.1 Saving Information of Symbolic expressions, Equations and Pronouns

For an AST, we record every occurrence of symbolic expressions, equations, pronouns and references. To explain the procedure, we give below the analysis of the first two sentences (line 1–2) of figure 2.4.

For the first sentence, we record: its sentence number in the text; the object about which we gather information (object is an anaphoric referent); its number (1 if singular, length of the object if plural); whether it is universally quantified or existentially; is it explicitly declared in the text by the user or implicitly by the system MathNat; and finally, its type; as shown below:

Sentence No.	Object	Number	Quantification	Declaration	Type
1.	$\sqrt{2}$	1	None	Explicit	Irrational

In the quantification column above, we save “none” because $\sqrt{2}$ is a literal.

We may have more than one objects that occur in a sentence and therefore, record them all. For instance, for the second sentence shown bellow,

> “If $\sqrt{2}$ is rational, then $a^2 = 2 * b^2 - (4.3.1)$ where a and b are positive integers with $\gcd(a, b) = 1$”,

We record information in the following way. First, when the system reads “$\sqrt{2}$ is rational”, it records the following information:

Sentence No.	Object(s)	Number	Quantification	Declaration	Type
2.	$\sqrt{2}$	1	None	Explicit	Rational

Then the system reads the equation “$a^2 = 2 * b^2 - (4.3.1)$”. At this point we do not have any information about a and b, so only partial information is stored. Also because these two variables are not yet declared, MathNat implicitly declares them. (cf. first two line below.)

We proceed further and record only the left hand side of the equation as a convention (see §8.3 for the motivation of this convention). Therefore, information about a^2 is stored, as shown in the third line of the table below.

Sentence No.	Object(s)	Number	Quantification	Declaration	Type
2.	a	1	QLet	Implicit	Unknown
2.	b	1	QLet	Implicit	Unknown
2.	a^2	1	QLet	Explicit	Unknown

Note that in the quantification column above, we save "QLet" because a is quantified by the MathAbs rule let.

Finally, MathNat reads the last part of this sentence. So we come to know that a, b are integers and record this information. This information about a and b supersedes the old information in the context regarding them (so the old information is no longer available for anaphoric resolution).

Sentence No.	Object(s)	Number	Quantification	Declaration	Type
2.	a, b	2	QLet	Explicit	Integer
2.	$\gcd(a, b)$	1	QLet	Explicit	Unknown

This information allows us to support the following linguistic features:

Naive Anaphoric Resolution for Pronouns If a sentence contains the pronoun "it", it is replaced by the latest singular object appeared before in the context with a condition that it should be declared explicitly (`Explicit`). E.g. the second "it" in sentence:

> "it is clear that a^2 is even because **it** is a multiple of 2."

on line 19 of figure 2.4 is replaced by a^2. Similarly, in sentence:

> "if a and b are even then **they** have a common factor.",

on line 24 of figure 2.4, the pronoun "they" is replace by "a and b". Furthermore, the information regarding number and type solves anaphora of the following kind:

> "Let x, y, z be integers. [...] Suppose that $a + b + c > 0$. assume that a and b are positive numbers. [...] **They** are three even integers."

Note that, we save expressions collectively for each sentence. Therefore, the following table is built for the above example sentences:

Sentence No.	Object(s)	Number	Quantification	Declaration	Type
$n+1$	x, y, z	3	QLet	Explicit	Integer
...	...	any but not 3	...	...	...
$n+2$	$x+y+z$	1	QLet	Explicit	Unknown
k	a, b	2	QLet	Explicit	Number
...	...	any but not 3	...	...	...
j	They (x, y, z)	3	QLet	Explicit	Integer

The last pronoun "they" is replaced by the objects in the first row above: "x, y and z". It is because we lookup for the object(s) whose number field is 'three' and type field is 'integer', as shown in the table.

Also note that the referent object in the third row: "$x + y + z$" is an expression (not three separate variables). So that is why we did not resolve anaphora with this object.

It may seem a bit strange, explaining one of the reasons why we call this algorithm 'naive'. This procedure is formally described in §8.3.1.1 on page 198.

Naive Anaphoric Resolution for Demonstrative Pronouns Similarly, (number, type) pair of the above table allows to solve the anaphora of demonstrative pronouns (“this” and “these”). E.g. for a statement such as “this integer is positive”, we lookup the latest object with (1, Integer) pair.

This procedure is skipped in Chapter 8 due to space limitations.

2.5.2.2 Saving Information of Logical Formulas

In §2.3, we have translated the statements of our running example into equivalent logical formulas. We save this information in a second table of our context. This information is useful in many ways as described below, as well as in bullet §1 of §4.6 on page 74.

For instance, for the first statement, we record its sentence number in the text, logical formula and statement type, as shown below:

S#	Logical Formula	Statement Type
1.	$\sqrt{2} \notin \mathbb{Q}$	Goal

In contrast, the second statement is recorded in parts as shown below:

S#	Logical Formula	Statement Type
2.	$(a, b \in \mathbb{Z}) \wedge \text{positive}(a) \wedge \text{positive}(b) \wedge \gcd(a, b) = 1$	Hypothesis
2.	$a^2 = 2 * b^2$	Deduction

This information allows us to support the following linguistic features:

Anaphoric Resolution for References This table helps to solve the reference to hypotheses, deductions and statements. For instance, when we see a sentence such as “we deduce that $x + y = 2 * (a + b)$ by the (first|last) hypothesis”, we solve it with a simple lookup for the (first | last) hypothesis in the context which we have recorded. Similarly, for a statement such as “we deduce ... by the (first | last) deduction”, we lookup for the (first | last) deduction in the context which we have recorded. And finally, for a statement such as “we deduce ... by the (first | last) statement”, we lookup for the (first | last) hypothesis or deduction in the context which we have recorded. It is further described in §8.3.6.2 on page 212.

2.5.2.3 Saving Information of Equations and References

Finally, we save the information regarding equations and their references in a third table[16]. The first statement of text contains no equation, and therefore, there is nothing to record. In the second sentence, we record the equations and their references shown below:

S#	Equation	Reference
2.	$a^2 = 2 * b^2$	4.3.1
2.	$\gcd(a, b) = 1$	no reference

This information allows us to support the following linguistic features:

[16] These three tables collectively form the context. Indeed we can combine this information in a single table. However, we found separate tables easier to manage and explain.

Explicit References to Equation

1. Similar to the anaphoric resolution for logical formulas, we can solve anaphora to the equation of the form "by the (last | first) equation, ...", etc, with this information. For instance, the sentence "By the last equation a^2 is even; ..." on line 9 and "The last equation implies ..." on line 13 of figure 2.4.

2. With available information, it is also possible to solve anaphora of references. For instance the sentence such as "Substituting the value of a in equation (4.3.1) returns $4 * c^2 = 2 * b^2$" is interpreted as: "Substituting the value of a in equation $a^2 = 2 * b^2$ returns $4 * c^2 = 2 * b^2$".

Implicit References to Equation Operations such "multiplying both sides by ...", "dividing both sides by ...", "squaring both sides ...", etc, implies that there is an equation in some previous sentence. So we check this condition and if an equation is found, we put it in place. For instance, we interpret the sentence on line 18 of figure 2.4: "squaring both sides yields $2 * b^2 = a^2 \ -(1)$" as "squaring both sides of equation $\sqrt{2} * b = a$ yields $2 * b^2 = a^2 \ -(1)$".

This procedure is skipped in Chapter 8 due to space limitations.

2.6 Related Work

AutoMath [de Bruijn 1994] of N.d. Bruijn, is one of the pioneering works in which a very restricted proof language was proposed. After that such restricted languages are presented by many systems. For instance, the Mizar language[Trybulec *et al.* 1973], Isar [Wenzel 1999] for Isabelle, the notion of *formal proof sketches*[Wiedijk 2003] for Mizar and Mathematical Proof Language *MPL* [Barendregt 2003] for Coq.

Among all of these languages, Mizar has the strongest claim of resembling the declarative style of the language of mathematics. However, Mizar or *Formal Proof Sketches* (and all the languages mentioned above) are quite restricted, non ambiguous, and have a programming language like syntax with a few syntactic constructions. Therefore, like MathAbs, we consider them as intermediate languages between mathematical text and proof checking system.

The MathLang project [Kamareddine & Wells 2008] goes one step further by supporting the manual annotation of mathematical texts. Once the annotation is done by the author, a number of transformations to the annotated text are automatically performed for automatic verification. For instance, it has been tried to translate some mathematical text from the core language of MathLang (called *CGa*) to the syntax of three proof assistants (Mizar – *formal proof sketches* [Kamareddine *et al.* 2007], Coq [Kamareddine *et al.* 2008] and Isabelle [Lamar *et al.* 2009]).

Although these translations are still in embryonic stage and the treated examples have limited reasoning gaps[17], they inspire us for the possibility of similar work in MathNat. Therefore, in future, we would like to translate MathAbs of some mathematical text in the language of theorem provers, such as Mizar [Trybulec *et al.* 1973], Isabelle

[17] It is mainly the first chapter of Landau's book of analysis [Landau 1966]. However, in [Lamar *et al.* 2009]) some mathematical text from contemporary abstract algebra.

[Paulson 1994], HOL [Gordon & Melham 1993], Coq [Team 2010], etc. For the reasoning gaps, it will require the use of automatic tactics that these systems provide.

In conclusion, MathLang mainly focuses on the second question raised in the introduction but neglects the treatment of the language of mathematics (for instance, its parsing) completely.

The work of Hallgren and Ranta [Hallgren & Ranta 2000] presents an extension to the type-theoretical syntax of logical framework Alfa, supporting a self extensible natural language input and output. One of the authors, Aarne Ranta claims that:

> "[This work] shows that texts consisting of definition and theorem block can be made natural and readable". [Ranta 2011b]

Like MathNat, its natural language grammar is developed in GF but it is not specifically designed for the elementary mathematical texts. Also, it does not support the grammar for textual proofs, as well as, neither it supports the rich linguistic features as we do (such as anaphoric pronouns, anaphoric references, distributive and collective readings, etc).

In historical perspective, the STUDENT system by Bobrow [Bobrow 1964] and Nthchecker of Simon [Simon 1988, Simon 1990] are perhaps the pioneering works for their times. The program STUDENT is a simple question answer system with a very restricted English grammar.

In contrast, the Nthchecker by Simon [Simon 1988, Simon 1990] is probably the first major serious work which tries to answer the question raised in the abstract. The system works in three phases:

1. Lexical analysis
2. Sentence parsing i.e. each sentence is parsed independently.
3. Proof connector. It works in two parts:
   (a) Combines the output from 'mathematics parser' (for symbolic mathematics) and 'sentence parser'.
   (b) Refinement to the formal proof.

The system understands proofs from the book Elementary Theory of Numbers [LeVeque 1965]. Its sentence parser parses 34 proofs. Whereas, the proof connectors combines the output from sentence parser with mathematics parser for 15 proofs, as stated by Simon (excerpts taken from [Simon 1990, page 17]):

> "The proof connector's rules are sufficient to parse 15 of the proofs."

Among them two proofs are mechanically verified, as stated by Simon:

> "The entire proofs of Theorem 1–1 and Theorem 2–3 do go through the proof connector and theorem prover mechanically."

However, its linguistic and mathematical analysis seems shallow and ad-hoc. For instance, the general account on the analysis of the language of mathematics is missing. At the same time, it is not clear how the linguistic features such as anaphoric pronouns, anaphoric references, distributive and collective readings, etc, are supported (if they are supported at all). Finally, Simon does not build or use a linguistic theory for the language of mathematics. Instead, as stated by Simon ([Simon 1990, page 19]):

> "However we make no claims about this system as a linguistic theory, only as a computationally effective device for deciphering the natural language input."

In recent times, Vip – the prototype of Zinn [Zinn 2004, Zinn 2006] is a promising work. In his doctoral thesis and papers, Zinn gives a linguistic and logical analysis of textbook proofs drawn from two textbooks. In the prototype Vip, Zinn develops an extension of discourse representation theory (DRT) [Kamp & Reyle 1993] for parsing and integrates proof planning techniques [Bundy 1996] for verification. The prototype Vip can process two theorem with their proofs (nine sentences) from number theory [Zinn 2004, page 166]. These are given below:

> THEOREM 3 (EUCLID'S FIRST THEOREM). If p is prime and $p \mid ab$, then $p \mid a$ or $p \mid b$.
> Suppose that p is prime and $p \mid ab$. If $p \nmid ab$ then $(a, p) = 1$, and therefore, by theorem 24, there are an x and a y for which $xa + yp = 1$ or
> $$xab + ypb = b.$$
> But $p \mid ab$ and $p \mid pb$, and therefore $p \mid b$. [Zinn 2004, page 146]

> THEOREM 2-2. Every integer $a > 1$ can be represented as a product of one or more primes.
> *Proof.* The theorem is true for $a = 2$. Assume it to be true for $2, 3, 4, \ldots, a-1$. If a is prime, we are through. Otherwise a has a divisor different from 1 and a, and we have $a = bc$, with $1 < b < a$, $1 < c < a$. The induction hypothesis then implies that
> $$b = \prod_{i=1}^{s} p'_i,\ c = \prod_{i=1}^{t} p''_i$$
> with p'_i, p''_j primes and hence $a = p'_1 p'_2 \ldots p'_s p''_1 \ldots p''_t$. [Zinn 2004, page 155]

In our opinion, this coverage is too limited to verify the usefulness of the presented concepts. Like Simon, Zinn tried to answer the two hard problems discussed in the introduction (Chapter 1) at once. Therefore, he was also drifted towards giving the treatment which is shallow rather than deep. In his own words:

> "Our analysis, given the complexity of the task, had to be more shallow than deep." [Zinn 2004, page 164]

Instead of repeating ourself, we refer to [Ganesalingam 2009, 19–21] for an account on shortcomings of this work.

Regarding the above two theorems and proofs, MathNat can parse the first one 'as it is', and the second one partially. To be more precise, all the structural phrases in the second proof are supported by MathNat, including the case markers 'if' and 'otherwise' for 'proof by case' method. The only unsupported feature is ellipses, for which a general account requires further investigation (even in the work of Zinn).

Similar to the work of Zinn on proof checking, we may consider to integrate with proof planning techniques [Bundy 1996] for verification in future.

The Naproche system [Cramer *et al.* 2010b] is also based on an extension of DRT. Like MathNat, Naproche translates its output to the first order formulas. Currently, it has a quite limited controlled language without rich linguistic features. For instance, this work only deals with the proofs taken from the first chapter of Landau's book "Foundations of Analysis" [Landau 1966] (also some proofs from group theory, and Burali-Forti paradox. These proofs also have very limited controlled language). The writing style of this book (and other proofs) is very simple, which we do not see in the other published books. For instance, an excerpt from it is given below:

Axiom 1: 1 is a natural number.
Axiom 2: For every x, x' is a natural number.
Axiom 3: For every x, $x' \neq 1$.
Axiom 4: If $x' = y'$, then $x = y$.
Theorem 1: If $x \neq y$ then $x' \neq y'$.
Proof:
Assume that $x \neq y$ and $x' = y'$. Then by axiom 4, $x = y$. Qed.
Theorem 2: For all x $x' \neq x$.
Proof:
By axiom 3, $1' \neq 1$. Suppose $x' \neq x$. Then by theorem 1, $(x')' \neq x'$. Thus by induction, for all x $x' \neq x$. Qed.
...

MathNat can parse most of the text 'as it is' from the first chapter of Landau's book.

As regards the proof checking, Landau tries to make the text unambiguous and the proofs presented in it do not have very big reasoning gaps. In this sense, like MathNat, the problem of reasoning gaps is yet to be answered.

Having said that, their work on proof checking [Cramer *et al.* 2010a] is still worthy to notice. It must be interesting to see how they manage to select the needed hypotheses, definitions, axioms, etc, from the large poll.

As compared to the Naproche system and the work of Zinn, we did not use DRT because we didn't need its expressive power. Instead, we develop our own theory which analyzes and then, compositionally treat the language of mathematics, as well as, its discourse (as already discussed in §2.3. We will discuss it further in Chapter 4).

Another recent framework FMathL is proposed by Arnold Neumaier [Neumaier & Schodl 2010]. It aims to provide a modeling and documentation language for the working mathematician. The project is promising in its ambition but still in its embryonic stage.

Finally, we refer to an article giving a comprehensive review for systems developed for natural language mathematical problems between 1959–2008 [Mukherjee & Garain 2008].

Theoretical Work

Aarne Ranta has done substantive theoretical work on the language of mathematics by studying and formalizing it in Martin löf's constructive type theory [Martin-Löf 1984]. [Ranta 1994, Ranta 1995, Ranta 1996, Ranta 1997, Ranta 2011b]. His work is mainly concerned with studying the language of mathematics for small domains from linguistic and logical perspective. In Ranta's own words:

> "Following roughly the format of Montague grammar [see the collection [Montague 1974]], I have been working within the constructive type theory of Martin-Löf."

The thesis of Mohan Ganesalingam is a recent work which gives a comprehensive, detailed and convincing analysis of the language of mathematics from linguistics perspective; see chapter 2 and 8 of [Ganesalingam 2009]. It tries to give an analysis covering all of pure mathematics (§1.1 of [Ganesalingam 2009]). However, only a brief account for proof block is given. Similarly only a brief account is given for features such as anaphora, definite descriptions and prepositional phrases.

We also try to give a comprehensive and in-depth analysis of the language of mathematics in Chapter 3. However, our analysis is inclined towards the logical aspects. We discuss various structural blocks and linguistic features (such as anaphora, definite descriptions and prepositional phrases) in a great detail. Like the analysis of Ganesalingam, this chapter is also filled with a lot of examples, that we have taken from various books on number theory, set theory, analysis and algebra; a complete list is given in 3.4.

Ganesalingam proposes a convincing account for "mathematical types" (cf. chapter 5 of [Ganesalingam 2009]) that can sufficiently express the mathematical language. He also propose a novel algorithm for typed parsing (cf. chapter 6 of [Ganesalingam 2009]). However, it is a theoretical work which is not (yet) bound by practical considerations about building a system. Therefore, it seems a bit early to see if his theoretical model can be implemented in the current state of the art. We shall discus it further in Chapter 9.

Another idea that Ganesalingam proposes is the creation of new syntax from math definitions instead of a manually added lexicon on compile time. However, it is not yet clear to us how to implement such a mechanism.

Finally, it seems right to say that the work of Ranta and Ganesalingam is a 'must read' to get a significant insight in this field.

Chapter 3

The Language of Mathematics

Contents

3.1 Introduction **29**

3.2 Mathematical Discourse **31**

3.2.1 Axiom 31

3.2.2 Definition 31

3.2.3 Theorem 33

3.2.4 Proof 35

3.2.5 Remark, Example, Exercise 39

3.3 Micro Structure of the Language of Mathematics **40**

3.3.1 Intermixing of Text and Symbols 40

3.3.2 Ambiguity 42

3.3.3 Statements and Logical Operators 44

3.3.4 Anaphoric Pronouns 49

3.3.5 Anaphoric References 52

3.3.6 Miscellaneous Features 54

3.4 List of Mathematical texts used for analysis **56**

3.1 Introduction

The language of mathematics is a specific scientific prose which is evolved in centuries. Mathematics has a reputation of being an exact science. Consequently, the language that transmits it (i.e. the mathematical language) is also considered to be exact and somewhat easier than the natural language. Ranta states this fact in following words:

> Linguistically, the study of mathematical language rather than everyday language is rewarding because it offers examples that have complicated grammatical structure but are free from ambiguities. We always know exactly what a sentence means, and there is a determinate structure to be revealed. [Ranta 1994, p. 354]

Although, the natural language used in mathematical texts is simpler and restricted to a particular domain, as we'll see, it still contains complex and rich linguistic features.

We can divide mathematical language in two parts: the **mathematical statements** and the **prose of structural blocks** i.e. definition, theorem, proof, etc. The mathematical statements mainly represent atomic facts depending upon the mathematical domain in consideration. For instance, for number theory we may have relations such as equality,

inequality, etc and properties such as positiveness, evenness, etc. These facts are normally glued together by various connectives and quantifiers to form bigger statements. For example in the following statement:

> x is positive and even.

the conjunction 'and' is used to connect two atomic facts to form a bigger statement. These facts are 'x is positive' and 'x is even'. Mathematical statements and connectives are further explained in §3.3.3.

In contrast, mathematical discourse is highly structured and highly personalized. We can classify it in the **prose of structural blocks**, such as definition, axiom, proposition, theorem, proof, lemma, remark, example, etc.

The role of a statement in mathematical discourse is defined by two things. First, the structural block in which it appears and second, the specific clue word or phrase which are added to the mathematical statement. For instance, if we add clue phrases such as 'suppose that' to the above example statement and if it appears in definition, theorem or proof, it will become an assumption. But if we add a phrase 'prove that' and if it appears in theorem, then it'll become a statement that we need to prove. The prose of different structural blocks refers to the **discourse level**; see §3.2 for details.

However, sometimes there is no clue word or phrase added to the statements but it can still play a specific role in the discourse. For instance, in theorem 43 as shown in figure 3.3 on page 34, there is a statement which we have to prove; it doesn't matter if a clue phrase such as 'prove that' is added or not, as shown below:

> $\sqrt{2}$ is irrational. **vs.** Prove that $\sqrt{2}$ is irrational.

Another example could be the following conditional statement in proof which is also taken from figure 3.3 on page 34 and it has no clue word or phrase. Unlike the above statement, it acts as a deduction. It is because it appears in the proof block.

> If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$ and b is also even, $[\ldots]$.

However, if it had appeared in the theorem block, it would be the statement that we have to prove.

At **sentence level**, the language of mathematics consists of a fragment of natural language along with symbolic expressions and notations. For instance, consider again the above example sentence. The natural language parts are typeset as **bold**.

> **If** $a = 2c$**, then** $4c^2 = 2b^2$, $2c^2 = b^2$ **and** b **is also even,** $[\ldots]$.

The use of symbols in mathematical texts makes it easier to read and refer, and make the form compact. For instance, the above statement with less symbols could be as followed:

> **If** a **is equal to** $2c$**, then** $4c^2$ **is equal to** $2b^2$**,** $2c^2$ **is equal to** b^2 **and** b **is also even,** $[\ldots]$.

Or naïvely, the above statement without any symbols at all could be as followed:

> If the first integer is equal to the double of the second integer, then four times the square of the second integer is equal to the double of the square of the third integer, the double of the square of the second integer is equal to the square of the third integer and the third integer is also even, [. . .].

We defer further discussion on intermixing of text and symbols to §3.3.1.

The language of mathematics also contains anaphoric pronouns and references (cf. §3.3.4 and 3.3.5), the possibility of rephrasing of a sentence in multiple ways (cf. §3.3.6.2), distributive and collective readings (cf. §3.3.6.3). Interestingly, these features extend to the symbolic expressions and notations as well in a novel way, which is evident in almost all of the example texts given in this chapter. We also discuss symbolic mathematics in §3.3.1.2 and textual mathematics in §3.3.1.1.

The language of mathematics may also contain graphics such as diagrams in geometry and logic. However, we do not consider them in this thesis.

We proceed to the next section by describing the overall structure of the language of mathematics.

3.2 Mathematical Discourse

The mathematical texts are normally presented in an axiomatic fashion. Mathematical discourse is highly structured. It mainly consists of theorems, lemmas, propositions, corollaries and their proofs along with some supporting axioms, definitions, remarks and examples.

However it is rare to find all of these structural blocks in one text. It depends on the level of formality intended by the author. In general, a text contains theorems, lemmas, propositions and their proofs. While both definitions and axioms may not be mentioned sometimes and taken as granted (e.g. see ([Hardy & Wright 1975], etc). Sometimes axioms are mentioned but definitions are mostly omitted (e.g. see [Jech 2000], [Lang 1997], [Landau 1966], etc) and sometimes definitions are mentioned but axioms are mostly omitted (e.g. see [Baker 2009], [Landau 1958], [Hackman 2007], [Burton 2007], [Jones & Jones 2007] etc)[1]. This enterprise also shows the flexibility and high level of personalization that the language of mathematics offers for authoring mathematical texts. We now give some details about these structural blocks.

3.2.1 Axiom

An axiom consists of a statement or a few, expressing a proposition or fact that is considered to be true without a proof. We give a few excerpts from different math books containing axioms in figure 3.1. We can observe two different styles for the same axiom. The first author intermixes text and symbols. In contrast, the second author uses text only.

3.2.2 Definition

> A definition prescribes the meaning of a [symbol,] word or phrase in terms of other [symbols,] words or phrases that have previously been defined or whose meanings are assumed known. [Bagchi & Wells 1998]

[1]These omitted structural blocks are sometimes given informally.

1.1. Axiom of Extensionality. If X and Y have the same elements, then $X = Y$.
1.2. Axiom of Pairing. For any a and b there exists a set $\{a, b\}$ that contains exactly a and b. [Jech 2000, p. 12]

(1) **Axiom of Extensionality.** If two sets have the same elements, then they are equal.
(2) **Null Set Axiom.** There is a set, $\emptyset$, which has no members.
(3) **Axiom of Infinity.** There is a set x such that $\emptyset \in x$ and such that $\{a\} \in x$ whenever $a \in x$. [Devlin 1993, p. 44]

Figure 3.1: Some axioms from mathematical texts

Similar to the axiom, it also defines a fact which is considered to be true without a proof; and consists of a statement or a few. A definition extends vocabulary of the language of mathematics. We give a few excerpts in figure 3.2. For instance, definitions 2.3, 2 and A.I.5 define **relatively prime** in different ways[2]. A definition may have some or all of the following three parts:

The assumption part: It normally occurs in the beginning of a definition and declares some variables on which the definition applies to. However it is rare to find an assumption without some conditions, which is explained in 'the condition part' below. For example, we declare integers a, b, m, n with some conditions on them, as shown in figure 3.2.

The statement containing symbol, word or phrase being defined: It has two parts: (1) a symbol, word or phrase which is being defined and (2) other symbol, word or phrase which has already been defined. Both parts are joined by clue phrases: 'said to be', 'is defined', 'is called', etc. As an example, see 'said to be' in definition 2.3 of figure 3.2. However similar to axioms, sometimes this statement can be a proposition, containing no clue word or phrase. For example, the second sentence in definition 2.2 of figure 3.2.

The condition part: Both parts mentioned above are normally complemented by some conditions; as evident by definitions 2.2, 2.3, 1.1.1. A condition is normally signaled by clue word or phrase such as 'if' , 'whenever', 'satisfying the following', 'such that', etc.

Conditionals: By contrast, the above mentioned three parts sometimes also incorporated in a conditional statement in which the assumption part is given as a conditional clause (i.e. between 'if' and 'then') and the main statement is given in the consequence clause (i.e. after 'then'). An example of such pattern is definition 2 of figure 3.2.

It is possible to give extra conditions in the consequence clause:

If $a \in G$ has order $n = mk$ then a^k has order m, **where** $\boldsymbol{m, k \geq 1}$.

but these conditions may directly be added in the conditional clause as well:

If $\boldsymbol{m, k \geq 1}$ and $a \in G$ has order $n = mk$, then a^k has order m.

[2] More on rephrasing is given in §3.3.6.2

Definition 2.2. Let a and b be given integers, with at least one of them different from zero. The greatest common divisor of a and b, denoted by $\gcd(a, b)$, is the positive integer d satisfying the following:

- $d \mid a$ and $d \mid b$.
- If $c \mid a$ and $c \mid b$, then $c \leq d$. [Burton 2007, p. 21]

Definition 2.3. Two integers a and b, not both of which are zero, are said to be relatively prime whenever $\gcd(a, b) = 1$. [Burton 2007, p. 22]

DEFINITION 2: If $(a, b) = 1$, that is, if 1 is the only positive common divisor of a and b, then a and b are called relatively prime.
[Landau 1958, p. 16]

A.I.5 Definition. The integers $m, n = 0$ are relatively prime if $(m, n) = 1$, i.e., if their only common factors are 1. [Hackman 2007, p. 03]

Definition 1.1.1. A set A is called an ordered set, if there exists a relation $<$ such that

- For any $x, y \in A$, exactly one of $x < y$, $x = y$, or $y < x$ holds.
- If $x < y$ and $y < z$, then $x < z$. [Lebl 2010, p. 21]

Figure 3.2: Some definitions from mathematical texts

There is another phenomena which appears in all structural blocks. It is the restatement of some parts of a sentence for explanation. For instance, it appears in definition 2 (partially shown below) and it is triggered with a clue word ‘that is’.

> If $(a, b) = 1$, **that is**, if 1 is the only positive common divisor of a and b [. . .]

3.2.3 Theorem

The theorem is a mathematical statement that we have to prove. Once it is proved, it may be used freely as an established fact anywhere in the text. It may have following two parts or only second:

The assumption part: A statement or more may occur in the beginning that assumes some facts about the theorem, declares some variables and/or introduces some conditions. For example, in theorem 1.17 of figure 3.3, we declare a prime number p and two integers a and b.

The proposition we have to prove: The main part of the theorem is the proposition that we have to prove. Following are such propositions:

> If $p \mid ab$, then $p \mid a$ or $p \mid b$. Theorem 1.17
> $\sqrt{2}$ is irrational. Theorem 43

Further, a theorem or proposition given as an exercise to the reader may be aided by a clue word or phrase such as ‘show that’, ‘prove that’, etc. Two example of such

Theorem 1.17 (Euclid's Lemma). Let p be a prime and $a, b \in \mathbb{Z}$. If $p \mid ab$, then $p \mid a$ or $p \mid b$.
Proof. Suppose that $p \nmid a$. Since $\gcd(p, a) \mid p$, we have $\gcd(p, a) = 1$ or $\gcd(p, a) = p$; but the latter implies $p \mid a$, contradicting our assumption, thus $\gcd(p, a) = 1$. Let $r, s \in \mathbb{Z}$ be such that $rp + sa = 1$. Then $rpb + sab = b$ and so $p \mid b$. □ [Baker 2009, p. 11]

Theorem 43 (PYTHAGORAS' THEOREM) $\sqrt{2}$ is irrational.
Proof. If $\sqrt{2}$ is rational, then the equation

$$a^2 = 2b^2 \qquad (4.3.1)$$

is soluble in positive integers a, b with $\gcd(a, b) = 1$. Hence a^2 is even, and therefore a is even. If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$ and b is also even, contrary to the hypothesis that $\gcd(a, b) = 1$. □
[Hardy & Wright 1975, pp. 39–40]

Figure 3.3: Some theorems and their proofs from mathematical texts

propositions are following.

> 1-8. Show that if a prime p divides a product of integers $a1 \ldots a_n$, then $p \mid a_j$ for some j. [Baker 2009, p. 25]
> **Problem 3.11** Show that the function *sign* satisfies $sign(\sigma\tau) = sign(\sigma)sign(\tau)$ for all σ and τ in S_n. [Clark 2001, p. 28]

3.2.3.1 Lemma, Corollary and Proposition

At the level of prose theorem, lemma, corollary and proposition are same. It is also evident from the mathematical texts shown in figure 3.4. Even in usage, the difference between them is rather arbitrary and there does not exist any universally accepted conventions. It is rather a matter of personal style.

However, as noted by [Bagchi & Wells 1998, p. 121][3], lemma, corollary and proposition are normally considered logically subsidiary, easy to prove and having less impact than a theorem. According to Nicholas J. Higham: "a lemma is an auxiliary result–a stepping stone towards a theorem" [Higham 1998, p. 16].

Nevertheless, it is not completely true for a lemma. For example some of the very powerful results in mathematics are represented as lemma. A few examples are following: DuBois-Reymond lemma, Dehn's lemma, Fatou's lemma, Gauss's lemma, Nakayama's lemma, Couchman's Lemma, Zorn's lemma, etc. An example lemma is shown in figure 3.4.

As regards the corollary, "it is a direct or easy consequence of a lemma, theorem or proposition." [Higham 1998, p. 16]. That means a corollary is deduced with a little or no proof from a lemma, theorem, proposition or definition. For instance, in figure 3.4, corollary A.IV.8 is an example for which no proof is given and Corollary 2.5 is an example for which a simple proof is given.

For a proposition, "[it] is less widely used than lemma and theorem and its meaning is less clear. It tends to be used as a way to denote a minor theorem." [Higham 1998, p. 16]. An example text containing proposition is shown in figure 3.4.

[3]The paper [Bagchi & Wells 1998] only says it for lemma and corollary. But we consider it true for

Lemma 1.11. Suppose that $a, b \in \mathbb{N}_0$ are coprime and $n \in \mathbb{Z}$. If $a \mid n$ and $b \mid n$, then $ab \mid n$.
Proof. Let $a \mid n$ and $b \mid n$ and choose $r, s \in \mathbb{Z}$ so that $n = ra = sb$. Then if $ua + vb = 1$,

$$n = n(ua + vb) = nua + nvb = su(ab) + rv(ab) = (su + rv)ab.$$

Since $su + rv \in \mathbb{Z}$, this implies $ab \mid n$. □
[Baker 2009, p. 10]

A.IV.8 Corollary. For a prime number n all non-zero classes modulo n are invertible. □ [Hackman 2007, p. 14]

Corollary 2.5. The only automorphism of a well-ordered set is the identity.
Proof. By Lemma 2.4, $f(x) \geq x$ for all x, and $f^{-1}(x) \geq x$ for all x. □
[Jech 2000, p. 18]

Proposition 1.1.9. Let $x, y \in F$ where F is an ordered field. Suppose that $xy > 0$. Then either both x and y are positive, or both are negative.
Proof. [. . .] Without loss of generality suppose that $x > 0$ and $y < 0$. Multiply $y < 0$ by x to get $xy < 0x = 0$. The result follows by contrapositive.
□ [Lebl 2010, p. 24]

Figure 3.4: Mathematical texts showing proposition, lemma, corollary and their proofs (when given)

3.2.4 Proof

A proof is probably the most important part of mathematical texts. It is a collection of arguments presented to establish the truth of a theorem, lemma, corollary or proposition. It mainly follows a narrative style and its structure mostly remains the same for all mathematical domains. A proof block may have some of the following parts:

Restatements: Sometimes the statement that we have to prove is restated before its proof. The reasons among many can be following. First, an author might want to break it into smaller parts to prove one part at a time. Second, instead of this statement, author might want to prove some other statement which is logically equivalent to it. Following are some examples of restatements.

we have to prove only that D is unbounded. [Jech 2000, p. 428]

we have to prove that the least such multiple of p is p itself.
[Hardy & Wright 1975, p. 302]

Theorem 4. Let A and B be two sets. If $A \cup B = A \cap B$ then $A \subseteq B$.
Proof. [. . .] We shall prove that $x \in A \implies x \in B$, which by definition is equivalent to the consequence of the theorem. [Goldberger 2002, p. 07]

In a similar way, restatements may also occur after its proof.

proposition as well.

Assumptions or Hypothesis: Like the theorem, the assumption that can be called the hypothesis as well, may occur in the proof. The most frequently used keywords for assuming a fact are: 'let', 'suppose that', 'assume that', etc. In figure 3.4, first statements of lemma 1.11 and its proof, and first two statements of Proposition 1.1.9 are assumptions. One of the most important functions of an assumption is to introduce variable(s) and then define some properties on it, which is also evident from these examples. More on variable scope could be found in §4.6.

Conclusions or Deductions: Conclusions are the main part of a proof in which we deduce facts based on our assumptions, axioms, definitions, previously proved theorems, etc. Conclusions are normally triggered by keywords such as 'we conclude that', 'then', 'thus', etc. Furthermore, if a statement appears in a proof not having any kind of keyword, most of the time it is a conclusion. Loosely speaking, most of statements other then assumptions and restatements are conclusions.

Conditionals: As we have already described on page 32, conditional statements have two clauses: the conditional clause (i.e. the part between 'if' and 'then') and the consequence clause (i.e. the part after 'then'). The conditional clause forms assumptions and consequence clause forms conclusions. The fact that conditionals can introduce assumptions is already been observed by many (cf. [Zinn 2006, p. 617], [Roberts 1989] and [Frank & Kamp 1997]).

As an example, consider the conditional below which is taken from figure 3.3.

$$\text{If } \underbrace{\sqrt{2} \text{ is rational}}_{\textit{Assumption}}, \text{ then } \underbrace{\text{the equation } a^2 = 2b^2 \, [\ldots] \text{ with } \gcd(a,b) = 1}_{\textit{Conclusion}}.$$

[Hardy & Wright 1975, p. 40]

More on conditionals which are in fact *implications* is given in §3.3.3.

It is also worth noting that conditionals can easily be replaced by the keywords used to describe an assumption and conclusion. For instance 'Suppose/Let ...' for assumption and 'Then ...' for conclusion. In a similar way their converse is also possible. More on such rephrasing is given in §3.3.6.2.

Justifications: A proof has a precise logical structure in which each statement follows one or more of the following as justification.

- Some of the previous statements in the proof i.e. proof steps.
- Definitions or axioms.
- Theorems, propositions, lemmas or corollaries that has been proved.

These justifications are sometimes made explicit and sometimes left implicit. The most frequently used keywords with explicit justifications are: 'since', 'by', 'because' and 'by the fact that', etc. A few example statements containing explicit justifications are following in which justifications are enclosed in brackets or boldfaced:

$\underbrace{\text{Since } su + rv \in \mathbb{Z}}$, this implies $ab \mid n$. [Baker 2009, p. 10]

We shall prove that $x \in A \Longrightarrow x \in B$, **which by definition is equivalent to the consequence of the theorem** [Goldberger 2002, p. 07]

> $\underbrace{\text{By hypothesis}}$, S must also contain $(a-1)+1=a$, [...]
>
> [Burton 2007, p. 02]

> $\underbrace{\text{By Lemma 2.4}}$, $f(x) \geq x$ for all x, and $f^{-1}(x) \geq x$ for all x.
>
> [Jech 2000, p. 18]

Similarly, 'from this', 'therefore', 'so' and 'hence' can indicate implicit justifications. According to our analysis of different texts and as noted by [Zinn 2004, p. 79] as well, the first three[4] are likely to refer to the last statement in the text, and the last seems referring to all prior premises and asserted statements. For instance, see the use of 'hence' and 'therefore' in Theorem 43 of figure 3.3 on page 34. A few more examples are given below:

> If r were positive, then this representation would imply that r is a member of S, contradicting the fact that d is the least integer in S (recall that $r < d$). **Therefore**, $r = 0$, and so $a = qd$, or equivalently $d \mid a$.
>
> [Burton 2007, p. 22]

> Let $r_1 = f(a)$ and $r_2 = f(b)$. This implies that
>
> $$a = nq_1 + r_1, \quad 0 \leq r_1 < n$$
>
> and
>
> $$b = nq_2 + r_2, \quad 0 \leq r_2 < n$$
>
> **Hence** $a + b = nq_1 + r_1 + nq_2 + r_2 = n(q_1 + q_2) + r_1 + r_2$
>
> [Clark 2001, p. 91]

3.2.4.1 Proof Methods

In this section, we describe some of the most commonly used methods of proof. Loosely speaking, these proof techniques correspond to the laws of sequent calculus and natural deduction (cf. §4.5).

Direct Proof: As noted by [Cupillari 2001, p. 12], a direct proof is based on the assumption that the hypothesis contains enough information to allow the construction of a series of logically connected steps leading to the conclusion. In addition to the hypothesis, these steps may also be aided by the axioms, definitions, and earlier proved theorems. There are no clue words to recognize a direct proof.

Proof by Contradiction: In this method, the contrapositive of conclusion is assumed. Then using this hypothesis a contradiction is established. A famous example of 'proof by contradiction' is the famous 'irrationality of $\sqrt{2}$' proof. Most of the proofs by contradiction end at a phrase similar to: 'This contradicts our assumption. Therefore, we are forced to conclude that ...', 'It is contrary to the hypothesis that ...', etc.

Proof by Case or Exhaustion: Sometimes to prove a theorem, we have to break the problem in parts and then solve them one by one. Parsing this method poses a major

[4] [Zinn 2004, p. 79] do not include 'so' in this list.

challenge of overcoming ambiguity (cf. §7.2.1.7). We describe some of the common patterns along with some examples:

Pattern 1: In this pattern, each case starts with one or more assumptions as shown below:

> We have n cases.
> **Case** 1**:** *assumption(s)* ...
> **Case** 2**:** *assumption(s)* ...
> ...
> **Case** n**:** *assumption(s)* ...

Following are two typical examples of this pattern:

> **Theorem:** For all integers x, the expression $x^2 - x$ is even.
> **Proof:** Let x be an arbitrary integer. There are two cases to consider: Either x is even or x is odd.
> **Case** 1**:** Suppose that x is even. It follows that x^2 is even. Because the difference between two even numbers is even, we conclude that $x^2 - x$ is even.
> **Case** 2**:** Suppose that x is odd. Then x^2 is odd. $x^2 - x$ is even because the difference between two odd numbers is even.
>
> So in either case, $x^2 - x$ is even. [Erickson & Heeren 2007, p. 10]

Similarly,

> *Proof:*
> Case I: Let $a > 0, b > 0$. ...
> Case II: Suppose that the assumption $a > 0, b > 0$ is not satisfied but that a and b are still both $\neq 0$. ...
> Case III: Let one of the two numbers be 0, say $a = 0$, so that $b \neq 0$. ...
> [Landau 1958, pp. 15–16]

Pattern 2: Instead of having the above pattern, sometimes cases are introduced informally as shown below.

> [...] $c \neq 0$; for otherwise we would have ....
> In the **case** $c > 0$, the choice $x = b$, $y = c$ accomplishes what we want.
> In **case** $c < 0$, the choice $x = -c, y = b$ does it. ... [Landau 1958, p. 136]

> **Proof.** The cases $n = 0, 1, 2$ clearly hold. We will prove the result by Induction on n. ... [Baker 2009, p. 19]

Pattern 3: A disjunction such as 'either A or B ...' in the following proof indicates *proof by case*. A proof with a disjunction pattern always has two cases.

> **Either** n is prime, when there is nothing to prove, **or** n has divisors between 1 and n. [...] [Hardy & Wright 1975, p. 02]

Pattern 4: Similarly, patterns such as 'if-then', 'if-then-otherwise' or 'if-otherwise' may also indicate 'proof by case' method. A few examples of such case analysis are:

> **If** $p = n$ we are through. **Otherwise** we apply induction to the quotient $1 < n/p < n$. [Hackman 2007, p. 08]

Pattern 5: Sometimes, bullet list is also used to represent cases as shown in the following example:

> *Proof:* By assumption, $a \neq 0$.
>
> 1. If $b = 0$, then $a = \pm 1$, since $(a, 0) = 1$; and hence $a \mid e$.
> 2. If $b \neq 0$, let m be the smallest positive common multiple of the relatively prime positive numbers ...
>
> [Landau 1958, p. 17]

Some proofs are only aided by one case; may be because the other case(s) can be deduced from it or the proof for other case(s) is similar:

> ***Proof.*** It is enough to consider the case in which b is negative. Then $\mid b \mid > 0$, [...] [Burton 2007, p. 18]

Proof by Mathematical Induction: Most of the proofs that use induction, include a statement such as 'induction on ...', 'We use induction to prove it', etc. We give a few examples:

> *Proof.* **Induction on** n**.** Let p be the smallest divisor > 1 of n. It is a prime number. If $p = n$ we are through. Otherwise we apply induction to the quotient $1 < n/p < n$. [Hackman 2007, p. 08]

> *Proof.* **We will use Induction on** n**.** We can easily verify the cases $n = 1, 2$. Assume that the equations hold when $n = k$ for some $k \geq 2$.... [Baker 2009, p. 19]

An induction proof is a special case of 'proof by case' method. In such a proof we typically have one or more *base cases* and one or more *inductive cases*. It is also evident by these example.

3.2.5 Remark, Example, Exercise

Apart from the above mentioned blocks, we may also find some other kinds of prose in mathematical texts. It does not directly present mathematical facts. Instead it offers an explanation to the reader. A few examples of such prose are: the miscellaneous discussions (normally named as 'remark'), examples and exercises that appear in mathematical texts. However, propositions can also occur in this prose sometimes. For instance, a proposition for which the proof is not given and left as an exercise. We will not discuss them anymore; see [Bagchi & Wells 1998] for more details.

3.3 Micro Structure of the Language of Mathematics

We now discuss the various characteristics of the language of mathematics at sentence level that make it unique and hard to formalize.

3.3.1 Intermixing of Text and Symbols

As introduced in §2.2 and §3.1, textual mathematics is often intermixed with symbolic expressions and notations. To be precise, it is hard to find statements that contain only textual or symbolic material. Text and symbols complement each other in a way that they cannot be studied separately. For instance, in the sentence below, variables a and b are first used in equation 4.3.1 but introduced in the later part of the same sentence. So we have to interpret the second part first and then the first part.

> If $\sqrt{2}$ is rational, then the equation $a^2 = 2b^2$ (4.3.1) is soluble in positive integers a, b with $(a, b) = 1$. [Theorem 43 of figure 3 at p. 34]

Abbreviation by symbols makes an argument compact, concise, and easier to read and refer. In most of mathematical texts, symbols are encapsulated by natural language text. However it could also be the other way around sometimes:

$A \cap B = \{x : x \in A \text{ and } x \in B\}$ [Taylor 2010, p. 03]
$\bigcap \mathcal{A} = \{x : x \in A \text{ for all } A \in \mathcal{A}\}$ [Ibid.]
$\{(x, y) \in \mathbb{N}^2 \mid x \text{ and } y \text{ are coprime}\}$ [Ganesalingam 2009, p. 39]

The proportion of symbols in a mathematical text may vary from one domain to another. It may also depend on the taste and writing style of the author. For instance, one may prefer the statement: 'let x be an integer' as compared to the statement: 'let $x \in \mathbb{Z}$'; although both are commonly used. See more rephrasing in §3.3.6.2.

As regards the predicates and functions[5], [Ranta 2009b, pp. 17–18] notes that their usage is also quite flexible. In general, most of the predicates are textual; e.g. x is even, etc. However some conventional two place predicates are symbolic as well; e.g. equality $(=)$, inequalities $(>, <)$, etc. Similarly, most of 2-place functions are also textual. For instance, 'greatest prime factor', 'common multiple', etc. But some conventional two place functions are symbolic also; e.g. $+, -$, etc.

3.3.1.1 Textual Part of Mathematics

First, the objects of study in mathematics are rather abstract in nature, and belong to Platonic universe. That is why, it limits the use of pronouns to third person (cf. §3.3.4 for details regarding anaphoric pronouns). Furthermore, these abstract objects are mostly timeless and therefore the textual part of mathematics mostly contains present tense[6]. Consequently, the morphology used in mathematics is quite simple and plays limited role.

[5] We discuss them very briefly in §3.3.3

[6] An exceptions is the limited use of future tense in restatements mentioned in §3.2.4. Consider also the following two examples:
We will prove the result by Induction on n. ... [Baker 2009, p. 19]
The set of all n for which $X_1 + nA$ is an upper number contains a smallest integer, by Theorem 27; **we will** denote it by u. [Landau 1966, p. 50]

In semantics, the vocabulary and grammatical constructions used in mathematics may have different meaning and interpretation than their normal usage in a natural language. For instance the word 'increasing' in the following statement does not mean the usual meaning found in English vocabulary. Instead, its semantic meaning must have been defined.

> If f is a differentiable function, then if f is **increasing**, $f'(x) > 0$ for every x. [Ranta 1996, p. 231]

3.3.1.2 Symbolic Part of Mathematics

Mathematics is full of symbols and notations ranging from simple ones (e.g. $x + y$, $\sqrt{2}$, etc) to the complex ones (e.g. $\lim_{n\to\infty} \int_n^b f(x)\ dx$, $\prod_{n=1}^{k} \frac{n}{n-1}$, etc). However these visually complex symbols are easily representable in linear typographic languages such as LaTeX[Lamport 1986].

Both symbols and notations can be read aloud as if they are written in natural language. For instance, '$x + y = 10$' could be read as 'the sum of x and y is 10' or 'x plus y equals 10'. Similarly, the statement mentioned in Theorem 1.17 of figure 3.3 on page 34:

> 'Since $\gcd(p, a) \mid p, \ldots$' [Baker 2009, p. 11]

could be read aloud as: 'Since the greatest common divisor of p and a divides p, ...'. It suggests that sometimes, a symbolic construction can play the same grammatical role as its constituent natural language text. Following are some of its characteristics:

Notational collision: Symbolic mathematics is ambiguous in nature. It is based on the collective intuition and wisdom of mathematicians. It tries to use the same few notations over and over again for concepts that share similar properties or intuitive meaning. For instance, symbol '$-$' in context of number theory subtracts two numbers, but in set theory it gives the difference of two sets (which is intuitively similar to subtraction). In other words, mathematical notations and operators are heavily overloaded which may cause notational collisions.

To solve this problem, written and unwritten conventions are heavily used. It is then the combination of intuition, context, kind and conventions that allows to understand a piece of symbolic mathematics unambiguously.

An optional use of operators: It is a special case of notational collision. Expression 'ab' evidently represents a product of variables a and b, in the following statement:

> Let p be a prime and $a, b \in \mathbb{Z}$. If $p \mid \mathbf{ab}$, then $p \mid a$ or $p \mid b$.

The symbol $(\cdot)$ or $(\times)$ is used whenever there is a danger for an expression to become ambiguous or to be interpreted wrongly. For instance, if we would like mention a product of numbers 12 and 2, instead of writing 122 we'll write $12 \cdot 2$ or 12×2. This means ab is simply a short hand for $a \cdot b$ or $a \times b$.

However, a series of concatenated symbols does not always represent a product. For instance, expression ABC contains three lexical items which represent a single triangle in statement:

Let ABC be a triangle.

Precedence and Conventions: When we write an expression from number theory such as '$x + y + z$', there are two possible interpretations: '$(x + y) + z$', '$x + (y + z)$'. Both are fine but majority of mathematicians would prefer the first to later as a convention of left associativity.

However, in an expression such as '$ab+c = d$', we must interpret it as '$((a\times b)+c) = d$'. It is so because multiplication has higher precedence than addition, and addition has higher precedence than equality.

Precedence is context dependent. For instance, as noted by [Sacerdoti Coen 2009], "equality on propositions (denoted by $=$, a notational abuse for co-implication) has precedence higher than conjunction (denoted by $\wedge$), which is higher than equality on set elements (also denoted by $=$), which is higher than meet for lattice elements (also denoted by $\wedge$). Thus $A = B \wedge P$ can be parsed either as $(A = B) \wedge P$ (a conjunction of propositions) or as $A = (B \wedge P)$ (equality of lattice elements)".

But sometimes even context is not sufficient. We need to follow some conventions to find the correct interpretation of symbolic mathematics. As stated by [Sacerdoti Coen 2009], it is more likely that $\phi^2(x)$ be interpreted as $(\phi \circ \phi)(x)$, but $\sin^2(x)$ be interpreted as $(\sin x)^2$. These conventions usually informs us about the 'kind' or 'type' of an object, which in return, helps us to give correct interpretation.

In conclusion, we need to analyze the 'type' of an object, the context in which that object occurs and general conventions to understand a piece of symbolic mathematics unambiguously.

This leads to the next section in which we shall discuss other kinds of ambiguity which occur in the language of mathematics.

3.3.2 Ambiguity

Like natural languages, ambiguity is also a serious problem for mathematical language. We have discussed ambiguity in §2.2 from a certain length. In this section we'll describe it in detail.

We start by a very common form of structural ambiguity, called **coordination ambiguity**. Consider a part of sentence taken from figure 3.3 on page 34:

If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$ and b is also even.

It is extremely ambiguous and there are many interpretations possible for it. First, we need to solve the meaning of comma (,) used between two equations. Does it mean 'and'? as shown below:

If $a = 2c$, then $4c^2 = 2b^2$ and $2c^2 = b^2$ and b is also even.

If the authors would have intended 'and', they must have used it explicitly, as they have done in other part of the same sentence.

Does it mean that the second equation is a consequence of the first one? This view seems to be in accordance with the logical structure of arguments presented in it. We may have several ways to show this consequence in textual form. Two of them are following:

1. If $a = 2c$, then $4c^2 = 2b^2$ which implies that $2c^2 = b^2$ and b is also even.
2. If $a = 2c$, then $4c^2 = 2b^2$ **and** if $4c^2 = 2b^2$, then $2c^2 = b^2$ **and** b is also even.

However, these sentences are still ambiguous. We give below some of possible interpretations for the first sentence only. Priority to interpret (i.e. precedence) is shown with brackets.

- Two statements joined by 'and':
  $\underbrace{\underbrace{\text{If } a = 2c\text{, then } \underbrace{4c^2 = 2b^2 \text{ which implies that } 2c^2 = b^2}}\text{ and } b \text{ is also even.}}$
- A conditional statement whose conclusion part contains two textual statements joined by 'and':
  $\underbrace{\text{If } a = 2c\text{, then } \underbrace{\underbrace{4c^2 = 2b^2 \text{ which implies that } 2c^2 = b^2}\text{ and } b \text{ is also even.}}}$
- A conditional statement whose conclusion part contains a statement ($2c^2 = b^2$ and b is also even) which is a consequence of $4c^2 = 2b^2$:
  $\underbrace{\text{If } a = 2c\text{, then } \underbrace{4c^2 = 2b^2 \text{ which implies that } \underbrace{2c^2 = b^2 \text{and } b \text{ is also even.}}}}$
- And so on ...

As we have seen that the meaning of comma in '$4c^2 = 2b^2$, $2c^2 = b^2$' can only be understood if we look at the logical structure of this proof as well as the whole statement carefully. **It demonstrates the fact that we need an analysis which is not only linguistic but also logical.**

Consider two more statements given below. Both are ambiguous and their interpretations are discussed in §3.3.3.2 and §3.3.3.3 respectively.

Let $a \mid n$ **and** $b \mid n$ **and** choose $r, s \in \mathbb{Z}$ so that $n = ra = sb$.

[...] there are x and y such that $xa + yp = 1$ **or** $xab + ypb = b$.

Sometimes the coordination ambiguity is solved with the conventions. For instance, 'A or B and C' usually means 'A or (B and C)', 'A and B implies A or C' usually means '(A and B) implies (A or C)' and 'it is not the case that A and B or C', usually means 'it is not the case that ((A and B) or C)'. In fact this order of precedence corresponds to logical connectives. The words used for logical connectives sometimes may have different meaning. In that case, exception to such conventions usually occurs. See more details in §3.3.3.

As regards the **quantifier scope ambiguity**, it normally occurs in statements as following:

Some point lies on every line. [Ranta 1994]

Every element of some set of natural numbers is prime. [Ganesalingam 2009]

[Ranta 1994] also notes that quantifier scope ambiguity does not occur in mathematics because a strong convention of "translating quantifiers in the same order as they appear" is used. However, reading some of the textbooks reveals that statement of this kind (i.e. having implicit quantification) is not very common. It is because mathematicians usually prefer to use variables to make it more precise and free of ambiguity.

[Ganesalingam 2009] gives an account of **word sense ambiguity** in mathematical texts. He gives examples of two kind. The first is *polysemy*, whose examples could be 'prime number', 'prime field', 'prime manifolds', etc, having an intuitive connection with the adjective prime. In contrast, the second is *homonymy*, whose examples could be: 'normal subgroups', 'normal polynomials', etc, having nothing in common.

And finally, **structural ambiguity** which occurs in proofs having multiple cases, as shown in §3.2.4.1. We have seen some patterns which may help us to disambiguate. But these patterns are often not so obvious, specially when cases are introduced informally. For instance, in pattern such as 'either A or B', 'if-then-otherwise', etc, linguistic information can hardly disambiguate the branches. We also need information about the logical structure and line of reasoning. Worse, there are so many ways to write such proofs that it requires a lot of work to disambiguate between different patterns, especially when we have nested sub-proofs.

3.3.3 Statements and Logical Operators

As mentioned briefly in §3.1, mathematical statements mainly represent atomic facts. The atomic facts are formed by predicates[7] and functions. They are also formed by relations, properties and operations (sometimes). For instance, in elementary number theory we may have relations such as 'equality', 'inequality', etc; we may have properties such as 'positiveness', 'evenness', etc; and we may have operations such as 'squaring both sides', 'dividing the equation by x', etc.

These facts are normally glued together by various logical connectives and quantifiers to form bigger statements. They are mostly used in their textual form[8]. We describe them below:

3.3.3.1 Negation ($\neg$):

Negation can be represented by many clue words or phrases such as: 'x is **not** positive', '**it is not the case that** x is positive', etc. However, "not" is mostly used for simple predicates; whereas, "it is not the case that" is mostly used for complex statements.

3.3.3.2 Conjunction ($\wedge$):

Conjunction is represented by 'and'. For instance, in proposition: 'x, y and z are positive and even', it glues together six propositions as shown below:

$$\underbrace{\underbrace{\underbrace{\text{positive}(x) \wedge \text{even}(x)} \wedge \underbrace{\text{positive}(y) \wedge \text{even}(y)}} \wedge \underbrace{\text{positive}(z) \wedge \text{even}(z)}}$$

[7] Predicates are special case of functions in which a function can only return a boolean i.e. true or false.

[8] The exception to this rule is the logical systems, which rejects the use natural language in favor of formal languages.

We have interpreted from left to right. But we can also do the opposite (i.e. right to left):

$$\underbrace{\underbrace{\text{positive}(x) \wedge \text{even}(x)} \wedge \underbrace{\underbrace{\text{positive}(y) \wedge \text{even}(y)} \wedge \underbrace{\text{positive}(z) \wedge \text{even}(z)}}}$$

Conjunction may also occur between statements. Consider the following statement taken from Lemma 1.11 of figure 3.4 on page 35.

> Let $a \mid n$ **and** $b \mid n$ **and** choose $r, s \in \mathbb{Z}$ so that $n = ra = sb$.

First of all, this sentence is ambiguous (cf. bullet 3.3.2 on page 43 in §2.2 and §9.2.1). Only a careful analysis of the symbolic parts reveals that the first conjunction appears in it has lower precedence than the keyword the 'Let', and should be treated as shown below[9]:

> Let $\underbrace{\text{divide}(a, n) \wedge \text{divide}(b, n)}$ **and** choose $r, s \in \mathbb{Z}$ so that $n = ra = sb$.

For the second, 'and' in the statement, we have two options. First, we can treat it as a boolean conjunction as we did for the previous one. Second, we can also interpret it as a sequential composition operator[10]. It is because it appears between two sub-statements. These interpretations depend on the formalism in which we want to translate it. For instance, it will be translated as boolean '$\wedge$' in first order logic, as shown below (for description of $\forall$ and $\exists$, see §3.3.3.6 on page 46):

$$\forall a, b, n(\underbrace{\underbrace{\ldots\ \underbrace{\text{divide}(a, n) \wedge \text{divide}(b, n)}} \wedge \exists r, s \in \mathbb{Z}(\underbrace{\underbrace{n = ra} = sb})})$$

But in MathAbs (cf. §2.3 and §4), we translate it in two commands (assume A deduce B, where A is the first sub-statement and B is the second) as shown below:

> let $(a, b, n \in \text{Their Corresponding Types})$ $\ldots$ assume $\text{divide}(a, n) \wedge \text{divide}(b, n)$ deduce $\exists r, s \in \mathbb{Z}(n = ra = sb)$ $\bullet$

3.3.3.3 Disjunction ($\vee$):

It is represented by 'or' in natural language. e.g. x is positive or negative, etc. This example statement can only be false if both of the two statements (i.e. x is positive, x is negative) are false. This example can be represented as:

> $\text{positive}(x) \vee \text{negative}(x)$

In contrast, if we add a clue word 'either' forming: 'either x is positive or negative'. Then it means following:

$$\underbrace{(\text{positive}(x) \wedge \neg\text{negative}(x))} \vee \underbrace{(\text{negative}(x) \wedge \neg\text{positive}(x))}$$

[9] We discuss such interdependence of text and symbols in §3.3.1

[10] In terms of C/C++/Java, instead of conjunction (&&), it could be considered as semicolon.

The use of natural language in mathematics could be confusing at many occasions. For instance, consider the disjunction that appears in the following statement:

> [...] there are x and y such that $xa + yp = 1$ or $xab + ypb = b$.
> [Hardy & Wright 1975, p. 21]

Again, a careful analysis of symbolic parts (equations: $xa + yp = 1$, $xab + ypb = b$) reveals that the first key phrase: 'there are x and y such that', has more precedence than the disjunction 'or'. Therefore, we must interpret it as:

$$\underbrace{\text{there are } x \text{ and } y \text{ such that } \underbrace{(xa + yp = 1) \text{ or } (xab + ypb = b)}}.$$

Further investigation also reveals that we can obtain the second equation from the first one if we multiply b on its both sides. It means that this disjunction should be taken as a consequence (i.e. implication, given below). Then, why Hardy & Wright uses 'or' instead of an implication (e.g. 'which implies') in this proof?

We can think of three reasons: First, the first equation is given only for the purpose of explanation to an unacquainted reader. Therefore, Hardy & Wright use 'or' in its informal sense, where it does not mean logical disjunction ($\vee$). This leads to the second reason: "or" in natural language could be used as a substitute to the phrase "in other words". Third, treating 'or' as a logical disjunction ($\vee$) does not invalidate these equations. It is so because in the case when '$A \Rightarrow B$' is true, '$A \vee B$' becomes equivalent to B.

Finally, pattern such as 'either A or B ...' in a proof may also trigger *proof by case* method, where A and B are arbitrary statements. (cf. §3.2.4.1).

3.3.3.4 Implication ($\Rightarrow$):

Implication is represented by conditionals in mathematical texts. The most common pattern is: 'if A then B'. However, other patterns such as: 'B only if A', 'since, A then B', 'because A, B', 'by the fact that A, B', etc, also represent implication. We can translate all these patterns to the logical formula: '$A \Rightarrow B$'.

In contrast, patterns such as 'if-then', 'if-then-otherwise' or 'if-otherwise' in proofs may also trigger *proof by case* method (cf. §3.2.4.1).

3.3.3.5 Biconditional ($\Leftrightarrow$):

The formula '$A \Leftrightarrow B$' is logically equivalent to the formula '$(A \Rightarrow B) \wedge (B \Rightarrow A)$'. In textual form the patterns 'if and only if' or 'iff' represent it.

3.3.3.6 Quantifiers:

There are two kinds of quantifiers: *universal* and *existential*.

Universal Quantifiers: In mathematical language, universal quantification ($\forall$) of variables is presented in two ways: *explicitly* or *implicitly*. Explicit universal quantification is often represented by keywords: 'for all', 'for every', 'for any', 'let', 'assume that', 'suppose that', etc. We again give some examples from those which are already introduced in this chapter:

> **1.2. Axiom of Pairing. For any** a and b there exists a set $\{a, b\}$ that contains exactly a and b. [Jech 2000, p. 12]

> **Theorem 1.17 (Euclid's Lemma). Let** p be a prime **and** $a, b \in \mathbb{Z}$. [Baker 2009, p. 11]

> **Lemma 1.11. Suppose that** $a, b \in \mathbb{N}_0$ are coprime **and** $n \in \mathbb{Z}$. [Baker 2009, p. 10]

> **Let** n be an **arbitrary** positive integer. Assume that $P(k)$ is true **for all** positive integers $k < n$. [Erickson & Heeren 2007, p. 18]

Universal quantification may also be represented by articles (a, an, the) and numbers as shown below:

> **A.I.5 Definition. The** integers $m, n = 0$ are relatively prime if $(m, n) = 1$, i.e., if their only common factors are 1. [Hackman 2007, p. 03]

> Given **two** integers $m, n \in \mathbb{Z}$ we say that m divides n [...] [Baker 2009, p. 03]

The following example is worth noting because it demonstrates an important fact: we can seemingly replace the variables by constants as we do for variable a below ($a = 2$). It is called 'universal instantiation'.

> **THEOREM 2–2. Every** integer $a > 1$ can be represented as a product of one or more primes.
> *Proof*: The theorem is true for $a = 2$. [...] [LeVeque 1965]

A statement or expression without any explicit universal quantification in discourse is always implicitly quantified as universal (also noted by [Peters & Westerstahl 2006, p. 34]). For instance, algebraic equalities as shown below, have implicit universal quantifiers at the front:

> **Theorem.** $x + (y + z) = (x + y) + z$.

Similarly, in the following statements variable c is not introduced, and therefore, it is implicitly quantified as universal.

> If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$ and b is also even, [...] [Hardy & Wright 1975, pp. 39–40]

It is worth noting that when a variable is universally quantified in a statement, it remains available in the subsequent sentences of that block. For instance, when we say:

> Let x be an integer. $S_1, S_2, \ldots, S_n$ (S is a sentence)

We interpret it as following, where $\overline{S}$ is an interpretation of S, and because each sentence is a logical consequence of its previous sentence in a discourse, we add implications between their interpretations.

$$\forall x \in \mathbb{Z}(\ (((\overline{S}_1 \Rightarrow \overline{S}_2) \Rightarrow \ldots) \Rightarrow \overline{S}_n)\ )$$

Existential Quantifiers: Existential quantifier ($\exists$) is often represented by 'for some', 'there exists', 'there is', etc, in natural language. For instance, consider the following examples:

> Given two integers $m, n \in \mathbb{Z}$ we say that m divides n and write $m \mid n$ if **there is** an integer $k \in \mathbb{Z}$ such that $n = km$;[...] [Baker 2009, p. 03]

> Suppose n is odd. Then $n = 2k + 1$ **for some** integer k.

There is a second kind of existential quantifier called 'unique existential ($\exists!$)', for which we add phrases such as 'unique', 'one and only one', etc. Here is an example:

> Then **there are unique** natural numbers $q, r \in \mathbb{N}_0$ satisfying the two conditions $n = qd + r$ and $0 \leq r < d$. [Baker 2009, p. 02]

Similar to 'universal instantiation', we can also instantiate existentially quantified variables. The keyword: 'we take' is used for it in most cases, but sometimes no keyword is used. We give examples for both cases below:

> THEOREM 1.2 (Long Division Property). Let $n, d \in \mathbb{N}_0$ with $0 < d$. Then **there are unique** natural numbers $q, r \in \mathbb{N}_0$ satisfying the two conditions $n = qd + r$ and $0 \leq r < d$.
> PROOF. [...] Notice that if $\mathbf{r} = \mathbf{n} - \mathbf{qd}$, then
> $0 \leq r = n - qd < (q + 1)d - qd = d$.[...] [Baker 2009, pp. 02–03]

> THEOREM 1.3. [...] Then **there are unique** integers $q, r \in \mathbb{Z}$ for which $r <\mid d \mid$ and $n = qd + r$.
> PROOF. [...] If $r' = 0$ then **we take** $\mathbf{q} = -\mathbf{q}'$ and $\mathbf{r} = \mathbf{0}$. [...]
> [Baker 2009, p. 03]

Finally, It is not always the keywords that decide whether a variable be quantified universally or existentially. The logical status (i.e. whether a statement is an hypothesis, deduction or goal) also matters. For instance, in the assumption:

> We assume that there is an integer x such that [...],

the variable x should not be existentially quantified, although the keyword 'there is' appears. Instead, it should be universally quantified (i.e. $(\forall x : \mathbb{Z}(\ldots))$).

In contrast a goal (or a deduction) as following:

> Then there is an integer x such that [...],

the variable x should be existentially quantified $(\exists x : \mathbb{Z}(\ldots))$.

The rules we follow for this translation are taken from the Gentzen's sequent calculus (more precisely the left and right rules for $\forall$ and $\exists$). For more details, see §4.6 on page 72.

3.3.4 Anaphoric Pronouns

The use of anaphoric pronouns is quite frequent in the language of mathematics. In general, pronouns refer to nouns, definite descriptions, variables, symbolic expressions, statements and propositions. We can find following pronouns in mathematical texts: 'it', 'itself', 'they', 'this', 'these', 'both', 'other', 'such', 'their', etc. However this list is not meant to be exhaustive. We describe their usage below with some typical examples found in mathematical texts.

3.3.4.1 Pronouns referring to nouns, definite descriptions, variables and symbolic expressions:

First we highlight the use of 'it' with different examples. For instance, in the following statement, 'it' refers to class F.

> A class F is a function if **it** is a relation such that $(x, y) \in F$ and $(x, z) \in F$ implies $y = z$. [Jech 2000, pp. 11–12]

Similarly, in the following statement, we have two pronouns, both refer to the 'set'.

> A set is finite if **it** has n elements for some $n \in N$, and infinite if **it** is not finite. [Jech 2000, p. 14]

It is not difficult to imagine such statements as shown below, in which 'it' refer to the set '$A \cap B$' and its textual form respectively:

> The set $A \cap B$ is finite if **it** has ...
> The intersection of sets A and B is finite if **it** has ...

We may also frequently find statements in which pronoun refers to a definite description. For instance in the following statement, 'it' refers to the description 'smallest integer'.

> The set of all n for which $X_1 + nA$ is an upper number contains a smallest integer, by Theorem 27; we will denote **it** by u. [Landau 1966, p. 50]

Another usage of 'it' could be seen in the following statements. The pronoun in the first statement refers to 'prime divisor' and the pronoun in the second statement refers to 'smallest positive integer'.

1. So a must have a prime divisor, call **it** p. [Clark 2002, p. 31]
2. By the Well-Ordering Property for N, S contains a smallest positive integer, call **it** d. [Clark 2002, p. 25]

In a similar way, we now describe some typical usage of pronoun 'they'. Its usage is quite similar to 'it' with two exceptions. First, obviously, it refers to plural objects; and secondly, it normally does not refer to propositions and statements.

For instance in the first example below, 'they' refers to variables x_1, x_2, x_3, x_4. While in the second example, 'they' refers to variables t, t'. We can also note the anaphoric usage of phrase 'original pair' which refers back to the pair given in the theorem of this proof.

> ...If m_0 is even, then $x_1+x_2+x_3+x_4$ is even and so either (i) x_1, x_2, x_3, x_4 are all even, or (ii) **they** are all odd, or (iii) two are even and two are odd. In the last case, let us suppose that x_1, x_2 are even and x_3, x_q are odd. Then in all three cases ... [Hardy & Wright 1975, p. 302]

> Theorem 1.12 (The Chinese Remainder Theorem). Suppose $n_1, n_2 \in \mathbb{Z}+$ are coprime and $b_1, b_2 \in \mathbb{Z}$. Then the pair of simultaneous congruences $x \equiv_{n_1} b_1, x \equiv_{n_2} b_2$, [...]
> Proof. [...] To prove uniqueness modulo $n_1 n_2$, note that if t, t' are both solutions to the **original pair** of simultaneous congruences then **they** satisfy the pair of congruences $t' \equiv_{n_1} t, t' \equiv_{n_2} t$. [Baker 2009, p. 10]

In the following example, we highlight the use of 'one' and 'other' along with 'they' which refers to two integers.

> Two integers have the same parity if **they** are both even or if **they** are both odd. **They** have different parity if **one** is even and the **other** is odd. [Clark 2001, p. 27]

The use of reflexive pronoun 'itself' is also common as shown below, where 'itself' refers to π.

> Since $\lambda \subset B$, π maps every subset of λ onto **itself**, and so $P(\lambda) \cap M = P(\lambda) \cap B$. [Jech 2000, p. 340]

The statements given below, highlight the use of 'such' in mathematical texts. Here, quite evidently 'such integers' refers to such integers that are greater than 1 and has no prime divisor. Similarly the second 'such' appearing in the last line refers to the smallest element of S.

> Assume there is some integer $n > 1$ which has no prime divisor. Let S denote the set of all **such integers**. By the Well-Ordering Property there is a smallest **such** integer, call it m. [Clark 2002, p. 31]

Now we give a few examples, highlighting the use of demonstrative pronouns. For instance, in the following example we can see the use of 'these' among other pronouns, where 'these' refers to u, v.

> If $u \neq v$, then using the Prime Ideal Theorem, one can find an ultrafilter p on B containing **one** of **these two** elements but not the **other**. [Jech 2000, p. 81]

While in the following example, 'this' refers to the number $g^{(p-1)/2}$.

> Notice that the power $g^{(p-1)/2}$ satisfies $(g^{(p-1)/2})^2 \equiv 1$. Since **this** number is not congruent to 1 modulo p, Proposition 1.23 implies that $g^{(p-1)/2} \equiv_p -1$. [Baker 2009, p. 15]

An interesting passage to quote would be the following which is full of different kinds of anaphora. First of all, it is a *proof by case* which is described in §3.2.4.1. In the first line, the use of 'this' (which could also be replaced by 'it') refers to the case condition $(a, n) > 1$. More on pronouns referring to statements and propositions will appear in the next section. Another instance of 'this' appears in the third line which refers to the current case. Further, we can see that use of 'both' and 'their' which refer to p, q.

> Next consider the case where $(a, n) > 1$. **This** means that a is divisible by p or q. If a is divisible by **both**, **it** is divisible by **their** product n, and the result is trivial in **this case**. [...] Consider the difference $b = a^{k\phi(n)+1} - a$. Under our assumption **it** is trivially divisible by p.
>
> [Hackman 2007, p. 33]

Sometimes it is not easy to resolve anaphora as the anaphoric statements which look similar may refer to different objects. The only way to resolve them would be to go deeper and look inside the statements, and then, take the decision based on their semantics. Worse, there could be a lot of cases (if not infinitely many) to be captured in such deep analysis. Anyways, in the first statement below, 'it' refers to a. In contrast, in second statement 'it' refers to b.

> Because $a \mid p$, **it** is a either prime or 1.
> Since $2 \mid b$, we conclude that **it** is even.

Finally, consider two more examples which are more convincing:

> If $a \mid b$ then **it** is prime or one. ('it' refers to a)
> If $a \mid b$ then **it** is not prime nor one. ('it' refers to b)

3.3.4.2 Pronouns referring to statements and propositions:

Pronouns referring to statements and propositions mostly occur in proofs. In a proof each statement is mostly a consequence of previously stated statements. To show this dependence, key phrases such as: 'it shows that', 'it is easy to see that', 'it follows that', 'it is trivial', 'this means', 'it means that', etc, are used. In all such key phrases, 'it', 'this', or 'these' mostly refer to the previous statement(s), condition(s) or proposition(s).

For instance, in the examples below, in the first example 'it' refers to the condition $n < e$. While in the second example, 'it' refers to the fact that 'α is transitive'.

> Choosing $n < e$, **it** shows that $(a^{p^e} - 1)/(a - 1) \equiv 0 (\bmod\ p^e)$, hence the period cannot be a smaller power of p. [Hackman 2007, p. 90]

> If $\alpha \subset \beta$, let γ be the least element of the set $\beta - \alpha$. Since α is transitive, **it** follows that α is the initial segment of β given by γ.
>
> [Jech 2000, p. 19]

Similarly in the first example below, 'this' refers to the fact that $q > q'$. While, in the second example, 'each of these' refers to both $x > y$ and $y < x$. Finally, in the third example, 'this' and 'it' refer to previous statements.

> If $q > q'$, **this** implies $d \leq (q - q')d$,... [Baker 2009, p. 03]

> **Theorem 11:** If $x > y$ then $y < x$.
> **Proof. Each of these** means that $x = y + u$ for some suitable u.
>
> [Landau 1966, p. 09]

> Then by (4.2) we have $n \leq x < n + 1$. **This** gives immediately that $\lfloor x \rfloor \leq x$, as already noted above. **It** also gives $x < n + 1$ which implies that $x - 1 < n$, that is, $x - 1 < \lfloor x \rfloor$. [Hackman 2007, p. 14]

Pronouns may also refer to algorithmic procedure in some proofs. Here is an example text.

> **Proof.** We begin by setting $\gamma_0 = \gamma$ and $c_0 = [\gamma_0]$. Then if
> $$\gamma_1 = \frac{1}{\gamma_0 - c_0},$$
> we can define $c_1 = [\gamma_1]$. Continuing in **this way**, we can inductively define sequences of real numbers γ_n and integers c_n satisfying ...
>
> [Baker 2009, p. 20]

Further, in the following example text, 'this' refers to a procedure which is informally given in the previous sentence.

> The idea is to divide up G into disjoint subsets of size H. We do **this** by defining for each $x \in G$ the *left coset of* x with respect to H, ...
>
> [Baker 2009, p. 37]

Not all occurrences of pronouns are anaphoric: However, sometimes their usage does not necessarily refer to anything. Instead, they only appear as glue words whose sole purpose is to just connect two statements or part of speech. We give a few examples below.

> Let $f = \{(x, y) \in W_1 \times W_2 : W_1(x)$ is isomorphic to $W_2(y)\}$. Using Lemma 2.7, **it** is easy to see that f is a one-to-one function.
>
> [Jech 2000, p. 18]

> **Theorem.** For positive integers m, n
> $$[m, n] = \frac{mn}{(m, n)}.$$
> **Proof. It** is enough to compare the multiplicities of any prime p.
>
> [Hackman 2007, p. 47]

3.3.5 Anaphoric References

Anaphoric references are quite frequent in mathematical texts. We can roughly divide them in two: explicit and implicit as described below.

3.3.5.1 Explicit:

Sometimes symbolic expressions such as equations are tagged with references. It enables us to refer back to them whenever needed in some other part of mathematical texts. We present two examples in which equations are first tagged and then later referenced.

> If $\sqrt{2}$ is rational, then the equation $a^2 = 2b^2$ **(4.3.1)** is soluble in positive integers a, b with $\gcd(a, b) = 1$. [...]
> (b) *Second proof.* It follows from **(4.3.1)** that $b \mid a^2$, and a fortiori that $p \mid a^2$ for any prime factor p of b. [Hardy & Wright 1975, p. 40]

> Moreover, this rule must satisfy the condition
> $$x = y \Longrightarrow f(x) = f(y) \qquad (1.1)$$
> [...] Equality of ordered pairs is defined by the rule
> $$a = c \text{ and } b = d \Longleftrightarrow (a, b) = (c, d). \qquad (1.2)$$
> [...] Now implication (1.1) becomes
> $$(a, b) = (c, d) \Longrightarrow a * b = c * d. \qquad (1.3)$$
> From (1.2) and (1.3) we obtain
> $$a = c \text{ and } b = d \Longrightarrow a * b = c * d. \qquad (1.4)$$
> [...] [Clark 2001, pp. 01–02]

In a similar way, sometimes textual statements may also be tagged to refer later. An example is the following theorem in which the hypothesis is tagged and referred later in its proof.

> **Theorem 2.6.** [...]
> (a) [...]
> (b) Whenever $c \mid a$ and $c \mid b$, then $c \mid d$.
> **Proof.** [...] Given any common divisor c of a and b, we have $c \mid d$ from hypothesis (b) [Burton 2007, p. 24]

3.3.5.2 Implicit:

Mathematical texts are also full of implicit references. We present below some examples to highlight different kinds of implicit references.

We start with the most common operations in number theory such as 'squaring', 'dividing', 'factoring', etc, which are proceeded and succeeded by equations. For instance, in the following example, instead of writing '**squaring** both sides', author mentions '**squaring**' only. However the proceeding equation in the sentence indicates that there must be an equation before it on which this operation is applied.

> Suppose that $\sqrt{p} = \frac{a}{b}$ for integers a, b. We can assume that $\gcd(a, b) = 1$ since common factors can be canceled. Then on **squaring** we have $p = \frac{a^2}{b^2}$ and hence $a^2 = pb^2$. [Baker 2009, p. 13]

In the example below, 'hypothesis' refers to the assumption given in the theorem.

> **Theorem 1:** If a/b, then $a/-b$, $-a/b$, $-a/-b$, $\mid a \mid / \mid b \mid$.
> *Proof:* By hypothesis we have $b = qa$; [...] [Landau 1958, p. 11]

We may also have patterns such as:

> '(first| second | ... | last) case',
> '(first|last) (hypothesis | inequality | formula | deduction | statement)',
> 'each side of this equation', etc.

Following are a few random examples without any further explanation.

Raise both sides of this **last equation** to the pth power and expand to obtain $r^{p^{k-1}(p-1)} = (1 + ap^{k-1})^p \equiv 1 + ap^k(\text{mod} p^{k+1})$
[Burton 2007, p. 161]

From the **last theorem**, it is known that $P(a) = P(b)(\text{mod } n)$.
[Burton 2007, p. 72]

Letting $QR \equiv 1(\bmod M)$, and multiplying the **last congruence** by R^n, we get ... [Hackman 2007, p. 361]

Otherwise, the power of 2, if there is any, supplies a factor of 1 to the **last formula** if $l = 1$; ... [Landau 1958, p. 64]

Proof. Let us multiply the **first** congruence of the system by d, the **second** congruence by b, and subtract the **lower result** from the **upper**. These calculations yield ... [Burton 2007, p. 81]

$L \vDash$ every $p \in P_\alpha$ forces $\exists x(A(x) \wedge \parallel x \parallel = \alpha)$. (25.20)
[...]$L \vDash (p \Vdash \varphi)$if and only if for every L-generic $G \ni p, L[G] \Vdash \varphi$.(25.21)
[...]But in view of (25.21) this **last statement** is equivalent to (25.20).
[Jech 2000, p. 492]

[...] it is possible to find integers x and y such that $d = ax + by$. Upon dividing **each side of this equation** by d, we obtain the expression

$$1 = (\frac{a}{d})x + (\frac{b}{d})y$$

[Burton 2007, p. 23]

3.3.6 Miscellaneous Features

3.3.6.1 The Length of the Sentence

Writing short and concise is a normal practice in mathematics. However, it is also not so rare to find long sentences. To get the feeling, consider a few example sentences without any further explanation.

1. If a and b are relatively prime so that $\gcd(a, b) = 1$, then Theorem 2.3 guarantees the existence of integers x and y satisfying $1 = ax + by$. [Burton 2007, p. 23]

2. Suppose that $\sqrt{2}$ is a rational number, so $\sqrt{2} = \frac{a}{b}$ where a and b are non-zero integers with no common factor (definition of a rational number). [Wikipedia 2010]

3. By the definition of rational numbers, we can assume that $\sqrt{2} = \frac{a}{b}$ where a and b are non-zero integers with no common factor.

4. Then by definition of the rationals, there exist non-zero integers p and q such that $\frac{p}{q} = \sqrt{2}$, where p and q do not have common divisors (in other words, p/q is in lowest terms). [Chan 2010, p. 03].

5. If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$, and b is also even, contrary to the hypothesis that $(a, b) = 1$. [Hardy & Wright 1975, p. 40]

6. If g is a primitive root modulo p^2, where p is an odd prime, then it is also one modulo every power $p^k, k \geq 2$. [Hackman 2007, p. 79]

Grammar modeling of such long sentences is an extra work if one parses general patterns instead of tailor made sentences. It may also increases the complexity and parsing time of the grammar. Furthermore, from an authors point of view who is writing in a controlled language, it may be difficult to keep track of the available grammar[11].

3.3.6.2 Rephrasing

Being able to rephrase sentences is a basic yet very powerful capability of natural languages[12], which the language of mathematics also inherits. It can, not only rephrase a textual statement to textual statement (could be more than one) but also rephrase it to symbolic expressions and notations.

One of the most common pattern we found in mathematics text is conditionals which can easily be rephrase as 'since ... then ...', 'because ..., we conclude that ...' or 'let/suppose ... then ...' patterns. We have already mentioned the third pattern in §3.2.4 on page 36. We also quote the same example given there.

> If $\sqrt{2}$ is rational, then the equation $a^2 = 2b^2$ (4.3.1) is soluble in positive integers a, b with $\gcd(a, b) = 1$. ([Hardy & Wright 1975]:40)

Keeping the same arguments and line of reasoning, it can easily be rephrased in many ways. Three possible rephrased statements could be following:

1. Let a and b be two positive integers. Assuming that $\sqrt{2}$ is rational, the equation $a^2 = 2b^2$ (4.3.1) is soluble with a condition that the greatest common divisor of a and b is 1.

2. Assuming that $\sqrt{2}$ is rational, we conclude that the equation $a^2 = 2b^2$ (4.3.1) is soluble in positive integers a, b with $\gcd(a, b) = 1$.

3. Suppose that $\sqrt{2}$ is rational. Then the equation $a^2 = 2b^2$ (4.3.1) is soluble in positive integers a, b with $\gcd(a, b) = 1$.

We have discussed the intermixing of text and symbols in §3.3.1. Symbols in textual statements are sometimes embedded in a way that they could be replaced by natural language part of speech (but on the expense of compactness) and vice versa. For instance in the following example,

> If $c \neq 0$ and $ca = cb$, then $a = b$.

clauses such as 'c is not equal to 0', 'ca is equal to cb' (or alternatively 'ca and cb are equal'), 'a is equal to b' respectively can easily replace symbolic equations; hence rephrasing the statement. Further, as given in §3.3.1, one may rephrase a statement such as 'let x be an integer' to a statement such as 'let $x \in \mathbb{Z}$' and vice versa.

[11] In general, this point could be raised for controlled languages as a whole by asking a question: 'Is it easy to write in a controlled language?' We contribute our few cents regarding this in the Introduction chapter.

[12] In a limited way, it is also a property of any formal language which distinguishes between syntax and semantics.

3.3.6.3 Distributive and Collective Readings

Distributive and collective reading are also common in mathematical texts. For example, the statements such as 'x, y are positive' and 'x, y are equal' are distributive and collective respectively.

Some collective readings could be ambiguous. Consider the statement 'a,b,c are distinct'. Are variables a, b, c pair-wise distinct or just some of them distinct?

Further, because collective readings require their subjects to be plural, statements such as 'x is equal' never occur.

3.3.6.4 An optional Introduction of Variables

Normally a variable in first introduced in previous or the same sentence and then used. However, variables are not introduced and directly used Sometimes. For instance, c is used directly without any introduction in the third sentence of Theorem 43 in figure 3.3.

3.4 List of Mathematical texts used for analysis

1. Set Theory, 3rd edition by Thomas Jech. [Jech 2000]
2. Undergraduate Analysis, 2nd edition by Serge Lang. [Lang 1997]
3. Foundations of Analysis, 3rd edition by Edmund Landau. [Landau 1966]
4. Basic analysis - Introduction to Real Analysis. [Lebl 2010]
5. Algebra and Number Theory by Andrew Baker. [Baker 2009]
6. An Introduction to the Theory of Numbers, 4th edition by Hardy and Wright. [Hardy & Wright 1975]
7. Elementary Number Theory by Edmund Landau. [Landau 1958]
8. Elementary Number Theory by Peter Hackman. [Hackman 2007]
9. Elementary Number Theory, 6th edition by David M. Burton. [Burton 2007]
10. Elementary Number Theory by Gareth Jones and J. Mary Jones. [Jones & Jones 2007]
11. Elementary Number Theory by W. Edwin Clark. [Clark 2002]
12. Elementary Abstract Algebra by W. Edwin Clark. [Clark 2001]

Chapter 4

MathAbs - The Abstract Mathematical Language

Contents

4.1 Introduction **57**

4.2 Why MathAbs **58**

4.2.1 Origin of MathAbs 59

4.3 Syntax of MathAbs **59**

4.3.1 MathAbs and Extensibility 64

4.4 The MathAbs Semantics **64**

4.5 Completeness **68**

4.6 Translating Textual Quantifiers to MathAbs **72**

4.6.1 Simplifying MathAbs 76

4.7 Proof checking **79**

4.7.1 Proof checking using Automated Theorem Provers 79

4.7.2 Proof Checking Limitations and Future Directions 84

4.8 Conclusion **85**

4.9 Appendix **85**

4.9.1 An Example: 85

4.1 Introduction

MathAbs (**Math**ematical **Abs**tract language) is a system independent formal language for mathematical texts that represents its semantics, and preserves its logical and reasoning structure. By system independent, we mean that its formalism is not based on any specific logic, theory or proof assistant.

Why must such a formal language be independent of any logic, theory or prover? It is because different logics and theories[1] have both merits and demerits for the formalization of mathematics. Since we intend to represent the language of mathematics "as it is", it is better to postpone such decisions till we do proof checking.

MathAbs is intended only for machine manipulation and it is a part of MathNat. Here, MathAbs is used as an intermediary between the natural language of the mathematician and the formal language of the logician.

An overview is already given in §2.3. The aim of this chapter is to give a detailed account of MathAbs. We give its motivation in §4.2, syntax in §4.3, formal definition

[1]Logics and theories such as first-order logic, higher-order logic, predicative logic, impredicative logic, category theory, different versions of set theories, different versions of type theories, etc.

and semantics in §4.4, completeness in §4.5 and proof checking in §4.7. Assigning the right quantification to variables is a very important task which is given in §4.6. A typical mathematical text and its MathAbs version is given in figure 4.1.

4.2 Why MathAbs

The use of an intermediate language between the language of mathematics and theorem provers is not new. AutoMath [de Bruijn 1994] of N.d. Bruijn, is one of the pioneering works in which a very restricted proof language was proposed. Similarly, a recent framework MathLang [Kamareddine & Wells 2008], proposes an intermediate language named Core Grammatical aspect (CGa) which is a predecessor of Weak Type Theory (WTT) and similar to AutoMath. MathLang supports the manual annotation of mathematical texts, instead of automatic formalization by parsing (as MathNat does). Once the annotation is done by the author, a number of transformations to the annotated text is automatically performed for automatic but partial verification in Mizar and Isabelle.

Like these languages, MathAbs acts as a mediator between the language of mathematics and the proof checking systems. However, unlike these languages, the content representation of MathAbs is very close to the textual proofs. Furthermore, MathAbs has a clear semantics for axioms, definitions, theorems and proofs. MathAbs follows arbitrary rules of reasoning, having no dependence on any particular theory or logical system. It attempts to represent the language of mathematics which may have reasoning gaps faithfully. By "faithfully", we mean that the formal version of mathematics is as close as possible to the intentions expressed in the mathematical text. This language is intended to be simpler and non-ambiguous than the language of mathematics found in the published literature, so that we can interpret it strictly.

The proof language of MathAbs (which is its main aspect) is constructed by looking at proofs in natural language and observing that mathematicians do not give the name of the logical rules they use (except for induction and absurdity reasoning). Instead they usually explain how the *context* is evolving, i.e. what the new hypotheses and the current goals are; what justifications are given as evidence, etc. By "justifications", we mean those explanations or details whose purpose is to guide a reader about the reasoning steps.

It is important to note that a wrong proof can still be represented in MathAbs. It is so because, MathAbs is only a representation language, therefore, no validation or proof checking is performed directly in MathAbs as opposite to proof systems like natural deduction. However, due to the semantics of MathAbs §4.4, we can manually trace a wrong proof step, and therefore, there are more chances to know in advance that a certain proof will succeed or fail by the automated theorem provers. It allows to define a term "valid" MathAbs. It is a MathAbs proof containing no invalid proof step according to the semantics of MathAbs.

Note that, there is no question of considering semantic markup languages such as ActiveMath, OpenMath, OMDoc, etc for this task. Generally, they are low level and have no adequate language (with semantics) to deal with the reasoning expressed in textual proofs. Therefore, the content representation of these languages is inadequate for the textual proofs. See [Maarek 2007, pp. 41–43, 53–58] for a detailed account on the shortcomings of these languages.

4.2.1 Origin of MathAbs

MathAbs is an extension of a proof language: new_command. It was designed as an intermediate formalism between natural language proofs and the proof assistant PhoX [Raffalli 2005] in the DemoNat project[2] [Thévenon 2005, Thévenon 2006].

In DemoNat, the work was mainly focused on the realization and development of a proof assistant that can utilize justifications found in informal proofs, to reduce it's search space. However, this proof assistant was only developed partially, and therefore, it is not beneficial for us in the context of the verification of MathAbs (cf. §4.7). Although an automatic translation from informal proofs to new_command was proposed but it has never been implemented. In this sense, MathNat is the successor of DemoNat.

The proof language new_command includes some of the standard commands of PhoX and supports a proposition (formula) as a theorem statement. We adapt it after removing such dependencies and simplify the proof language by reducing the constructs to minimal. Further, we add a language of definition and theorem. We also provide the semantics of MathAbs for axioms, definitions, theorems and their proofs, which was not done before. We have published these new results in [Humayoun & Raffalli 2010a].

4.3 Syntax of MathAbs

MathAbs can represent theorems and their proofs along with supporting axioms and definitions. On a macro level, a MathAbs' document is a sequence of definitions, axioms and theorems with their proofs. However, analogous to the language of mathematics, the most significant part of MathAbs is the language of proof; while the definitions and axioms are probably just a fraction.

A proof in MathAbs is described as a tree of logical (meta) rules. These rules are arbitrary and do not necessarily have to be the rules of natural deduction or sequent calculus. Intuitively, at each proof step, there is an implicit active sequent with some hypotheses and one conclusion, which is being proved, and some other sequents to be proved later. The math text explains how the active sequent is modified to progress in the proof and some justifications (hints) may be given. A theorem forms the initial sequent with some hypotheses and one goal, which is then passed to the proof. While axioms and definitions are added in the initial sequent as hypotheses. The proof in figure 4.1 is a shortened version (we have merged some rules) of the semantics of the proof tree shown in figure 4.2.

Formally, a MathAbs' document is a non-empty sequence of definitions, axioms and theorems with their proofs. We use BNF notation to describe the syntax of MathAbs.

$$
\begin{array}{lcl}
\langle \mathit{Document} \rangle & ::= & \langle \mathit{Math} \rangle \;;\; \langle \mathit{Math} \rangle \;;\; \ldots \\
\langle \mathit{Math} \rangle & ::= & \mathsf{Axiom}\ \langle \mathit{Formula} \rangle \\
 & | & \mathsf{Definition}\ \langle \mathit{Definition} \rangle \\
 & | & \mathsf{Theorem}\ \langle \mathit{Theorem} \rangle \;;\; \mathsf{Proof}\ \langle \mathit{Proof} \rangle
\end{array}
$$

An Axiom is simply a formula while the other constructs are described later in this section. The language of ⟨*Formula*⟩ is arbitrary (it is a parameter of MathAbs) and subject to change from one mathematical domain to another. In first-order logic or set theory, ⟨*Formula*⟩ is simply a language for first-order formulas. However, it could be

[2] DemoNat homepage: `http://wiki.loria.fr/wiki/Demonat`

second order or higher order formulas also (according to the contents of the mathematical text).

It means, ⟨*Formula*⟩ represents mathematical statements inside various structural blocks (i.e. axiom, definition, theorem, etc); see figure 6.2 on page 109 for the supported grammar for ⟨*Formula*⟩ by MathNat. In contrast, the language of axiom, definition and proof remain the same for all mathematical domains. See §4.3.1 for more details.

Consider the following axiom as an example, which is only a statement; while its MathAbs is simply a ⟨*Formula*⟩:

Axiom 1. 1 is a natural number.
Axiom 1. $1 \in \mathrm{Nat}$

The ⟨*Definition*⟩, ⟨*Assignment*⟩ and ⟨*Ident*⟩ are defined below.

⟨*Definition*⟩	::=	⟨*Assignment*⟩ ⟨*Definition*⟩
	|	define ⟨*Formula**⟩ as ⟨*Formula*⟩

⟨*Assignment*⟩	::=	let ⟨*Ident*⟩:⟨*Type*⟩
	|	assume ⟨*Formula*⟩
⟨*Ident*⟩	::=	⟨*String*⟩

Only function names with varying number of variables as parameter (e.g. f, $f(x)$, $f(x, y)$, ...) are allowed in ⟨*Formula**⟩ as a left hand side of a definition. We will explain more when giving the semantics in §4.4.

The ⟨*Assignment*⟩ only deals with universal variables. In ⟨*Assignment*⟩, let allows to introduce a new variable in the sequent; while assume allows to add a new hypothesis to deal with conditional definitions (e.g. the definition of division requires the divider to be non zero). More details will appear later in this section, in the explanation of ⟨*Proof*⟩ language. Two examples of definitions and their MathAbs translation could be seen on lines 1–2 of figure 4.1.

The ⟨*Type*⟩ which appears in ⟨*Assignment*⟩ is also a parameter of MathAbs. In set theory, a type would simply be a set; whereas in first order logic, it would be a predicate symbol. In contrast, it also could have the linguistic characteristics: a ⟨*Type*⟩ is a set or predicate whose elements have a generic common name. e.g. Integer, Rational, etc. We keep this notion of 'type' to save the linguistic information.

However, this notion does not exactly correspond to type theory and it is not even very precise (for instance, linguistically it is not easy to decide whether *prime* is a type or just a property of numbers). It is one of the main problems if we would like to translate CLM to a typed framework such as Agda [Norell 2007a, Norell 2007b] or Coq [Team 2010].

Following are the types supported in the current implementation of MathNat. However note the use of ⟨*Ident*⟩ below. It reveals that ⟨*Type*⟩ is open to new constructs.

⟨*Type*⟩	::=	`Prop`	|	`Set`	|	`Integer`
	|	`Number`	|	`Rational`	|	`Irrational`
	|	`NoType`	|	`Nat`	|	⟨*Ident*⟩

A theorem could have zero or more assignment statements, followed by a show statement:

$$\begin{array}{lcl} \langle \textit{Theorem} \rangle & ::= & \langle \textit{Assignment} \rangle \ \langle \textit{Theorem} \rangle \\ & | & \textsf{show} \ \langle \textit{Formula} \rangle \\ & | & \bullet \ \langle \textit{Theorem} \rangle \end{array}$$

In theorems, as shown in §3.2.3 on page 33, introduction of variables and assumptions using the same language as in proofs is commonly found in mathematical texts. Therefore, in the definition of theorem shown above, ⟨*Assignment*⟩ is allowed to preserve its NL structure. With rule show in the second line, we give the formula which we need to prove. Finally, full-stop (•) marks the end of a sentence in theorem. This information is kept to make the MathAbs' theorem closer to the textual theorem. It could be useful when we translate MathAbs to first-order formulas, or to the language of various theorem provers.

Recall that we have already given three versions of the proof "irrationality of $\sqrt{2}$" in Chapter 2 (figure 2.4 on page 14). Also, in §2.3, we explained MathAbs with the help of a proof which is similar to the first and second version of proof given in figure 2.4. However, the third proof is a bit large and its MathAbs is given in §4.9. We now give another example theorem and its MathAbs in figure 4.1.

---

1. **Definition 1 (divisibility).** Let m and n be arbitrary integers with a condition that $m > 0$. Then n is said to be divisible by m if there is a number q, such that $n = q * m$.
2. **Definition 2 (evenness).** An integer n is even if it is divisible by 2.
   [. . .]
3. **Theorem.** If x and y are two even integers then $x + y$ is even.
4. **Proof.** By the definition of even numbers, we assume that $x + y = 2 * a + 2 * b$, where a and b are integers.
5. We deduce that $x + y = 2 * (a + b)$ by the last statement.
6. Therefore, $x + y$ is an even integer because it is a multiple of 2.

MathAbs:

1. Definition 1 (divisibility). let $m, n : \mathbb{Z}$ assume $m > 0$
   define $divisible(n, m)$ as $\exists q :$ `Number` $(n = q * m)$
2. Definition 2 (evenness). let $n : \mathbb{Z}$ define $even(n)$ as $divisible(n, 2)$
   [. . .]
3. Theorem 1. let $x, y : \mathbb{Z}$ assume $even(x) \wedge even(y)$ show $even(x + y)$ •
4. Proof. let $a, b : \mathbb{Z}$ assume $x + y = 2 * a + 2 * b$ by def `Even_Number` •
5. deduce $x + y = 2 * (a + b)$ by form $x + y = 2 * a + 2 * b$ •

6.1. deduce $multiple_of([x + y], 2)$†
6.2. show $even(x + y)$ by form $multiple_of([x + y], 2)$ •
6.3. trivial

---

†In line 6, first, we deduce the justification i.e. $multiple_of([x + y], 2)$, because it is not in the list of available hypothesis. Then we deduce the whole statement. However on line 5, we do not deduce the justification $(x + y = 2 * a + 2 * b)$ because it is already in the list of available hypothesis.

Figure 4.1: A typical text from elementary number theory and its MathAbs

The language of proof is substantially richer than the language of theorem and definition as shown below:

$$
\begin{array}{lll}
\langle \mathit{Proof} \rangle & ::= & \mathsf{trivial}\ \langle \mathit{Hint} \rangle \\
& | & \mathsf{show}\ \langle \mathit{Formula} \rangle\ \langle \mathit{Hint} \rangle\ \langle \mathit{Proof} \rangle \\
& | & \mathsf{deduce}\ \langle \mathit{Formula} \rangle\ \langle \mathit{Hint} \rangle\ \langle \mathit{Proof} \rangle \\
& | & \langle \mathit{Assignment} \rangle\ \langle \mathit{Hint} \rangle\ \langle \mathit{Proof} \rangle \\
& | & \{\ \langle \mathit{Proof} \rangle;\ \langle \mathit{Proof} \rangle;\ \ldots \}\ \langle \mathit{Hint} \rangle \\
& | & \mathsf{unfinished} \\
& | & \bullet\ \langle \mathit{Proof} \rangle
\end{array}
$$

Here are explanations of $\langle \mathit{Proof} \rangle$ language constructs:

1. As mentioned earlier, let allows to introduce a new variable in the sequent. For instance, "let x be an integer" is represented as "let $x : \mathbb{Z}$". Similarly, "prove that $\forall x \in \mathbb{R}\ P(x)$" is represented as "let $x : \mathbb{R}$ show $P(x)$" (see §4.6 for more details).

2. Similarly, assume allows to add a new hypothesis in the sequent. For instance, "let x be a positive even integer" is represented as "let $x : \mathbb{Z}$ assume $\mathit{positive}(x) \wedge \mathit{even}(x)$".

3. show allows to change the conclusion of the sequent. It should be understood as "it is enough to show" or "we must prove". For instance, the sentence "prove that x is a positive integer" is represented as "let x : `NoType` show $x : \mathbb{Z} \wedge \mathit{positive}(x)$", if x is not declared in the previous statements. Similarly, the sentence "show that there is an integer x such that $x > 0$" is represented as "show $\exists x(x : \mathbb{Z} \wedge x > 0)$.

4. trivial means that the proof of the active sequent is finished successfully. In mathematical texts it would be represented by key phrases such as "this ends the proof", "it is trivial", etc. However, such key phrases are not always given at the end of a finished proof. In such case, we add the rule trivial at the end of the proof automatically.

5. unfinished means that the proof of the active sequent is not finished but the rest of proof will be provided by the user later. We add this rule only when the key

$$
\dfrac{\dfrac{}{\Gamma_3 \vdash \text{multiple_of([x+y],2)}} \qquad 6.2.\ \dfrac{6.3.\ \dfrac{}{\Gamma_3,\ \text{multiple_of([x+y],2)} \vdash \mathit{even}(x+y)}\ \mathsf{trivial}}{\Gamma_3,\ \text{multiple_of([x+y],2)} \vdash \mathit{even}(x+y)}\ \mathsf{show(multiple_of([x+y],2))}}{**}\ \mathsf{deduce}\ (6.1.)
$$

$$
4.\ \dfrac{5.\ \dfrac{\dfrac{}{\Gamma_2 \vdash \text{x+y=2*(a+b)}}\ \mathsf{trivial} \qquad 6.1.\ \dfrac{**}{\Gamma_3 := (\Gamma_2,\ \text{x+y=2*(a+b)} \vdash \mathit{even}(x+y)}\ \mathsf{deduce}}{\Gamma_2 := (\Gamma_1, a, b : \mathbb{Z}, (x+y = 2*a+2*b)) \vdash \mathit{even}(x+y)}\ \mathsf{deduce(x+y = 2*a+2*b)}}{3.\ \ \Gamma_1 \equiv (\Gamma_0, x, y : \mathbb{Z}, \mathit{even}(x) \wedge \mathit{even}(y)) \vdash \mathit{even}(x+y)\ \ (\mathsf{let, assume, show})}\ \mathsf{let,\ assume(Even_Number)}
$$

Figure 4.2: Semantics of the proof in figure 4.1 as a proof tree. Justifications are given as arguments to rules as η in assume(η). Γ_0 is the initial context, which contains useful definitions (including definition 1 and 2) and axioms, needed to validate this proof. Line numbers are given at left hand sides of proof tree. Label (3.) represents the theorem formed by let, assume and show in proof tree. Double bar in the proof tree represents full-stop (•). Proof tree is given in two parts, where ** joins them.

phrases such as “we will prove it later”, “The proof will be completed later”, etc, are encountered in the textual proof.

6. full-stop ($\bullet$) marks the end of a sentence in proof. This information could be useful specially when we translate MathAbs in first-order formulas. Again, this information is kept to make the MathAbs’ proof closer to the original proof in NL.

7. $\{ \langle Proof \rangle; \langle Proof \rangle; \ldots \}$ is used for case analysis. It allows to split a proof in cases to use *proof-by-exhaustion* method. By using split, the active sequent is replaced by two or more sequents. $\langle Proof \rangle; \langle Proof \rangle; \ldots$ denotes a list with at least two items which themselves are $\langle Proof \rangle$. The scope of variables introduced by the rule let covers all cases. So we don’t have to declare the same variable in different branches. Consider the following example, showing a proof by case having 3 cases and its MathAbs; complete example proof is given in figure 4.4 on page 77.

Proof. If $n = 0$ then $m = 1$. we can choose $u = 0 \ldots$
Otherwise if $m = 0$ then $n = 1 \ldots$
Otherwise there exist r and q such that ...

Proof. {
let $u, y : \mathbb{Z}$ assume $n = 0$ show $m = 1$ trivial by form replace$(u, 0) \ldots$;
assume $n \neq 0$ assume $m = 0$ deduce $n = 1 \ldots$;
assume $n \neq 0$ assume $m \neq 0$ deduce $\exists r, q(\ldots)$
} ;

8. deduce allows to deduce something from the existing hypotheses. In concrete terms, deduce $\mathbb{A}$ ... is a syntactic sugar for:

{ show $\mathbb{A}$ trivial ; assume $\mathbb{A}$ ... },

where $\mathbb{A}$ is an arbitrary formula. It means that we distinguish two branches in the proof by replacing the active sequent by two sequents. In the first branch we prove a statement and then, in the second sequent, we use it as an hypothesis. For instance, “... we conclude that $x = 10$.” is translated as “... deduce $x = 10$” which is a syntactic sugar for

“... { show $x = 10$ trivial ; assume $x = 10$ ... }”

Operations in MathAbs:

We have three important operations in MathAbs proof as shown below:

Branching We divide a proof in sub proofs by using split, deduce and trivial, where trivial is zero-branch sub-proof.

Evolution We update the context by using let and assume and update the goal by using show.

Justifications ⟨*Hint*⟩ appears in almost all constructs of MathAbs ⟨*Proof*⟩. In informal mathematical proofs, it is common to give some justification(s) for each statement or proof step. In MathAbs, these justifications are also preserved and we call them ⟨*Hint*⟩. We define them as shown below:

$$
\begin{array}{lll}
\langle \mathit{Hint} \rangle & ::= & \textsf{by form } \langle \mathit{Formula} \rangle \ \langle \mathit{Hint} \rangle \\
& | & \textsf{by axiom } \langle \mathit{Ident} \rangle \ \langle \mathit{Hint} \rangle \\
& | & \textsf{by def } \langle \mathit{Ident} \rangle \ \langle \mathit{Hint} \rangle \\
& | & \textsf{by oper } \langle \mathit{Ident} \rangle \ \langle \mathit{Hint} \rangle \\
& | & \ldots \ \langle \mathit{Hint} \rangle \\
& | & \epsilon \qquad (\mathit{empty})
\end{array}
$$

⟨*Hint*⟩ is quite liberal in the sense that it is open to new constructs as shown by (... ⟨*Hint*⟩). Following are examples explaining them:

1. A formula e.g. $even(a)$ in a sentence such as "since, a is even, a^2 is even" (... deduce $even(a^2)$ trivial by $even(a)$)
2. A definition or axiom e.g. "we conclude that a is even by the definition of even number" (... deduce $even(a)$ trivial by def `Even_Number`)
3. An operation e.g. *squaring both sides* that should be translated into a prover specific tactic or command.

At the time of translation from MathAbs to the language of a certain proof assistant, it should be used as a supporting argument. However, a lot of work is required before we can use them in proof checking using an automated theorem prover (ATP) or a proof assistant. The reason for this enterprise is discussed in detail in §4.7 on page 79.

4.3.1 MathAbs and Extensibility

As it is evident from the MathAbs syntax and also from the short description in the explanation of axiom in §4.3, the language of *definition*, *theorem* and *proof* remain the same for all mathematical domains. In contrast, the language of ⟨*Formula*⟩, which represents mathematical statements, is arbitrary and subject to change from one mathematical domain to another.

In other words, the language of axiom, definition and proof is universal in mathematics; only the language for statement is domain dependent. That is why the various proof methods such as "proof by induction", "proof by contradiction", "proof by case", etc, are also common to all mathematical domains. Hence, the above discussion leads to the conclusion that the language of axiom, definition and proof in MathAbs is rather complete. However, the language of ⟨*Formula*⟩ should be extensible.

However, the language of ⟨*Formula*⟩ is fixed for the moment. It means that it is assumed to have all the formulas and predicate symbols we need (see definition 3 in §4.4 on page 65). Also note that, in §4.8, we suggest the introduction of meta variables as a possible extension to the MathAbs in future.

4.4 The MathAbs Semantics

The semantic of a MathAbs document D is a pair $(\mathbb{T}(D), \mathbb{P}(D))$. $\mathbb{T}(D)$ is the theory of the documents. It contains all of its axioms and definitions as set of formulas. It is used

as an initial context in all proofs. The language of formula ($\langle$*Formula*$\rangle$) is open and we assume it to be already defined. On the other hand, $\mathbb{P}(D)$ are the proved theorems together with their proof-trees. In the definitions below, A and B are MathAbs formulas.

Definition 1 (Empty Document)**.** *As a base case, if the document D is empty then:*

$$\mathbb{T}(D) := \{\} \qquad \mathbb{P}(D) := \{\}$$

Definition 2 (Axiom)**.** *Axioms are just added to the theory:*

$$\mathbb{T}(D; \mathsf{Axiom}\, A) := \mathbb{T}(D) \cup \{A\} \qquad \mathbb{P}(D; \mathsf{Axiom}\, A) := \mathbb{P}(D)$$

Definition 3 (Definition)**.** $\mathbb{D}(\theta)$ *takes a definition θ to build a MathAbs formula expressing the definition as an equivalence.*

We define its semantics as follows:

1. $\mathbb{D}(\mathsf{let}\ x{:}T\ \theta) := \forall x{:}T\ \mathbb{D}(\theta)$

   This definition requires that the language of formulas allows universal quantification.

2. $\mathbb{D}(\mathsf{assume}\ H\ \theta) := (H \Rightarrow \mathbb{D}(\theta))$

   This definition requires that the language of formulas contains implication.

3. $\mathbb{D}(\mathsf{define}\ A\ \mathsf{as}\ B) := (A = B)$,

   where A and B are arbitrary formulas or terms. Recall that A should follow some restrictions for it to be a definition as mentioned in §4.3 on page 60. However, having no restrictions at all should be fine also; if the definitions are understood as arbitrary equational axioms, possibly with some conditions which allow to handle partial definition.

   Remark: For the rule define we must take one of the following options:

   - We should support equality on formulas.
   - We should replace equality by equivalence and distinguish two kind of definitions.

   We prefer to take the first option and allow equality for the language of formulas. Now, we can give the semantics of definitions, which simply adds $\mathbb{D}(\theta)$ in the theory:

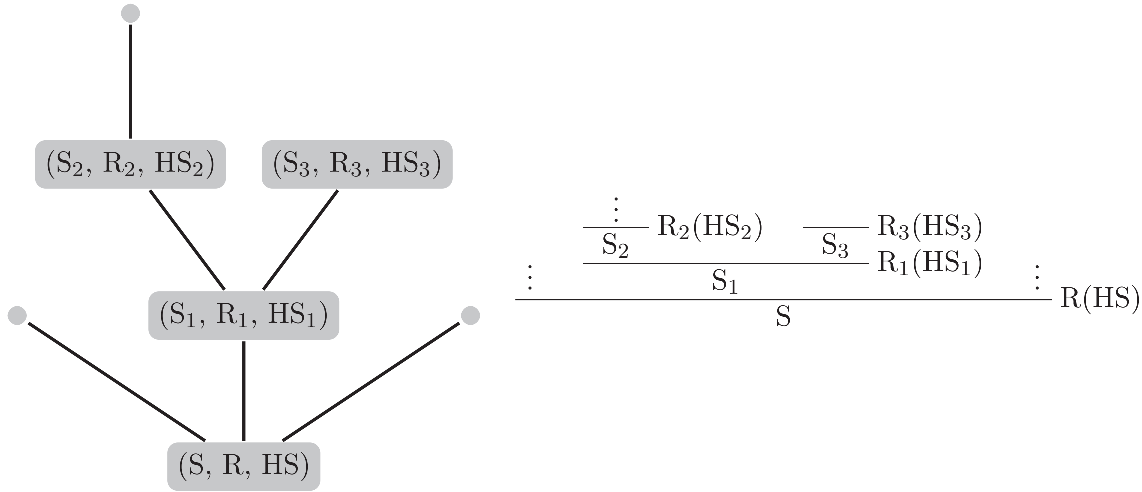
$$\begin{array}{rcl} \mathbb{T}(D; \mathsf{Definition}\,\theta) & := & \mathbb{T}(D) \cup \{\mathbb{D}(\theta)\} \\ \mathbb{P}(D; \mathsf{Definition}\,\theta) & := & \mathbb{P}(D) \end{array}$$

Finally, note that MathAbs does not allow language extension explicitly in any way (including by definition). The language of $\langle$*Formula*$\rangle$ is assumed to contain all the functions and predicate symbols needed, and definitions are treated as equational axioms. It is mainly because, a reliable method to extend CLM statements, functions and predicates for textual mathematics at run time with definitions is yet to be found. Therefore, we see no point adding this support for MathAbs now. We think that considering definition as language extension and adding some language constructions to add new constant symbols dynamically in MathAbs should be easy.

Definition 4 (Sequent)**.** *The sequent is of the form:* $\Gamma \vdash G$*, where* Γ *is the list of formulas (containing assumptions/hypotheses, deduced facts, axioms and definitions). We call it context. In contrast,* G *is a formula (the goal) which needs to be proved.*

Definition 5 (Proof)**.** *A proof is an n-branch labeled tree, whose vertices are labeled with 3–tuple (sequents, rule name, hint) on every node. The leaves of proof are zero-branch sub-proofs (trees). We use the term proof and proof tree synonymously to represent the MathAbs proof.*

To be more precise, we give below a drawing convention from the labeled proof tree found in logic to the style normally used for the natural deduction and sequent calculus proofs. Let S_i *be a sequent,* R_i *be MathAbs rule and* HS_i *be* ⟨*Hint*⟩*.*



Definition 6 (Semantics of Proof)**.** $\mathbb{S}(\pi;\ \Gamma \vdash G)$ *takes a proof* π *in MathAbs language and an initial sequent* $(\Gamma \vdash G)$*, and build a proof tree whose conclusion is the initial sequent.*

We use the letter Γ for *context* that contains a list of variables with types and formulas, letters H, G for formulas and η is for justification (⟨*Hint*⟩). We add rules and justifications as labels at the right side of the proof tree. We now give the semantics for MathAbs proof:

1. $\pi = \mathsf{let}\ x : \mathbb{T}\ \ \eta\ \ \pi'$,

   $$\mathbb{S}(\pi;\ \Gamma \vdash G) := \frac{\mathbb{S}(\pi';\ \Gamma, x : \mathbb{T} \vdash G)}{\Gamma \vdash G}\ \mathsf{let}(\eta)\ ,$$

   Here, $(\Gamma, x : \mathbb{T})$ means that variable x with its type T is added in the *context* Γ as an hypothesis and π' is the rest of the proof.

2. $\pi = \mathsf{assume}\ H\ \ \eta\ \ \pi'$,

   $$\mathbb{S}(\pi;\ \Gamma \vdash G) := \frac{\mathbb{S}(\pi';\ \Gamma, H \vdash G)}{\Gamma \vdash G}\ \mathsf{assume}(\eta)$$

   Here an arbitrary formula H is added in the *context* as an hypothesis.

3. $\pi = \mathsf{show}\ G_2\ \ \eta\ \ \pi'$,

   $$\mathbb{S}(\pi;\ \Gamma \vdash G_1) := \frac{\mathbb{S}(\pi';\ \Gamma \vdash G_2)}{\Gamma \vdash G_1}\ \mathsf{show}(\eta)$$

   Where G_1 is the previous goal which is changed to the new goal G_2.

4. $\pi = \mathsf{trivial}\ \eta$, $\mathbb{S}(\pi;\ \Gamma \vdash G) := \dfrac{}{\Gamma \vdash G}\ \mathsf{trivial}(\eta)$

5. $\pi = \mathsf{unfinished}$, $\mathbb{S}(\pi;\ \Gamma \vdash G) := \dfrac{}{\Gamma \vdash G}\ \mathsf{unfinished}$

6. $\pi = \bullet\ \ \pi'$, $\mathbb{S}(\pi;\ \Gamma \vdash G) := \mathbb{S}(\pi';\ \Gamma \vdash G)$

   i.e. full-stops has no semantics in MathAbs currently, and therefore, it is ignored. Again, this information could be useful when we translate MathAbs to the language of proof assistants or when we translate it to first-order formulas. In proof tree we represent it with double bar (cf. figures 4.1, 4.2 and 2.3).

7. $\pi = \{\pi_1, \ldots, \pi_n\}\ \eta$ (i.e. split)

   For all $i \in \{1, \ldots, n\}$, we find a proof tree $\mathbb{T}_i$ such that:

   $\mathbb{S}(\pi_i;\ \Gamma \vdash G) = \dfrac{\mathbb{T}_i}{\Gamma \vdash G}\ r_i$ then

   $\mathbb{S}(\pi_1, \ldots, \pi_n;\ \Gamma \vdash G) := \dfrac{\mathbb{T}_1 \ldots \mathbb{T}_n}{\Gamma \vdash G}\ \mathsf{split}(r_1, \ldots, r_n, \eta)$

   We keep the information regarding rules applied in proof at this step in $\mathsf{split}(r_1, \ldots, r_n, \eta)$.

   Remark: For all proof π and sequent $(\Gamma \vdash G)$ there is a unique proof tree $\mathbb{T}$ and rule r such that: $\mathbb{S}(\pi;\ \Gamma \vdash G) = \dfrac{\mathbb{T}}{\Gamma \vdash G}\ r$

Strange proof trees using split rule: The above definition permits the following proof trees, in which rules trivial and unfinished alone appear as a sub proof in the split rule. For instance:
$\{\mathsf{subproof}_1;\ \ldots;\ \mathsf{trivial};\ \ldots;\ \mathsf{subproof}_n\}$,
$\{\mathsf{subproof}_1;\ \ldots;\ \mathsf{unfinished};\ \ldots;\ \mathsf{subproof}_n\}$, etc.

First, such MathAbs proofs can never be generated by the CLM grammar. Second, such a proof has the same semantics as the same proof with trivial or unfinished removed.

Critic on split rule: Consider the following MathAbs proof (where π is the rest of the sub-proof): $\{\mathsf{show}\ A\ \mathsf{trivial}\ ;\ \mathsf{show}\ B\ \mathsf{trivial}\ ;\ \mathsf{assume}\ A\ \mathsf{assume}\ B\ \pi\}$. Unfortunately, it contains an invalid step as shown in the following proof tree:

$$\dfrac{\dfrac{}{\Gamma \vdash A} \qquad \dfrac{}{\Gamma \vdash B} \qquad \dfrac{\dfrac{\pi}{\Gamma, A, B \vdash C}}{\Gamma, A \vdash C}\ \text{Invalid}}{\Gamma \vdash C}$$

We do not yet know how to fix it. However, there are two points worth noting:

- It is quite hard to get this invalid rule from the textual proof.
- Instead of "$\mathsf{assume}\ A\ \mathsf{assume}\ B$" in the third branch, if we have "$\mathsf{deduce}\ A\ \mathsf{deduce}\ B$", we do not have this invalid step.
- Similarly, if we merge "$\mathsf{assume}\ A\ \mathsf{assume}\ B$" to "$\mathsf{assume}\ A \wedge B$", we can avoid this invalid step.

Definition 7 (Theorem)**.** *$\mathbb{D}'(\sigma)$ takes a theorem σ and build its statement. This definition is similar to $\mathbb{D}(\theta)$ of definition 3 in some parts, hence the similar choice for the name.*

1. $\mathbb{D}'(\mathsf{let}\ x{:}T\ \sigma) := \forall x{:}T\ \mathbb{D}'(\sigma)$
2. $\mathbb{D}'(\mathsf{assume}\ H\ \sigma) := (H \Rightarrow \mathbb{D}'(\sigma))$
3. $\mathbb{D}'(\mathsf{show}\ A) := A$

Definition 8 (Theorem with proof)**.** *Now, we give the interpretation of a theorem* (σ) *with its proof* (π) *in a document* (D)*:*

$$\begin{array}{lcl} \mathbb{T}(D; \mathit{Theorem}\ \sigma; \mathit{Proof}\ \pi) & := & \mathbb{T}(D) \\ \mathbb{P}(D; \mathit{Theorem}\ \sigma; \mathit{Proof}\ \pi) & := & \mathbb{P}(D) \cup \mathbb{S}(\sigma\pi, \mathbb{T}(D) \vdash \mathbb{D}'(\sigma)) \end{array}$$

In this definition, the theory is not changed as mentioned in the first line (but we could add the newly proved theorem in the theory). For theorem with proof (in second line), we first compute $\mathbb{D}'(\sigma)$, which is the statement of the theorem. Then, we reuse the definition of σ at the beginning of the proof ($\mathbb{S}(\sigma\pi, ...)$ in the second line above). It is because, the theorem may contains the introduction of variables or assumptions, which are intended to be available in its proof.

4.5 Completeness

By *completeness*, we mean that the language MathAbs is powerful enough to encode any proof written using natural deduction. Various complete notions of proofs have been proposed such as natural deduction, sequent calculus, Hilbert systems, etc. Mathematicians use their rules in textual proofs implicitly; even without knowing sometimes. Here, we give a translation of natural deduction in MathAbs to establish its completeness. For that, we first recall the rules of natural deduction for first order logic in figure 4.3. Note that this description also remain valid for higher order logic as well because of two reasons: First, the rules of natural deduction for first order logic are similar to the rules for higher order logic. Second, higher order logic can always be encoded to the first order logic fundamentally (even when the translation is too cumbersome) [Nour & Raffalli 2003, Prawitz 1965].

As far as we know, some of the rules we present here have rarely been considered before (rules in the decomposition of $\Rightarrow$–introduction and $\forall$–introduction). These rules are translated in more than one rule by the semantics of MathAbs, as first noted in the example given in §2.3 on page 10. Such decomposition of rules only occur because textual proofs may contain the intermediate steps. A sentence such as "we assume A and then we show B"[3] is a typical example causing such intermediate steps as shown in the $\Rightarrow$-introduction rule below.

Such intermediate steps of textual proofs are harmless, and in general, can lead to an interesting phenomenon. For instance, instead of the above example sentence, if we consider: "we show B and then assume A", it will lead to an invalid rule[4] of MathAbs as it inverts the commands assume and show (see bullet 1 below for the reason).

Due to the semantics of MathAbs (cf. §4.4), we can trace such invalid rules (of course not compulsory). Therefore, there are more chances to know in advance for a proof if it will succeed or fail, when given to the theorem prover (automated or interactive). Note

[3] Note that 'and' in this example is not a conjunction. Instead it is the sequential composition (like a semi-colon in some programming languages), best translated as implication.

[4] Recall that arbitrary rules can be represented in MathAbs including those which are invalid.

Axiom: $$\frac{}{\Gamma, A \vdash A}\ (\text{ax})$$

$$\frac{\Gamma, A \vdash C}{\Gamma, A, B \vdash C}\ (\text{weakening}) \qquad \frac{\Gamma, A \vdash B}{\Gamma, A, A \vdash B}\ (\text{contraction})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow I) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash A \Rightarrow B}{\Gamma \vdash B}\ (\Rightarrow E)$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge I) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge E_1) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge E_2)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee I_1) \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee I_2) \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee E)$$

We have two rules for $\bot$, both which eliminate it but introduce formula. RAA stands for *Reductio ad absurdum*.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot E) \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ (\text{RAA classic})$$

The formula "$\neg A$" is an abbreviation for "$A \Rightarrow \bot$".

$$\frac{\Gamma \vdash A(y) \quad y \text{ not free in } \Gamma}{\Gamma \vdash \forall x\, A(x)}\ (\forall I) \qquad \frac{\Gamma \vdash \forall x\, A(x)}{\Gamma \vdash A(t)}\ (\forall E)$$

$$\frac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x\, A(x)}\ (\exists I) \qquad \frac{\Gamma \vdash \exists x\, A(x) \quad \Gamma, A(y) \vdash C \quad y \text{ not free in } \Gamma \text{ and } C}{\Gamma \vdash C}\ (\exists E)$$

Γ is the context; A, B, C are arbitrary formulas; x, y are any variables; t is any term.

Figure 4.3: The rules of Gentzen's natural deduction (modified presentation of [David *et al.* 2001, van Dalen 1980]).

that, if a proof cannot be checked by the theorem prover (automated or interactive), it does not necessarily mean that the proof is invalid. It might be the case that theorem prover is unable to prove it due to some reason such as reasoning gaps (see §4.7.2 for more details). In such cases tracing invalid rules could be useful.

It also demonstrates that MathAbs tries to faithfully represent the textual proofs; as the later sentence does not seem to be accepted by mathematicians as well. So textual proof and MathAbs' seem to be in agreement on such examples.

This phenomenon is also interesting from a linguistic perspective. It means that when the incorrect MathAbs' interpretation for intermediate step of some sentence of the textual proof is produced, there are strong chances that such a sentence is also linguistically incorrect. So it seems that MathAbs' interpretation of textual proofs may somewhat give a little enlightenment to the linguistics of the language of mathematics.

1. $\Rightarrow$ **introduction**

   When the statement to be proved is of the form $A \Rightarrow B$, then after the step of assuming A the statement to be proved will be B. This means that $\Rightarrow$-introduction may be represented as "assume A show B" (where the goal in the context is: $A \Rightarrow B$).
   Here is the MathAbs semantics, with somewhat bizarre yet valid rules:

$$\cfrac{\cfrac{\cfrac{\vdots}{\Gamma, A \vdash B}}{\Gamma, A \vdash A \Rightarrow B}\,2.\ \text{show}}{\Gamma \vdash A \Rightarrow B}\,1.\ \text{assume}$$

   Using natural deduction, we now prove that each step of the above rules is valid. First, consider the step 1 (assume):

$$\cfrac{\cfrac{\cfrac{}{\Gamma, A \vdash A}\,\text{ax} \qquad \Gamma, A \vdash A \Rightarrow B}{\Gamma, A \vdash B}\,(\Rightarrow\text{E})}{\Gamma \vdash A \Rightarrow B}\,1.\ (\Rightarrow\text{I})$$

   Now the second step (show):

$$\cfrac{\cfrac{\Gamma, A \vdash B}{\Gamma, A, A \vdash B}\,(\text{contraction})}{\Gamma, A \vdash A \Rightarrow B}\,2.\ (\Rightarrow\text{I})$$

   So we have the formula A duplicated. But it does not make any problem as the old A and new A are the same (contraction is admissible).

   In contrast, if we translate sentence "we show B and then assume A". Its MathAbs "show B assume A" (where the goal in the context is: $A \Rightarrow B$) is correct as a whole. However, for each step, the first rule is correct but the second rule is incorrect:

$$\cfrac{\cfrac{\cfrac{\vdots}{\Gamma, A \vdash B}}{\Gamma \vdash B}\,\text{assume (incorrect in the natural deduction)}}{\Gamma \vdash A \Rightarrow B}\,\text{show – correct but too weak } (\Rightarrow\text{I and weakening})$$

2. $\Rightarrow$ **elimination (modus ponens):**

   This rule is translated as a split rule with two show constructs as "{ show $A \Rightarrow B$; show A }". The semantics is exactly the intended rule:

$$\dfrac{\dfrac{\vdots}{\Gamma, A \vdash B} \quad \dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash A \Rightarrow B}\ \text{split(show,show)}$$

3. $\forall$ **introduction:**

   When the statement that need to be proved is of the form $\forall_x P(x)$, then after the step 'let y' the statement to be proved will be $P(y)$. So we represent $\forall$-introduction as "let $y : \mathbb{T}$ show $P(y)$". Of course the name of the variable in the let usually will be the same as the name of the variable under the $\forall$ quantifier. Here is the semantics of this proof:

$$\begin{array}{l} \\ 2. \\ 1. \end{array} \dfrac{\dfrac{\dfrac{\vdots}{\Gamma, y : \mathbb{T} \vdash P(y)}}{\Gamma, y : \mathbb{T} \vdash \forall x : \mathbb{T}\ P(x)}\ \text{show}}{\Gamma \vdash \forall x : \mathbb{T}\ P(x)}\ \text{let}$$

   We see again that the usual rule of natural deduction is decomposed in two steps. It may look strange but these steps are logically valid when y is not free in the conclusion sequent of the proof.

   Using natural deduction, we now prove that each step of the above rules is valid. First the step 1 (let), as show below.

$$\dfrac{\dfrac{\Gamma, y : \mathbb{T} \vdash \forall x : \mathbb{T}\, P(x)}{\Gamma, y : \mathbb{T} \vdash P(y)}\ \forall E}{\Gamma \vdash \forall x : \mathbb{T}\, P(x)}\ \forall I$$

   Now the second step (show). Note that y, y' does not occur freely in Γ and $\forall x : \mathbb{T}\ P(x)$. Note that renaming and weakening are admissible:

$$\dfrac{\dfrac{\dfrac{\Gamma, y : \mathbb{T} \vdash P(y)}{\Gamma, y' : \mathbb{T} \vdash P(y')}\ \text{Renaming}}{\Gamma, y : \mathbb{T}, y' : \mathbb{T} \vdash P(y')}\ \text{Weakening}}{\Gamma, y : \mathbb{T} \vdash \forall x : \mathbb{T}\ P(x)}\ \forall\ \text{I}$$

4. $\forall$ **elimination:**

   It may be represented as "show $\forall x : \mathbb{T}\ P(x)$ by form replace(x, t)", where "by form replace(x, t)" is a justification $\langle Hint \rangle$, to hold the value of the variable x.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \forall x : \mathbb{T}\, P(x)}}{\Gamma \vdash P(t)}\ \text{show(replace(x,t))}$$

5. $\exists$ **introduction:**

   Its translation is similar to $\forall$-elimination: "show $P(t)$ by form replace(x, t)". We expect that t to be given immediately after $\exists x : \mathbb{T}\ P(x)$.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash P(t)}}{\Gamma \vdash \exists x : \mathbb{T}\ P(x)}\ \mathsf{show}(\mathsf{replace}(\mathrm{x,t}))$$

   Remark: Natural deduction always gives the term t. In contrast, the textual proofs often do not give t immediately after $\exists x : \mathbb{T}\ P(x)$; especially when it results from a long computation. See bullet 3 of §4.6 on page 75 for more details and alternates.

6. $\exists$ **left rule of sequent calculus:**

   Instead of $\exists$-elimination rule of natural deduction, mathematicians implicitly tend to use the left rule of sequent calculus in textual proofs. In fact, both rules are logically equivalent (see theorem 1 on page 73 for its proof). Also the left rule of sequent calculus is more natural for textual proofs.

   $\exists$ introduction at left (as an hypothesis) is represented as "let $y : \mathbb{T}$ assume $P(y)$"

$$\dfrac{\dfrac{\Gamma, \exists x : \mathbb{T}\ P(x), y : \mathbb{T}, P(y) \vdash \Delta}{\Gamma, \exists x : \mathbb{T}\ P(x), y : T \vdash \Delta}\ \text{assume}}{\Gamma, \exists x : \mathbb{T}\ P(x) \vdash \Delta}\ \text{let}$$

   Again, the rule is decomposed, but correct if y is not free in the conclusion sequent. In fact this rule is similar to the rule $\forall$ introduction (given in bullet 3 on page 71). Note that Δ contains only one goal because of the restriction imposed by MathAbs. Also note that at for all the rules of sequent calculus, on the left hand side of the turnstile $\vdash$, we have assumptions and on its right hand side we have goal.

7. The reader can easily complete this with the rules for other connectives and even consider all Gentzen's sequent calculus rules. The four rules among these help to translate textual quantifiers to MathAbs, given in §4.6.

4.6 Translating Textual Quantifiers to MathAbs

Translating quantifiers which occurs in textual theorem and proof is not an easy task. An introduction to this problem and examples from textual mathematics are already given in §3.3.3.6. In this section, we describe how we translate them to MathAbs. We already translated the $\forall, \exists$ introduction and elimination rules of natural deduction to MathAbs in §4.5. Here we are going to add more details and also mention various subtleties involved.

First and foremost, recall from bullets 1 on page 62 that we make the direct declaration of new variables with the rule "let". It is in fact the universal quantification of the variable for the rest of the block (definition, theorem, proof, etc). For instance,

---

Theorem 1. *Prove that $\exists$ left rule of Gentzen's sequent calculus is logically equivalent to $\exists E$ rule of Gentzen's natural deductions. Abbreviated as $\exists$ left $\Leftrightarrow \exists E$. (all other rules being unchanged)*

Proof.
(1) We prove that $\exists$ left $\Rightarrow \exists E$ (i.e. complete):

$$\dfrac{\Gamma \vdash \exists x P(x) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, P(y) \vdash P(y)}\text{ax} \qquad \dfrac{\dfrac{\Gamma, P(y) \vdash G}{\Gamma \vdash \forall x\,(P(x) \Rightarrow G)}{\Rightarrow}\text{I}, \forall\text{I}}{\Gamma \vdash P(y) \Rightarrow G}\forall\text{E}}{\Gamma, P(y) \vdash G}{\Rightarrow}\text{E}}{\Gamma, \exists x P(x) \vdash G}(\exists\text{ left})}{\Gamma \vdash \exists x P(x) \Rightarrow G}({\Rightarrow}\text{I})}{\Gamma \vdash G}({\Rightarrow}\text{E})$$

(2) We prove that $\exists E \Rightarrow \exists$ left (i.e. correct):

$$\dfrac{\dfrac{}{\Gamma, \exists x P(x) \vdash \exists x P(x)}\text{ax} \qquad \dfrac{\Gamma, P(y) \vdash G}{\Gamma, \exists x P(x), P(y) \vdash G}\text{weakening}}{\Gamma, \exists x P(x) \vdash G}(\exists\text{ E})$$

"Let x be a positive integer." or "Suppose that x is a positive integer."
Both are translated as: let $x : \mathbb{Z}$ assume positive(x) •

In general, we consider all the variables as universally quantified unless they are explicitly specified as existentially quantified (also noted by [Peters & Westerstahl 2006, p. 34]). In mathematical texts, variables are normally existentially quantified with key phrases such as "there is/are", "there exist(s)", etc. However, these key phrases does not always indicate an existentially quantified variable. Details are given in bullet 2 below.

In concrete terms, we translate quantifiers with the help of the following rules of the Gentzen's sequent calculus and natural deduction:

1. **Statement containing universal variable as goal** is in fact an easy case. It is translated as "let", as shown below using the $\forall$ right rule of sequent calculus:

$$\dfrac{\Gamma, y : \mathbb{T} \vdash P(y), \Delta}{\Gamma \vdash \forall x : \mathbb{T}\, P(x), \Delta}\ (\forall R)$$

   Of course Δ is empty in the context of MathAbs (cf. bullet 6 of §4.5). So the rule becomes the same as $\forall$ introduction rule of natural deduction:

$$\dfrac{\Gamma, y : \mathbb{T} \vdash P(y)}{\Gamma \vdash \forall x : \mathbb{T}\, P(x)}$$

   Therefore, by bullet 3 in §4.5, we translate it with the combinations of rules: let, assume and show. Here are a few examples:

Theorem. Prove that x is positive.
Logical formula as a goal: $\forall x : \texttt{NoType}\ (\text{positive}(x))$
MathAbs: $\mathsf{Theorem.}\ \mathsf{let}\ x : \texttt{NoType}\ \mathsf{show}\ \text{positive}(x) \bullet$

Theorem. If $x > 0$ then it is positive.
Logical Formula as a goal: $\forall x : \texttt{NoType}\ ((x > 0) \Rightarrow \text{positive}(x))$
MathAbs: $\mathsf{Theorem.}\ \mathsf{let}\ x : \texttt{NoType}\ \mathsf{assume}\ x > 0\ \mathsf{show}\ \text{positive}(x)\ \bullet$

Note that the equivalent logical formulas are only given for such occasions when we need it directly instead of its MathAbs (for instance, anaphoric resolution). Such an occasion could also be the following statements appearing later in the proof:

Proof. Assume that the statement of the theorem is not true.
([Burton 2007, Theorem 1.1]),
Suppose that it is not the case that the statement of the theorem is true,
etc.

As an example, consider the following theorem and its proof, which is translated in MathAbs afterwards:

Theorem. Prove that x is positive.
Proof. Suppose that it is not the case that the statement of the theorem is true. ...
MathAbs:
$\mathsf{Theorem.}\ \mathsf{let}\ x : \texttt{NoType}\ \mathsf{show}\ \text{positive}(x)\ \bullet$
$\mathsf{Proof.}\ \mathsf{assume}\ \neg\forall x : \texttt{NoType}\ (\text{positive}(x)) \bullet \ldots$

Another point worth nothing is this following. If we have a negative statement of the form:

Theorem. Prove that there does not exist an integer x such $x + 1 = x$.

It is translated as "$\neg\exists\, x\ :\ \mathbb{T}\, P(x)$" which is equivalent to "$\forall\, x\ :\ \mathbb{T}\, \neg P(x)$". In MathAbs, we keep it as: "$\mathsf{show}\ \forall\, x : \mathbb{T}\, \neg P(x)$" and do not translate it as "$\mathsf{let}\ x : \mathbb{T}$ $\mathsf{show}\ P(x)$" (The difference between $\forall x : \mathbb{T}$ and $\mathsf{let}\ x : \mathbb{T}$ is that: the former is not available in the proof but the later is). It is because of the following principle that:

"A negative statement cannot introduce a variable."

2. **Statement containing existential variable as an assumption** is translated as a combination of "let and assume" (cf. $\exists$ left rule of sequent calculus explained in bullet 6 of §4.5). An example with its MathAbs translation is shown below:

   Suppose that there (is | exists) an integer x such that $x > 0$.
   Logical formula as an assumption: $\exists x : \mathbb{Z}\ (x > 0)$.
   MathAbs: $\mathsf{let}\ x : \mathbb{Z}\ \mathsf{assume}\ x > 0 \bullet$

3. **Statement containing existential variable as goal** is “show $\exists x : \mathbb{T}\, P(x)$”, according to the $\exists$ right rule of sequent calculus:

$$\frac{\Gamma \vdash P(t), \Delta}{\Gamma \vdash \exists x : \mathbb{T}\, P(x), \Delta}\ (\exists R)$$

Again Δ is empty, so the rule becomes the same as $\exists$ introduction rule of natural deduction given in bullet 5 of §4.5:

$$\frac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x : \mathbb{T}\, P(x)}\ \mathsf{show(replace(x,t))}$$

This rule suggests that $\exists x : \mathbb{T}\, P(x)$ as a goal could be translated it as: “let $x : \mathbb{T}$ ... show $P(t)$ by form replace(x, t)”. In fact it could be translated to any of these optional proof steps by the mathematician, where replace(...) is given as justification ($\langle Hint \rangle$):

“let $x : \mathbb{T}$ ... show $P(t)$ by form replace(x, t)”,
“let $x : \mathbb{T}$ ... show $P(t)$ trivial by form replace(x, t)”,
“let $x : \mathbb{T}$ ... trivial by form replace(x, t)”, (the rule show not given)

Currently, we cannot utilize replace(x, t) in the proof checking. Because, we do not translate justifications when translating MathAbs to first order formulas. The reason is that: the automated theorem provers cannot handle them and it requires further investigation how we can use these justifications to aid ATP for reducing its search space of hypotheses and deductions. See §4.7 on page 79 for more details.

Also, the textual proofs do not give t immediately after $\exists x : \mathbb{T}\, P(x)$ often; especially when it results from a long computation. We cannot handle such proofs (although they are quite rare). We cannot handle them because it involves the introduction of meta-variables (i.e. existential variables) in MathAbs to take into account the use of unknown variables. The main problem is that introducing meta-variables prohibits to check each rule in the proof tree separately; as the check becomes global.

Finally, it is also very important to note that **none of the above mentioned forms are compulsory**. The mathematician usually skip them when the existential statement given as a goal is obvious. This could be seen as a fix to the problem discussed in the previous paragraph. Therefore, most of the time, the goal $\exists x : \mathbb{T}\ P(x)$ appears unchanged with the rule show (i.e. show $\exists x : \mathbb{T}\ P(x)$). It means that mathematicians usually do not decomposed it in the later proof steps. See figure 4.4 on page 77 for $\exists x : \mathbb{T}\, P(x)$ in action as a goal.

4. **Statement containing universal variable as assumption** is fundamentally similar to the statement containing existential variable as **goal** given in bullet 3 (reasons of this enterprise is given in subsequents paragraphs). Therefore, statement containing universal variable as **assumption** is translated as “assume $\forall x : \mathbb{T}\, P(x)$” most of the time. Here are a few examples:

Suppose that for all integers x and y, $x * y > 0$. or
Assume that $x * y > 0$, for all integers x and y.
Translated in MathAbs as: assume $\forall x, y : \mathbb{Z}\, (x * y > 0)$ •

> Let n be an arbitrary positive integer. Assume that $P(k)$ is true for all positive integers $k < n$. ([Erickson & Heeren 2007, p. 18])
> let $n : \mathbb{Z}$ assume positive(n) •
> assume $\forall k : \mathbb{Z}\,(P(k) \wedge\, k < n \wedge \text{positive}(k))$ •

The $\exists$ right rule and the $\forall$ left rule of sequent calculus are fundamentally interchangeable. For instance, consider $\forall$ left rule:

$$\frac{\Gamma, P(t) \vdash \Delta}{\Gamma, \forall x : \mathbb{T}\, P(x) \vdash \Delta} \quad (\forall L)$$

We can transform it to $\exists$ right rule if we take $\forall x : \mathbb{T}\ P(x)$ on the right side of the turnstile. By doing so, it becomes $\exists x : \mathbb{T}\ \neg P(x)$, as shown below. It is a kind of absurdity reasoning which can be done in textual proofs.

$$\frac{\Gamma \vdash \neg P(t)}{\Gamma \vdash \exists x : \mathbb{T}\ \neg P(x)} \quad \mathsf{show}(\mathsf{replace}(x, t))$$

Because, now it is the same as the rule in bullet 3, $\forall x : \mathbb{T}\, P(x)$ as assumption could be translated by any of these optional steps (in the later proof steps by the mathematician; of course none of these appear when mathematicians do not decomposed it in later proof steps):

> "let $x : \mathbb{T}$ ... show $\neg P(t)$ by form replace(x, t)",
> "let $x : \mathbb{T}$ ... show $\neg P(t)$ trivial by form replace(x, t)",
> "let $x : \mathbb{T}$ ... trivial by form replace(x, t)", (the rule show not given).

4.6.1 Simplifying MathAbs

As mentioned in bullet 8 on page 63, the rule deduce is simply a syntactic sugar. For instance, "deduce A π" is a syntactic sugar for: { show A trivial ; assume A π }, where A is an arbitrary formula and π is the rest of proof. The fact that the rule deduce is treated as a goal in one branch and hypothesis in the other branch, may cause confusion for the translation of quantifiers. Therefore, before proof checking (given in §4.7), we simplify MathAbs by replacing the rule deduce with its equivalent rules. For this, we apply the following procedure on MathAbs. Note that the keyword assumeC used in the procedure below is just an auxiliary procedure (and it is not a MathAbs extension), which is eventually replaced by the rule assume.

The procedure:

1. deduce A $\pi \equiv$ { show A trivial ; assumeC A π}, where:
   (a) assumeC $\exists x : \mathbb{T}\,(A)$ $\pi \equiv$ let $x : \mathbb{T}$ assumeC A π
   (b) assumeC $(A \wedge B)$ $\pi \equiv$ assumeC A assumeC B π (according to $\wedge$*left* rule of sequent calculus)
   (c) assumeC A $\pi \equiv$ assume A π (otherwise)
2. Some sanity checks to report warnings to give insight

1. **Theorem.** Assume that m and n are relatively prime integers.
2. Suppose that either $m \neq 0$ or $n \neq 0$.
3. Then prove that there exist two integers u and v such that $u * n + v * m = 1$ holds.
4. **Proof. If** $n = 0$ then $m = 1$ because m and n are coprime.
5. We can choose $u := 0$ and $v := 1$.
6. **Otherwise if** $m = 0$ and $n = 1$ then we can choose $u := 1$ and $v := 0$.
7. **Otherwise** there exist r and q such that $n = m * q + r$ holds by euclidean division.
8. It is obvious that m and r are coprime and $r < m$.
9. So by induction hypothesis there are u' and v' such that $u' * m + v' * r = 1$ holds.
10. It implies that $u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$.
11. So we can choose $u := v'$ and $v := u' - v' * q$.

MathAbs

1. **Theorem.** let $m, n : \mathbb{Z}$ assume $coprime(m, n)$ $\bullet$
2. assume $m \neq 0$ assume $n \neq 0$ $\bullet$
3. show $\exists(u, v : \text{Integer})(u * n + v * m = 1)$ $\bullet$
4. **Proof.** $\{$
   assume $n = 0$ deduce $coprime(m, n)$ $\bullet$
5. deduce $m = 1$ by form $coprime(m, n)$ trivial by form $\text{replace}(u, 0)$ by form $\text{replace}(v, 1)$ $\bullet$;
6. assume $n \neq 0$ assume $m = 0$ assume $n = 1$ trivial by form $\text{replace}(u, 1)$ by form $\text{replace}(v, 0)$ $\bullet$;
7. assume $n \neq 0$ assume $m \neq 0$ assume $n \neq 1$
   deduce $\exists(r, q : \texttt{NoType})(n = m * q + r)$ by def $\text{Euclidean_Division}$ $\bullet$
8. let $r : \texttt{NoType}$ deduce $coprime(m, r) \wedge r < m$ $\bullet$
9. deduce $\exists(u', v' : \texttt{NoType})(u' * m + v' * r = 1)$ by def $\text{Induction_Hypothesis}$ $\bullet$
10. let $q, u', v' : \texttt{NoType}$ deduce $u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$ $\bullet$
11. trivial by form $u := v'$ by form $v := u' - v' * q$ $\bullet$
   $\}$;

Figure 4.4: A proof by case and its MathAbs explaining existential quantifier.

Sanity checks: To explain the sanity checks, consider the following cases which are problematic somehow, and we should report warnings:

Problem Case 1: deduce $((\exists x : \mathbb{T}\, A) \wedge (\exists x : \mathbb{T}\, B))\ \ \pi$

It is equivalent to:

{ show $((\exists x : \mathbb{T}\, A) \wedge (\exists x : \mathbb{T}\, B))$ trivial ;
let $x : \mathbb{T}$ assume A let $x : \mathbb{T}$ assume $B\ \ \pi$
}

Which x should be available in the rest of proof π? Furthermore, it is hardly possible to find a textual proof which uses the same variable in two existentially quantified statements, which are then further connected by conjunction. Therefore, we send a warning message, but at the same time, we make available the second x in π.

Problem Case 2: let $x : \mathbb{T} \ldots$ deduce $((\exists x : \mathbb{T}\, A) \wedge B(x))\ \ \pi$

It is somewhat similar to the first case. Again, which x should be available in $B(x)$ and π? Similar to case 1, we send a warning message regarding the scope of x, but at the same time, we make available the second x in $B(x)$ and π.

Now we can translate textual quantifiers which occur as conclusion:

4.6.1.1 The Quantifier $\exists x : \mathbb{T}\, P(x)$ as deduction:

The rule deduce $\exists x : \mathbb{T}\, P(x)\ \ \pi$ (π is the rest of proof) is equivalent to:

{ show $\exists x : \mathbb{T}\, P(x)$ trivial ;
let $x : \mathbb{T}$ assume $P(x)\ \ \pi$ (x is bound in this branch)
}

$$\dfrac{\dfrac{}{\Gamma \vdash \exists x : \mathbb{T}\, P(x)}\text{trivial} \qquad \dfrac{\pi}{\Gamma,\, x : \mathbb{T},\, P(x) \vdash G}}{\Gamma \vdash G}\text{deduce}$$

We present here two examples from figure 4.4 (line 7 and 9):

Example 1:

There exist r and q such that $n = m * q + r$ holds by euclidean division.
deduce $\exists (r, q : \texttt{NoType})(n = m * q + r)$ by def Euclidean_Division $\bullet$

Which is a short hand of:

{
show $\exists (r, q : \texttt{NoType})(n = m * q + r)$ trivial by def Euclidean_Division ;
let r, q : NoType assume $(n = m * q + r) \bullet \ldots$
}

Example 2:

So by induction hypothesis there are u' and v' such that $u' * m + v' * r = 1$.
deduce $\exists (u', v' : \texttt{NoType})(u' * m + v' * r = 1)$ by def Induction_Hypothesis $\bullet$

It is a short hand of:

{
show $\exists(u', v' : \texttt{NoType})(u' * m + v' * r = 1)$ trivial by def Induction_Hypothesis ;
let u', v' : NoType assume $(u' * m + v' * r = 1) \bullet \ldots$
}

4.6.1.2 The Quantifier $\forall x : \mathbb{T}\, P(x)$ as deduction:

The rule deduce $\forall x : \mathbb{T}\, P(x)\ \ \pi$ is equivalent to:

{
show $\forall x : \mathbb{T}\, P(x)$ trivial ;
assume $\forall x : \mathbb{T}\, P(x)\ \ \pi$
}

In the first branch there is no point of writing let $x : \mathbb{T}$ show $P(x)$ instead. Because x is not used later in the same branch.

4.7 Proof checking

4.7.1 Proof checking using Automated Theorem Provers

We can translate MathAbs in first order or higher order formulas (with equality) for the purpose of verification by the automated theorem provers (ATP). As a first prototype, we have implemented a translation from MathAbs to its equivalent first-order formulas. As we see in §2.3, we can represent MathAbs proof as a proof tree using arbitrary rules (not just the rules of natural deduction or sequent calculus). Then, for each rule we can produce a formula that justifies it. We can informally describe this translation with the following pattern of MathAbs' and proof tree, (complete and formal procedure is given §4.7.1.4):

Theorem. show $G_0\ \bullet$;
Proof. assume H_1 show $G_1\ \bullet$
deduce G_2 ... trivial $\bullet$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0, H_1 \vdash G_2}\ \textit{show} \qquad \dfrac{\dfrac{\overline{\overline{\quad}}}{\vdots}}{\Gamma_0, H_1, G_2 \vdash G_1}\ \textit{assume}}{\Gamma_0, H_1 \vdash G_1}\ \text{deduce}\ (3.)}{\Gamma_0, H_1 \vdash G_0}\ \text{show}\ (2.)}{(\Gamma_0 \vdash G_0)^*}\ \text{assume}\ (1.)
$$

The theorem is labeled as * in the proof tree, where Γ_0 is the initial sequent. As already stated in figure 4.2 on page 62, it may contain necessary axioms, definitions, propositions, theorems, etc, that are needed to support this proof. In contrast, G_0 is the initial goal given in the theorem by the show rule).

The proof starts from $(\Gamma_0, H_1 \vdash G_0)$ labeled as (1). Goals G_0 and G_1 as shown in label (2) could either be same (as in the example given in figure 4.1) or different[5]. After

[5]Mathematicians normally do it with a statement in proof of kind: "we shall prove that G_1 holds which is logically equivalent to G_0", etc.

that, we have a command "deduce" (labeled as 3). It has two parts: the one shown at left side that is complete, and the second shown at right side. It is also complete but some proof steps are omitted here and shown with vertical dots ($\vdots$).

The turnstile ($\vdash$) and the line (single or double) are translated as the implication ($\Rightarrow$). In contrast, the comma (,) is translated as the conjunction ($\wedge$). We produce one formula for each rule. The rule deduce is an exception. It is a split with two branches, containing two rules show and assume. Therefore, we produce two formulas for it.

4.7.1.1 The first rule in the proof

Let us consider the first rule in the proof (i.e. assume) in the proof tree as shown below:

$$1. \frac{\Gamma_0, H_1 \vdash G_0}{\Gamma_0 \vdash G_0} \text{assume}$$

We can directly translate it in the following first order formula. The most natural translation would be the direct translation of the rule:

$$((\Gamma_0 \wedge H_1) \Rightarrow G_0) \Rightarrow (\Gamma_0 \Rightarrow G_0)$$

We can make it compact with factorization. For that we can exchange the conjunction between Γ_0 and H_1 with implication, because of the following equivalence relation:

$$(A \wedge B) \Rightarrow C \qquad \equiv \qquad A \Rightarrow (B \Rightarrow C)$$

So the above first order formula becomes the following by factorizing Γ_0:

$$\Gamma_0 \Rightarrow ((H_1 \Rightarrow G_0) \Rightarrow G_0)$$

4.7.1.2 The second rule in the proof

$$2. \frac{\Gamma_0, H_1 \vdash G_1}{\Gamma_0, H_1 \vdash G_0} \text{show}$$

The direct translation of this rule is:

$$((\Gamma_0 \wedge H_1) \Rightarrow G_1) \Rightarrow ((\Gamma_0 \wedge H_1) \Rightarrow G_0)$$

And the above first order formula becomes the following by factorizing $(\Gamma_0 \wedge H_1)$:

$$(\Gamma_0 \wedge H_1) \Rightarrow (G_1 \Rightarrow G_0)$$

4.7.1.3 The third rule in the proof

$$3. \frac{\overline{\Gamma_0, H_1 \vdash G_2 \quad \textit{show}} \qquad \Gamma_0, H_1, G_2 \vdash G_1 \quad \textit{assume}}{\Gamma_0, H_1 \vdash G_1} \text{deduce}$$

This translation will explain how we translate an $n-$branch proof by case (the split rule). Because deduce has exactly two branches, and therefore, it results in the following two first order formulas:

- **The first formula** formed by the first case (the one at left hand side with show) is following. Let us name it $formula_1$:

$$(\Gamma_0 \wedge H_1) \Rightarrow G_2$$

- The sub-formula formed by the second case is following. Let us name it $formula_2$: $(\Gamma_0 \wedge H_1 \wedge G_2) \Rightarrow G_1$

  Then the **second formula** which is formed by both first and second case would be (in short):

  $$(formula_1 \wedge formula_2) \Rightarrow ((\Gamma_0 \wedge H_1) \Rightarrow G_1)$$

  And in detail:

  $$\underbrace{(\underbrace{(\underbrace{(\Gamma_0 \wedge H_1) \Rightarrow G_2)}_{} \wedge \underbrace{((\Gamma_0 \wedge H_1 \wedge G_2) \Rightarrow G_1)}_{})}_{} \Rightarrow ((\Gamma_0 \wedge H_1) \Rightarrow G_1)}_{}$$

  By boolean algebra, we can have the following formula:

  $$(\Gamma_0 \wedge H_1) \Rightarrow ((G_2 \wedge (G_2 \Rightarrow G_1)) \Rightarrow G_1)$$

Because the command deduce G_2 is a syntactic sugar of:

{ show G_2 trivial ; assume G_2 ... }

It produces a lot of tautologies of the form $(G_2 \wedge (G_2 \Rightarrow G_1)) \Rightarrow G_1$ in the first-order formulas. To reduce the work load of the Automated Theorem Prover (ATP) for proof checking we have the following two options:

1. (a) Because we know that these tautologies are produced by the rule deduce, it is possible to identify them. Therefore, we could remove them manually, before sending the result to ATP for validation.
   (b) However, we may also apply a SAT-solver on all the generated formulas as preprocessor to filter such tautologies, and then, send the result to ATP for validation.
2. As a second option, we can send these MathAbs equivalent formulas to ATP as it is. of course, if all proof-steps are validated then the proof is valid and therefore, establishes the truth of the theorem.

4.7.1.4 Generalizing Translation

Consider the following split rule having n branches in the form of a proof tree. All of these branches terminate with trivial after some proof steps.

$$\cfrac{\cfrac{\nwarrow \vdots \nearrow}{\Gamma_0, \Gamma_1 \vdash G_1} \quad \cfrac{\nwarrow \vdots \nearrow}{\Gamma_0, \Gamma_2 \vdash G_2} \quad \cdots\cdots \quad \cfrac{\nwarrow \vdots \nearrow}{\Gamma_0, \Gamma_n \vdash G_n}}{\Gamma_0 \vdash G_0}$$

As usual, the comma is translated as logical conjunction and turnstile is translated as implication:

$$[((\Gamma_0 \wedge \Gamma_1) \Rightarrow G_1) \wedge ((\Gamma_0 \wedge \Gamma_2) \Rightarrow G_2) \wedge ... \wedge ((\Gamma_0 \wedge \Gamma_n) \Rightarrow G_n)] \Rightarrow (\Gamma_0 \Rightarrow G_0)$$

After factorizing:

$$\Gamma_0 \Rightarrow [((\Gamma_1 \Rightarrow G_1) \wedge (\Gamma_2 \Rightarrow G_2) \wedge ... \wedge (\Gamma_n \Rightarrow G_n)) \Rightarrow G_0]$$

We define $\Gamma \Rightarrow A$ by induction on Γ.

$$\Gamma \Rightarrow A \equiv \begin{cases} \text{Empty context} \Rightarrow A & = A \\ (B, \Gamma') \Rightarrow A & = B \Rightarrow (\Gamma' \Rightarrow A) \\ (x : \mathbb{T}, \Gamma') \Rightarrow A & = \forall x : \mathbb{T}(\Gamma' \Rightarrow A) \end{cases}$$

Types as Predicates: We treat types as predicates (For instance, "$\sqrt{2}$: Rational" as "rational($\sqrt{2}$)"). Also note that the variables introduced by let (e.g. let $x, y : \mathbb{Z}$) are available in any subsequent sentences (and hence in any subsequent formulas) in the form "$\forall\, x, y\, (\text{int}(x) \wedge \text{int}(y) \Rightarrow \ldots)$ ".

The notion of types used in the grammar is linguistic which is not exactly similar to the notion of type in a given type theory. This is one of the main problems if we want to translate CLM to a typed framework such as Coq[Team 2010] or Agda[Norell 2007a, Norell 2007b].

Justifications: Finally, the justifications $\langle Hint \rangle$, that are preserved in MathAbs are removed from the first order translation. Because such justifications are too difficult to handle with the current state of art ATP. See §4.7.2 for further details.

The translation from MathAbs to first order formulas for an example theorem and its proof is given in §4.9.

4.7.1.5 An Alternate Approach

As an alternate approach, we could interpret rules collectively on full-stops in future (i.e. at the end of each sentence). In other words, instead of producing a formula for each rule, we could produce formula(s) for each full-stop.

So in MathAbs proof tree, each double bar (i.e. full-stop $\bullet$) could be translated as an implication. However, for the single bar we can do following:

1. let, assume and all their combinations are translated as conjunction
2. show/deduce after let/assume are translated as implication

The reason for the second bullet (above) is the following rephrasing of conditional statements in the textual proofs:

If A then B.
Let A. Then B.
...

We give below the MathAbs pattern followed by the collapsed proof-tree on end of sentences:

Theorem. show G_0 $\bullet$;
Proof. assume H_1 assume H_2 show G_1 $\bullet$
deduce G_2 $\bullet \ldots$ trivial $\bullet$

$$
\dfrac{2.\ \dfrac{}{\Gamma_0, H_1, H_2 \vdash G_2}\ \text{trivial} \qquad \dfrac{\overline{\overline{\quad\vdots\quad}}}{\Gamma_0, H_1, H_2, G_2 \vdash G_1}}{1.\ \dfrac{\Gamma_0, H_1, H_2 \vdash G_1}{(\Gamma_0 \vdash G_0)^*}\ \text{assume, assume, show}}\ \text{deduce}
$$

For the proof tree labeled as (1), we'll have the following formula:

$$((\Gamma_0 \wedge H_1 \wedge H_2) \Rightarrow G_1) \Rightarrow (\Gamma_0 \Rightarrow G_0)$$

And the above first order formula becomes the following by factorizing Γ_0:

$$\Gamma_0 \Rightarrow ((H_1 \wedge H_2 \Rightarrow G_1) \Rightarrow G_0)$$

For the proof tree labeled as (2) we'll have the following two formulas:

1. $(\Gamma_0 \wedge H_1 \wedge H_2) \Rightarrow G_2$
2. $\underbrace{(\underbrace{\underbrace{((\Gamma_0 \wedge H_1 \wedge H_2) \Rightarrow G_2)}\wedge\underbrace{((\Gamma_0 \wedge H_1 \wedge H_2 \wedge G_2) \Rightarrow G_1)}}) \Rightarrow ((\Gamma_0 \wedge H_1 \wedge H_2) \Rightarrow G_1)}$

   By factorization, we'll have:

   $$(\Gamma_0 \wedge H_1 \wedge H_2) \Rightarrow ((G_2 \wedge (G_2 \Rightarrow G_1)) \Rightarrow G_1)$$

Recall the discussion regarding bizarre yet logically correct rules for the following sentence in §4.5 on page 68.

> We assume A and then we show B.

We also discussed that if we inverse both rules of MathAbs (assume, show) used in the above sentence, we'll have the following sentence which in mathematically as well as logically incorrect (we know it from its MathAbs).

> *We show B and then we assume A.

With this alternate approach of interpreting rules collectively at the end of each sentence, we cannot reject this mathematically incorrect sentence. Because in the corresponding first order formula we have no way of knowing if it was "assume A show B $\bullet$" or "show B assume A $\bullet$", demonstrated by the following proof trees:

assume A show B $\bullet$ show B assume A $\bullet$

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash G}\ \text{assume, show} \qquad \dfrac{\Gamma, A \vdash B}{\Gamma \vdash G}\ \text{show, assume}$$

4.7.1.6 A Better Alternate Approach (May be)

Because of the shortcoming of the above alternate method, we can modify our approach to the following:

We produce formula(s) for:

1. Each rule (except let, assume)

2. Each full-stop ($\bullet$)

So in MathAbs proof tree, all the combinations of assumption rules (let, assume) are joined together with conjunctions. The rest of the procedure will remain the same. Because of it, we can distinguish between "assume A show B $\bullet$" and "show B assume A $\bullet$".

Notwithstanding this, **what seems natural and logical to us, does not necessarily mean that it will be better for an ATP for verification** (cf. 4.7.2). An example translating MathAbs to first order formulas is given in §4.9.

4.7.2 Proof Checking Limitations and Future Directions

Unfortunately, these first-order formulas (or even higher order formulas) can hardly be validated because of the following reasons (but not limited to):

1. The reasoning gaps.
2. The reasoning details which do not match with underlining rules of the logic or calculus of ATP.
3. Textual proofs never give the exact list of hypotheses and definitions necessary at each step. Therefore, ATP need to search for the appropriate hypotheses, definitions and axioms. However, such proof search will often fail because this information can dramatically extend the search space of ATP.
4. Proofs similar to our running example contains equations. Guiding ATP with justifications for equational reasoning is even more difficult. Because of it, we need very strong equational reasoning which no ATP provides.

For the *proof search problem*, we should have a language for justifications, which not only reflects the information of the textual proofs but can also be used by ATP. In our opinion it will allow us to check more complex proofs. A starting point for such work would be the paper "Premise Selection in the Naproche System" [Cramer *et al.* 2010a]. However it seems worth mentioning that this work only deals with the proofs taken from the first chapter of Landau's book "Foundations of Analysis" [Landau 1966]. The writing style of this book is very simple, which we do not see in the other published books. Landau tries to make the text unambiguous and the proofs presented in it do not have very big reasoning gaps. Having said that, this work [Cramer *et al.* 2010a] is still worthy to notice and see how they manage to select the needed hypotheses, definitions, axioms, etc, from the large poll.

In contrast, the *proof planning* technique [Bundy 1996] could be employed. It also guides the search for a proof in ATP. As a results it has a capacity to perform somewhat better for mathematical texts (as demonstrated by Zinn in his dissertation [Zinn 2004], but only on one proof).

In future, we would also like to translate MathAbs of some mathematical text in the language of theorem provers, such as Mizar [Trybulec *et al.* 1973], Isabelle [Paulson 1994], HOL [Gordon & Melham 1993], Coq [Team 2010], etc. For the reasoning gaps, it will require the use of automatic tactics that these systems provide.

This approach is already employed in the work of MathLang [Kamareddine & Wells 2008] in which they have tried to translate some mathematical text from its core language to the syntax of three proof assistants

(Mizar [Kamareddine *et al.* 2007] or in *formal proof sketches* [Wiedijk 2003], Coq [Kamareddine *et al.* 2008] and Isabelle [Lamar *et al.* 2009]).

Although these translations are still in embryonic stage and the treated examples have limited reasoning gaps (the first chapter of Landau's book of analysis) but inspires us for the possibility of the similar work in MathNat.

4.8 Conclusion

In this chapter, we have given a detailed account of MathAbs, its formal definition and semantics. The usefulness of an intermediary language between the natural language of the mathematician and the formal language of the logician is evident. MathAbs is a formal language which does not contain natural language elements at all, yet tries to faithfully represent the language of mathematics. Furthermore, the view of a mathematical proof as a way of explaining how the "context" evolves seems promising.

In terms of processing, it is intended to be simpler than the language of mathematics. One does not have to learn MathAbs, as it is a part MathNat system and it is not visible to the end-user.

To answer the question of MathAbs' abstraction being expressive enough, the work presented here seems adequate in principle. However in practice, we have formalized only a few examples in MathAbs, which may not be sufficient to make a definite conclusion. At least we can safely claim that the initial results seem very promising.

Other than the future directions that we have already mentioned, the introduction of meta variables[6] in MathAbs to take into account the use of known and unknown variables would be worthwhile and interesting. However, such work will not be easy because it prohibits to check each rule in a proof tree separately; as the check becomes global.

4.9 Appendix

4.9.1 An Example:

We translate the theorem and its proof to first order formulas. This theorem and proof is given in 2.4 on page 14.

1. **Theorem 43 (PYTHAGORAS' THEOREM)** $\sqrt{2}$ is irrational.
2. **Third Proof.** Assume that $\sqrt{2}$ is a rational number.
3. By the definition of rational numbers, we can assume that $\sqrt{2} = a/b$ where a and b are non-zero integers with no common factor.
4. Thus, $b * \sqrt{2} = a$.
5. Squaring both sides yields $2 * b^2 = a^2 \ - \ (1)$.
6. It is clear that a^2 is even because it is a multiple of 2.
7. So we can write $a = 2 * c$, where c is an integer.
8. We get $2 * b^2 = (2 * c)^2 = 4 * c^2$, by substituting the value of a into equation (1).
9. Dividing both sides by 2, yields $b^2 = 2 * c^2$.

[6] Alternatively we can call them existential variables.

10. Thus b is even because 2 is a factor of b^2.
11. If a and b are even then they have a common factor.
12. It is a contradiction.
13. Therefore, we conclude that $\sqrt{2}$ is an irrational number.
14. This concludes the proof.

MathAbs

1. Theorem. show $\neg\sqrt{2} : \mathbb{Q}$
2. Proof. assume $\sqrt{2} : \mathbb{Q}$
3. let $a, b : \mathbb{Z}$ assume $\sqrt{2} = a/b$ assume `positive`(a) $\wedge$ `positive`(b) $\wedge$ `no_cmn_factor`(a, b) `by def rational_Number`
4. deduce $b\sqrt{2} = a$
5. deduce $2b^2 = a^2$ `1 by oper squaring_both_sides`$(b\sqrt{2} = a)$
6. deduce `multiple_of`$(a^2$`, 2)`
   deduce `even`(a^2) `by form multiple_of`$(a^2$`, 2)`
7. let $c \in$ `Z` assume $a = 2c$
8. deduce $2b^2 = (2c)^2 = 4c^2$ `by oper substitution`$(a, 2b^2 = a^2)$
9. deduce $b^2 = 2c^2$ `by oper division`$(2, 2b^2 = (2c)^2 = 4c^2)$
10. deduce `factor_of (2,` $b^2)$
    deduce `even`(b) `by form factor_of`$(2, b^2)$
11. deduce `(even`(a) $\wedge$ `even`(b)`)` $\Rightarrow$ `one_cmn_factor`(a, b)
12. show $\perp$ trivial

Remarks:

- At line 6, first, we deduce the justification i.e. `multiple_of`$(a^2$`, 2)`, and then deduce the whole statement. Same applies to 10.
- However, the above rule does not apply to definitional references and operations as shown in 3, 5, 8, 9.
- We can safely ignore Line 13 and 14 of figure 2.4 because the proof is already finished on line 12.

First Order Formulas:

2. $\vdash (\texttt{rational}(\sqrt{2}) \Rightarrow \texttt{irrational}(\sqrt{2})) \Rightarrow \texttt{irrational}(\sqrt{2})$
3. $\Gamma_1 \vdash \forall_{a,b}((\texttt{int}(a) \wedge \texttt{int}(b) \wedge \sqrt{2} = a/b \wedge a, b > 0 \wedge \texttt{gcd}(a, b) = 1) \Rightarrow \texttt{irrational}(\sqrt{2})) \Rightarrow \texttt{irrational}(\sqrt{2})$
   where $\Gamma_1 \equiv \texttt{rational}(\sqrt{2})$
4. $\Gamma_2 \vdash (b\sqrt{2} = a \wedge (b\sqrt{2} = a \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
   $\Gamma_2 \vdash b\sqrt{2} = a$
   where $\Gamma_2 \equiv \Gamma_1, \forall a, b\ (\texttt{int}(a) \wedge \texttt{int}(b) \wedge \sqrt{2} = a/b \wedge a, b > 0 \wedge \texttt{gcd}(a, b) = 1)$

5. $\Gamma_3 \vdash (2b^2 = a^2 \wedge (2b^2 = a^2 \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
   $\Gamma_3 \vdash 2b^2 = a^2$
   where $\Gamma_3 \equiv \Gamma_2, b\sqrt{2} = a$
6. $\Gamma_4 \vdash (\texttt{multiple_of}(a^2, 2) \wedge (\texttt{multiple_of}(a^2, 2) \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
   $\Gamma_4 \vdash \texttt{multiple_of}(a^2, 2)$
   where $\Gamma_4 \equiv \Gamma_3, 2b^2 = a^2$
   $\Gamma_5 \vdash (\texttt{even}(a^2) \wedge (\texttt{even}(a^2) \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
   $\Gamma_5 \vdash \texttt{even}(a^2)$
   where $\Gamma_5 \equiv \Gamma_4, \texttt{multiple_of}(a^2, 2)$
7. $\Gamma_6 \vdash \forall c((\texttt{int}(c) \wedge a = 2c) \Rightarrow \texttt{irrational}(\sqrt{2})) \Rightarrow \texttt{irrational}(\sqrt{2})$
   where $\Gamma_6 \equiv \Gamma_5, \texttt{even}(a^2)$
8. $\Gamma_7 \vdash (2b^2 = (2c)^2 = 4c^2 \wedge (2b^2 = (2c)^2 = 4c^2 \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
   $\Gamma_7 \vdash 2b^2 = (2c)^2 = 4c^2$
   where $\Gamma_7 \equiv \Gamma_6, \texttt{int}(c) \wedge a = 2c$
9. $\Gamma_8 \vdash (b^2 = 2c^2 \wedge (b^2 = 2c^2 \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
   $\Gamma_8 \vdash b^2 = 2c^2$
   where $\Gamma_8 \equiv \Gamma_7, (2b^2 = (2c)^2 = 4c^2)$
10. $\Gamma_9 \vdash (\texttt{factor_of}(2, b^2) \wedge (\texttt{factor_of}\ (2, b^2) \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
    $\Gamma_9 \vdash \texttt{factor_of}(2, b^2)$
    where $\Gamma_9 \equiv \Gamma_8, (b^2 = 2c^2)$
    $\Gamma_{10} \vdash (\texttt{even}(b)) \wedge (\texttt{even}(b) \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
    $\Gamma_{10} \vdash \texttt{even}(b)$
    where $\Gamma_{10} \equiv \Gamma_9, \texttt{factor_of}\ (2, b^2)$
11. $\Gamma_{11} \vdash ((\texttt{even}(a) \wedge \texttt{even}(b) \Rightarrow \texttt{one_cmn_factor}(a, b)) \wedge ((\texttt{even}(a) \wedge \texttt{even}(b) \Rightarrow \texttt{one_cmn_factor}(a, b)) \Rightarrow \texttt{irrational}(\sqrt{2}))) \Rightarrow \texttt{irrational}(\sqrt{2})*$
    $\Gamma_{11} \vdash \texttt{even}(a) \wedge \texttt{even}(b) \Rightarrow \texttt{one_cmn_factor}(a, b)$
    where $\Gamma_{11} \equiv \Gamma_{10}, \texttt{even}(b)$
12. $\Gamma_{12} \vdash (\perp \Rightarrow \texttt{irrational}(\sqrt{2}))$
    $\Gamma_{12} \vdash \perp$
    where $\Gamma_{12} \equiv \Gamma_{11}, (\texttt{even}(a) \wedge \texttt{even}(b) \Rightarrow \texttt{one_cmn_factor}(a, b))$

Remarks:

- Since deduce A is a syntactic sugar of { show A trivial ; assume A ... }, it produces a lot of tautologies of the form $(A \wedge (A \Rightarrow B)) \Rightarrow B$ in the first-order formulas. Where B is the main goal to prove. They are marked with `*` above.
- In proof, if we add in a sentence such as "proof by contradiction" this adds a MathAbs command show $\perp$ that would replace the conclusion $\texttt{irrational}(\sqrt{2})$ by $\perp$.
- The justifications such as "`by def rational_Number`" that are preserved in MathAbs were removed from the first order translation, because most of the automated theorem provers are unable to use such justifications.
- In the above first order formulas, types are treated as predicates. See §4.7.1.4 on page 82.

The Controlled Language for Mathematics

Chapter 5

The Controlled Language for Mathematics

Contents

5.1 Introduction **91**

5.2 Grammatical Framework **92**

5.2.1 Getting Started 93

5.3 Overview of CLM **96**

5.3.1 Macro Level 96

5.3.2 Micro Level 99

5.1 Introduction

The **C**ontrolled **L**anguage for **M**athematics (CLM) is a computer processable subset of English for writing mathematics which is found in the mathematical texts. Since mathematical language mainly consists of natural language, symbolic expressions and notations, CLM supports a subset of symbolic mathematics as well. It is a precisely defined subset of English with restricted grammar, dictionary, style and predefined conventions.

We support numerous linguistic features such as anaphoric pronouns and references, paraphrasing, rephrasing of a sentence, and the proper handling of distributive and collective readings. Such features give CLM the impression of being informal though the language is in fact formal and machine executable. Furthermore, the grammar and example texts given in this chapter reveal the efforts that we made to parse real mathematical texts. For a user, CLM does not presuppose any expertise in formal logic or computational linguistics.

The aim of subsequent chapters is to describe the CLM grammar and its implementation in detail. It may benefit the reader in two ways: First, the grammar description allows the user to remain within the scope of the grammar when writing mathematical text in MathNat[1]. Second, its implementation allows the user to understand the inner working of the grammar, extend it or even define similar grammars for other domains.

A brief introduction to GF was given in §2.5.1. In the following section, we motivate its use and extend its introduction a little further. In Chapter 6, we describe and define the micro level grammar for CLM. In contrast, we describe and define the macro level

[1] A useful tool in this respect could be the 'auto-completion utility' for parsing that GF provides. But we do not use it in the current version of MathNat software. It is because the current version is more like a parser rather than an interpretor. Until a dedicated MathNat shell is developed, we can use GF shell for this purpose.

grammar for CLM in Chapter 7. A reader unacquainted with GF must read a few sections of Chapter 6 first. After that the reader may go back and forth from macro to micro and from micro to macro level grammar. For that we give an overview of CLM in §5.3.

Controlled languages usually simplify their constructions and interpretation using conventions, which an author must follow. It is the same for CLM, whose conventions are already mentioned in §2.4 and in §2.5.2 informally. Such conventions allow resolving ambiguities and support various linguistic features. A somewhat formal description of these conventions is given in §7.5.

Before we proceed to the next section, another point worth nothing is: there might be other ways to define the same grammar. Similarly, there might be several ways in which we can group the rules of the CLM grammar that we present in this chapter. However, our classification presented here is mainly motivated by the simplicity, which in return, makes the grammar easily explainable.

In the course of the next few chapters, we will sometimes use rather artificial examples. Such examples should not be taken as a weakness, but rather the worst case scenarios. They usually do not occur often but we must prepare for them for the sake of completeness.

Also note that the rules implemented in the next two chapters, may have many similar variants which have not been implemented yet. Some of them could be added with little effort. However, it is a never ending quest, and we had to stop the development to write this report. Of course, the CLM will keep growing and evolving in future.

5.2 Grammatical Framework

There are many special purpose languages and formalisms designed for defining grammars. One well-known such formalism is context-free grammar. However, the context-free grammar can only express mathematical language for small and isolated fragments (for instance, see [Ganesalingam 2009] for such a treatment). Nevertheless, it is also true for any natural language grammar in general. It is because context-free grammar cannot scale up easily to the richness of the language of mathematics (or the natural language) in a working system which covers a large grammar. Even if it does, the implementation would be rather inelegant, clumsy and difficult to manage. For instance, in a context-free grammar the rich features such as agreement, case constraints, subcategorization, selection, etc, cannot be represented compactly. So, it would be difficult to avoid combinatorial explosion of implementation code for these features. Therefore, we need an attribute grammar [Knuth 1968, Deransart *et al.* 1988] to capture the language of mathematics in an effective, intuitive, elegant and manageable way. As a side note, it is worth emphasizing that the problem is not really the context-free grammar but rather its expressive power in terms of the compactness of the written grammar. We need a high level language which offers more expressive power such as the attribute grammar. Nevertheless, it has limitations as well which we elaborate in the course of coming paragraphs.

As we discuss in §2.5, Grammatical Framework (GF) [Ranta 2004, Ranta *et al.* 2010, Ranta 2011a], is a programming language for defining natural language grammars. It allows describing attribute grammar. Moreover, it is a typed functional programming language and a Curry-style categorial grammar formalism [Curry 1961] based on Martin-

Löf's dependent type theory [Martin-Löf 1984].

GF is specifically designed to describe domain-specific or controlled natural language grammars. That is why we choose to implement CLM (**C**ontrolled **L**anguage for **M**athematics) in it; see the last paragraph of this section for a detailed list of reasons.

The grammar designed in GF for CLM only deals with syntax in a rather liberal way. It means that in GF we do not treat semantically motivated constraints that appear on micro and macro levels.

For the micro or sentence level grammar, we may have statements which are well-formed linguistically but logically or mathematically ill-formed (cf. §2.5 and §8.2). Sometimes, it is not easy (or preferred) to enforce such constraints in GF, because it is not general enough such as Haskell or Java. Of course, we can enforce a lost of such semantically motivated constraints using dependent types. But in some cases, it would make the abstract syntax quite complicated. Therefore we enforce them in the host system MathNat (cf. §8.2 for further details).

On the macro or document level, we have structural blocks such as theorem, proof, etc. They have semantics that is hard to capture in an attribute grammar. Therefore, we treat structural blocks in GF grammar as only lists of sentences. Instead, we build the discourse of structural blocks, apply semantically motivated grammatical constraints, provide the miscellaneous linguistic features and give semantics in the host system MathNat (cf. Chapter 8).

A point worth noting is that: there is a difference between the expressive limitations of GF (what we can implement in GF) and what we prefer not to implement in GF. We do not want to capture semantics in the grammar anyway. It is because, when something is rejected in GF, the error message is not personalized. But if we reject something in the host system MathNat we can report a better and specific error message.

It is worth emphasizing that, GF is still best suited for our requirements. It is because, there are only few frameworks which successfully can parse natural language, and GF is one of them. There are also the following non-exhaustive reasons which make it the best candidate:

- As we've already mentioned, it is specifically designed to describe domain specific or controlled grammars [Angelov & Ranta 2010].
- It supports multilingual grammars. For multilingual grammars, it also provides a resource library [Ranta 2009a] to ease this task. We do not use it yet but see §9.2.3 for the future possibilities.
- Its syntax is compact and human friendly to code. Furthermore, the implemented code easily scales up.
- It support auto-completion for parsing.
- It is under active development, therefore, it will get even better with the passage of time, providing more robust parsing and better tools.
- There is an active community that can help.

5.2.1 Getting Started

To get started, we relate a typical excerpt of the grammar for a simple proposition in context-free notation. It is only limited to describe propositions such as "they are

even integers”, “it is a positive number”, etc. For the sake of readability, we keep this excerpt very simple and ignore the number agreement between subject and attribute in the context-free grammar which is presented in figure 5.1. We also ignore the number parameter of `Attribute`, `Type` and `Subject` in the context-free grammar.

```
1  Proposition ::= Subject "is" Attribute
2
3  Attribute   ::= Property Type
4
5  Property    ::= "even" | "odd" | ...
6  Type        ::= "integer" | "number" | ...
7
8  Subject     ::= "it" | "they"
```

Figure 5.1: Context-free grammar for simple propositions

```
1   cat Proposition; Subject; Attribute;
2   cat Property; Type; Pron;
3
4   fun MkProp: Subject -> Attribute -> Proposition;
5
6   fun MkSubj: Pron -> Subject;
7   fun It: Pron;
8   fun They: Pron;
9
10  fun MkAttrib: Property -> Type -> Attribute;
11
12  fun Even: Property;
13  fun Odd: Property;
14  ...
15
16  fun Integer: Type;
17  fun Number: Type;
18  ...
```

Figure 5.2: Abstract Syntax for simple propositions

It becomes in GF a pair of an abstract and a concrete syntax rules. First we give the rules for abstract syntax in figure 5.2. First, we need to define these categories with keyword `cat`, as shown on lines 1–2. The keyword `fun` stands for function declaration. So on line 3, `fun` declares the function `MkProp` of type `Subject -> Attribute -> Proposition`; meaning it takes two parameters (a subject and an attribute) and forms a proposition. The arrow `->` is the normal function type arrow of programming languages. A subject is formed by pronouns `It` or `They`, as shown on lines 6 to 8.

Similarly, `MkAttrib` function forms an attribute with a `Property` and `Type` on line 7. Next, we define functions `Even` and `Odd` of category `Property`; and functions `Integer` and `Number` of category `Type`. In full CLM grammar, we add properties (e.g. positive, odd, distinct, equal, etc) and types (e.g. rational, natural number, set, proposition, etc) in a similar fashion; more in Chapter 6.

To map this abstract syntax into its concrete syntax, we define a set of linguistic objects corresponding to the above categories and functions as shown in figure 5.3. The keyword `lin` stands for linearization. In lines 1–2, we say that categories `Proposition` and `Property` are simply string records. Lines 4–5, show this fact for the linearization

```
1   lincat Proposition = {s:Str};
2   lincat Property = {s:Str} ;
3
4   lin Even = {s ="even"};
5   lin Odd =  {s ="odd"};
6
7   lincat Subject = {s:Str ; n:Number};
8
9   lin It = {s="it" ; n=Sg};
10  lin They = {s="they" ; n=Pl};
11
12  lincat Type      = {s : Number => Str};
13  lincat Attribute = {s : Number => Str};
14
15  lin Integer = {s= table {
16                     Sg => "integer" ;
17                     Pl => "integers"
18                    }
19               };
20  lin MkAttrib prop type = {s = table {
21                Sg => artIndef ++ prop.s ++ type.s!Sg;
22                Pl =>             prop.s ++ type.s!Pl
23                }
24              };
25
26  lin MkProp subj attrb = {s: subj.s ++ be.s!subj.n ++ attrb.s!subj.n};
27
28  oper be = {s = table {Sg => "is" ; Pl => "are"}};
29  param Number = Sg | Pl ;
```

Figure 5.3: Concrete Syntax for simple propositions

of functions `Even` and `Odd`. In line 7, we describe the category `Subject` as a record containing a string and a number (which can be singular or plural, as defined in the last line).

Lines 9–10, define the linearization of the functions `It` and `They`. Here we mention the fact that the function `It` is singular and the function `They` is plural, which will help us to make the number agreement in other function linearizations.

The linearization of the categories `Type` and `Attribute` on lines 12–13, is an inflection table (a finite function) from number to string, having one string value for each (singular and plural). So the function `Integer` of the category `Type`, fills this table with two appropriate values as shown on lines 15–19.

In a similar way, on lines 20–24, we define the linearization of the function `MkAttrib`. For instance, for singular value, we select the string value of the category `Property` (props) with `(.s)`. Then, we select the singular string value of the category `Type` with `(type.s!Sg)`. The operator `(++)` concatenates these two strings along-with a space between them. The `artIndef` makes an agreement for an indefinite article with the first letter of next word; e.g. producing "an even integer" or "a positive number".

Finally, to form the linearization of a proposition, in function `(MkProp subj attrb)` on line 26 and 27, we select appropriate string values of table `be` (given on line 29) and attribute (attrb) by an agreement of number with subject (`be.s!subj.n` and `attrb.s!subj.n` respectively), and concatenate them with subject (`subj.s`).

If we parse the propositions such as "they are even integers" and "it is a positive number", we'll get abstract syntax trees (AST) given in figure 5.4.

```
MkProp (MkSubj They) (MkAttrib Even Integer)
MkProp (MkSubj It) (MkAttrib Positive Number)
```

Figure 5.4: Abstract syntax trees (AST) of two simple propositions

5.3 Overview of CLM

We hereby present an overview of the controlled language of mathematics. It is an almost complete skeleton of the grammar.

5.3.1 Macro Level

This section corresponds to Chapter 7 on page 155.

Theorem and its Proof: §7.2, page 155.

Proof: §7.2.1, page 156.

1. Restatements formed by rules:

   (a) Page 157. Key phrases `ShallProveThat` (§7.2.1.1, page 157), `Statements` (§6.3.6.1, page 144) and `Subordinate` (§7.2.1.1, page 158).

   "we have to prove that $x > y$, $x = y$ or $x < y$, where y is an integer",
   "it is sufficient to prove that $x > y$", etc.

(b) Page 158. Key phrases `ShallProveThat` (§7.2.1.1, page 157), `EStatements` (§6.3.6.2, page 145) and `Subordinate` (§7.2.1.1, page 158).

"we shall prove that either $x > y$ or $y > z$", etc.

(c) Page 159. Key phrases `ShallProveThat` (§7.2.1.1, page 157), `IfthenStmnt` (§6.3.7, page 146) and `SubordinateIfThen` (§7.2.1.1, page 159).

"it is sufficient to prove that if $x \in A$ then $x \in B$ where $A \subseteq B$", etc.

2. Assumptions formed by rules:

(a) Page 159. Keyword 'let', `LetStatements` (§6.3.6.3, page 146) and `Subordinate` (§7.2.1.1, page 158).

"let A and B be two sets, and $A \subseteq B$",
"let m and n be relatively prime integers",
"let $m = n$, and $n = r$", etc.

(b) Page 159. Key phrases `AsmThat`, `Statements` (§6.3.6.1, page 144) and `Subordinate` (§7.2.1.1, page 158).

"suppose that there are two integers u and v such that $u * n + v * m = 1$",
"assume that $\sqrt{2} = \frac{a}{b}$, where a and b are non zero integers with no common factor",
"we assume that $x > a$, $x > b$, and $x > c$, where a, b and c are integers", etc.

(c) Page 160. Key phrases `AsmThat`, `EStatements` (§6.3.6.2, page 145) and `Subordinate` (§7.2.1.1, page 158).

"assume that either m and n are relatively prime, or they are not relatively prime", etc.

3. Assumptions with justifications formed by rules:

(a) Page 160. Some key phrases, `Statements` (§6.3.6.1, page 144), `Justifications` (§6.3.9.3 on page 152) and `Subordinate` (§7.2.1.1, page 158).

"we suppose that $\sqrt{2} = \frac{a}{b}$ by the definition of rational number", etc.

(b) Page 160. Some key phrases, `Justifications` (§6.3.9.3 on page 152), `Statements` (§6.3.6.1, page 144) and `Subordinate` (§7.2.1.1, page 158).

"we can write that $2 * b^2 = (2 * c)^2 = 4 * c^2$ by substituting the value of a into equation (i)",
"by the definition of rational numbers, we suppose that $\sqrt{2} = \frac{a}{b}$", etc.

4. Deductions formed by rules:

(a) Page 162. Key phrases `ConcludeThat` (page 162), `Statements` (§6.3.6.1, page 144) and `Subordinate` (§7.2.1.1, page 158).

"we conclude that $\sqrt{2}$ is an irrational number",
"it implies that m and r are coprime, and $r < m$", etc.

(b) Page 162. Key phrases `ConcludeThat` (page 162), `EStatements` (§6.3.6.2, page 145) and `Subordinate` (§7.2.1.1, page 158).

"either $\sqrt{2}$ is an irrational number or $\sqrt{2}$ is not an irrational number", etc.

5. Deductions with justifications formed by rules:

(a) Page 163. Key phrases `ConcludeThat` (page 162), `Statements` (§6.3.6.1, page 144), `Justifications` (§6.3.9.3 on page 152) and `Subordinate` (§7.2.1.1, page 158).

"a^2 is even because it is a multiple of 2",
"there exist q and r such that $n = m * q + r$ by euclidean division",
"q divides r because $r = n - m * q$", etc.

(b) Page 163. `Justifications` (§6.3.9.3 on page 152), `Statements` (§6.3.6.1, page 144), optional key phrases `ConcludeThat` (page 162) and `Subordinate` (§7.2.1.1, page 158).

"because $p|a * b$ and $p|p * b$ it is clear that $p|b$",
"by induction hypothesis, there are u' and v' such that $u' * m + v' * r = 1$", etc.

(c) Page 163. `Justifications` (§6.3.9.3 on page 152), key phrases `ShowThat`, `Statements` (§6.3.6.1, page 144) and `Subordinate` (§7.2.1.1, page 158).

"dividing both sides by 2 and the fact that x is even, yields the result that $b^2 = 2 * c^2$",
"the fact that x is even and $y = x$ shows the result that y is even", etc.

(d) Page 165. `Statements` (§6.3.6.1, page 144) (as justifications), `Statements` (as conclusions) and `Subordinate` (§7.2.1.1, page 158).

"since $A \subseteq A \cup B$, then $x \in A \cap B$",
"since $A \subseteq A \cup B$ and $B \subseteq A \cup B$ then $A = B$", etc.

6. Miscellaneous proof Statements in §7.2.1.6 on page 166.

7. Proof by Cases in §7.2.1.7 on page 167.

Theorem: §7.2.2 on page 170.

1. Statement to prove formed by rules:

(a) Page 170. Universal or existential ⟨*Formula*⟩ (figure 6.2 on page 109).

$\forall (x, y : \mathbb{Z})\,(\text{even}(x) \wedge \text{even}(y) \Rightarrow \text{even}(x + y))$
$\forall (A, B : \mathbb{S}\text{et})\,((A \cup B = A \cap B) \Rightarrow A \subseteq B)$
$\forall x : \mathbb{Z}\,(\exists\, y : \mathbb{Z}(x + 1 = y))$, etc.

(b) Page 171. Key phrases, (`Statements`(page 144) | `EStatements`(page 145)), `Subordinate` (page 158).

"prove that there exists two integers u and v such that $u * n + v * m = 1$",
"either $x > y$, $x = y$ or $x < y$, where y is an integer",
"show that $x + y$ is even", etc.

(c) Page 171. Key phrases, (§6.3.7, page 146) and `SubordinateIfThen` (§7.2.1.1, page 159).

“prove that if $x \in A$ then $x \in B$”,
“show that if $A \cup B = A \cap B$ then $A \subseteq B$”,
“if a is integer then there is no integer x such that $a < x < a + 1$”, etc.

(d) Page 172. Key phrases, an optional `Typ` (§6.3.1.5 on page 117), `Exps` (§6.2.1 on page 111), (`Statements` | `EStatements` | `IfthenStmnt`) and `Subordinate`.

“prove that for (all | every | arbitrary) integer[s] x, x is positive and even”,
“for (all | every | arbitrary) number[s] x and y, x is an element of y”, etc.

2. Assumptions in theorem on page 173. Same as ‘assumptions in proof’.

3. Miscellaneous key phrases for theorem statements on page 173.

Axiom: §7.3, page 173. We reuse all the statements from theorem for axiom after removing the key phrases “(prove | show) that”).

Definition: §7.4, page 174.

1. Formed by conditional statements on page 175:

(a) `Statements` (§6.3.6.1, page 144) and `Statement` (§6.3.6, page 143)

“if $x > 0$ and $y > 0$, then we define x and y to be positive”,
“if $x > 0$ then x is positive”,
“if an integer n is divisible by 2 then it is even”, etc.

(b) Flipping the above categories.

“we define x and y to be positive if $x > 0$ and $y > 0$”,
“x is positive if $x > 0$”,
“we define an integer n to be even if it is divisible by 2”, etc.

(c) A different conditional.

“we define x and y to be positive iff $x > 0$ and $y > 0$”,
“x is positive only if $x > 0$”,
“an integer n is even if and only if it is divisible by 2”, etc.

2. Assumptions and ‘define’ statements on page 176.

5.3.2 Micro Level

Explanation of the categories at Micro Level. It corresponds to Chapter 6 on page 105.

1. Symbolic Mathematics in LBNF grammar defining expressions, equations and formulas on page 105.

2. Symbolic Mathematics in GF grammar on page 110.

3. A list of symbolic expressions (`Exps`) on page 111.

“x, y and z”, “$x + 2, x + y$ and z”, etc.

4. The Equation (`Equation`) and its List (`Equations`) on page 112.

   "$x + y = 3, y > x$ and $x + y < z$",
   "$x + y = 3, y > x$ or $x + y < z$", etc.

5. A few low level constructs on page 113:

   (a) Anaphoric Pronoun (`Pron`) is ('it' | 'they').
   (b) `Subject` is (`Exps` | `Pron`).
   (c) Demonstrative Pronoun (`DemPron`) is ('this' `Typ` | 'these' `Typ`).
   (d) Quantity on page 115 is represented by two categories:
      i. `Quant` is [('a'/'an' | 'two' | 'three' | ...| 'ten' | 'some')].
      ii. `Quant1` is ('no' | 'a'/'an'/'one' | 'two' | 'three' | ...| 'ten' | 'some').
   (e) Type (`Typ`) is (`Set` | `Integer` | `Number` | `Rational` | `Prime` | ...). (page 117).
   (f) `Property` is (`Positive` | `Even` | `Odd` | `Finite` | `Coprime` | `Equal` | `Arbitrary` | ...). (page 117).
   (g) List of properties: `Properties1`, `Properties2` and `EProperties`. (page 119).
   (h) Relational function[2] is (`Element` | `Factor` | `Square` | `Multiple` | `Divisor` | ...). (page 123).

6. Propositions (`Proposition`) in §6.3.2, page 124, formed by rules:

   (a) Page 125. `Subject`, `Quant`, `Properties1` and `Typ`.

      "x is a positive even integer" or "x be a positive even integer",
      "it is an arbitrary positive integer" or "it be an arbitrary positive integer", etc.

   (b) Page 131. `Subject` and `Properties2`.

      "x is positive and even" or "x be positive and even",
      "x and y are even or odd" or "x and y be even or odd", etc.

   (c) Page 131. `Subject` and `EProperties2`.

      "x and y are either even, odd or positive" or
      "x and y be either even, odd or positive", etc.

   (d) Page 132. `Subject`, `Relation` and `Exp`.

      "x is a divisor of y" or "x be a divisor of y",
      "x is not a multiple of y" or "x be not a multiple of y",
      "x, y and z are elements of $x * y * z$" or "x, y and z be elements of $x * y * z$",
      "x is a square of $\sqrt{x}$" or "x be a square of $\sqrt{x}$", etc.

   (e) Page 133. `Subject`, `Rel2` and `Exp`.

      "x divides y",
      "x does not multiply y",
      "x, y and z divide $x * y * z$", etc.

[2] This term is used as a category (§6.3.1.8 on page 123). It has nothing in common with 'relational functions' of mathematics (which can be written as the ratio of two polynomial functions).

(f) Page 134. Optional noun adjuncts, which we add to the first three rules only (a, b and c above).

7. Existential Statements (`PropExist`) in §6.3.3, page 136, formed by rules:

   (a) Page 136. `Quant`, `Properties1`, `Typ`, `Exps` and `Equations`.

   "there (exists | is) a positive number n such that $n * a > b$",
   "there (do not exist | are no) even integers x, y and z such that $x * a > b$, $y * a > b$ (and | or) $z * a > b$", etc.

   (b) Page 139. `Properties1`, `Exps` and `Equations`.

   "there (exists | is) positive even n such that $n * a > b$",
   "there (does not exist | is no) n such that $na > b$",
   "there (exist | are) positive x, y and z such that $x * a > b$ and $y * a > b$", etc.

8. Relational Statements (`PropRel`) in §6.3.4, page 140, formed by rules:

   (a) Page 140. `Subject`, `QuantRel` and `Relation`.

   "x, y and z have a common factor",
   "they have some common multiples",
   "x, y and z have no common divisor", etc.

   (b) Page 141. `Subject`, `Quant1`, `Relation` and `Exps`.

   "x, y and z have a common factor 2",
   "x, y and z have no common divisor d", etc.

   (c) Page 142. Above two functions without word 'common'.

   "n and m have a divisor d",
   "they have no factor 2", etc.

   and

   "n and m have a divisor",
   "they have three factors", etc.

9. Page 142. Equation with an optional reference (`EqWithRef`).

   "$x^2 + y^2 = (2 * a + 1)^2 + (2 * b + 1)^2$ – (1)",
   "$x^2 + y^2 = (2 * a + 1)^2 + (2 * b + 1)^2$ – i",
   "$x^2 = (2 * a + 1)^2$", etc.

10. `Statement` is (`Proposition | PropExist | PropRel | EqWithRef`). (§6.3.6, page 143).

11. The list of statements:

    (a) `Statements` (§6.3.6.1, page 144).

    `Statement`$_1$[, ..., `Statement`$_{n-1}$ (, and |, or) `Statement`$_n$]

    (b) `EStatements` is a list of `Statement` having at least two elements. (§6.3.6.2, page 145).

Either `Statement`$_1$, ..., `Statement`$_{n-1}$ or `Statement`$_n$.

(c) A list `LetStatements` for 'let' statements. (page 146).

12. Conditional Statements (`IfthenStmnt`) is formed by two lists (`Statements`). (Page 146).

"if m and r have a common divisor d then it divides n",
"if $a = 2 * c$, and $4 * c^2 = 2 * b^2$, and $2 * c^2 = b^2$ then b is even"
and a superficial conditional:
"if a is positive, b is negative, and c is even then x is even, y is odd, and $x * y * z = 10$", etc.

13. Take Statement (`TakeStmnt`) is formed by the list (`Equations`). (Page 147).

"we can choose $x := 10$",
"we take $a := x + 1$, $b := y + 1$ and $c := z + 1$", etc.

14. `Justification` is formed by (`Statement | Operation | Anaphor | DefReference`). (Page 147).

 (a) Operations as Justifications (`Operation`) are formed by rules:

  i. Page 148. `Rel1`.

  "(factoring | squaring | ...) [at] both sides",
  "taking (factor | square | ...) (at | from) both sides".

  ii. Page 148. `Rel2` and `Exp`.

  "(multiplying | dividing | ...) both sides by 2",
  "multiplying the equation by x", etc.

  iii. Page 148. `Rel2`, `EqAnaphora` and `Exp`.

  "multiplying the last equation on both sides by 2",
  "multiplying the last equation by 2 at both sides",
  "dividing our first equation by x", etc.

  iv. Page 149. `Rel3`.

  "taking factor from both sides".

  v. Page 149. `Exp` and (`Equation` | `Reference`).

  "substituting [the value of] x in [equation] $x = 2 * b + 1$",
  "substituting [the value of] x into [equation] (i)", etc.

 (b) Anaphoric References (`Anaphor`). §6.3.9.2 on page 149.

 "the first statement", "the last hypothesis",
 "our first equation", "theorem 24", "theorem", etc.

 (c) Reference to Definitions (`DefReference`). §6.3.9.3 on page 151.

 "the definition of [even] number[s]", "the definition of prime(s)",
 "the definition of euclidean division", "induction hypothesis", etc.

15. A non-empty list of Justifications (`Justifications`). Page 152.
    Parts in brackets {...} does not belong to `Justifications`.

    "**by** the definition of positive numbers **and by** euclidean division, {we (conclude | assume) that . . . }",
    "**by** the last statement **and by** substituting x in $y = x + 10$, {. . . }",
    "**by** the first statement **and because** it is even", {. . . }
    "**by** the last statement**, by** substituting x in $y = z * x + 10$ **and because** z is positive, {. . . }", etc.

Chapter 6

Micro Level CLM Grammar

Contents

6.1 Introduction **105**

6.2 Symbolic Mathematics **105**

6.2.1 Symbolic Mathematics in GF 110

6.3 Grammar Implementation for Textual Mathematics **113**

6.3.1 A Few Low Level Constructs 113

6.3.2 Propositions 124

6.3.3 Existential Statements 136

6.3.4 Relational Statements 140

6.3.5 Equation with a Reference 142

6.3.6 Statements 143

6.3.7 Conditional Statement 146

6.3.8 Take Statement 147

6.3.9 Justifications 147

6.1 Introduction

As we describe in §2.5.1, we treat symbolic and textual parts of mathematics separately in the implementation of the CLM grammar. We first describe the grammar for symbolic mathematics in §6.2. Then we describe the grammar for textual mathematics in §6.3.

In the course of defining them, we'll describe them in an abstract way, aided by examples. It obeys the following conventions: A string value will always be in single quotes ‘ ’. The convention [‘text’] means that ‘text’ is optional. The convention (‘text1’ | ‘text2’) means that both ‘text1’ and ‘text2’ are possible and dots (. . .) means that only a few constructions are given for the sake of brevity. The convention of writing `Text` in typewriter face means that it is not a string value. Instead it is either a record, category or function.

6.2 Symbolic Mathematics

As we described in §2.4, CLM is written in ASCII format. For typography of symbolic mathematics, we use ASCIIMath[1], which allows to render them properly on web pages. To describe its typography in a source file, we give below an example statement taken from figure 2.5 on page 15. However, it is worth mentioning that the grammar can still recognize the symbolic parts if we omit quotation marks (‘. . . ‘). For instance, both

[1] ASCIIMath homepage: `http://www1.chapman.edu/~jipsen/mathml/asciimath.html`

LaTeX	ASCII	LaTeX	ASCII
Type Membership ($\in$)	`:`	Assignment ($=$)	`:=`
Implication ($\Rightarrow$ or $\rightarrow$)	`=>`	Conjunction ($\wedge$)	`land`
Disjunction ($\vee$)	`lor`	Universal Quantifier ($\forall$)	`forall`
Existential Quantifier ($\exists$)	`exists`	Equality ($=$)	`=`
Not Equal ($\neq$)	`!=`	Predicate Divides	`|`
Less than ($<$)	`<`	Greater than ($>$)	`>`
Less than or equal ($\leq$)	`<=`	Greater than or equal ($\geq$)	`>=`
Set Membership ($\in$)	`in`	Set Non-membership ($\notin$)	`!in`
Subset Relation ($\subset$)	`sub`	Non Subset Relation ($\not\subset$)	`!sub`
Subset or Equal ($\subseteq$)	`sube`	Not subset or equal ($\not\subseteq$)	`!sube`
Union ($\cup$)	`uu`	Intersection ($\cap$)	`nn`
Plus Symbol ($+$)	`+`	Minus Symbol ($-$)	`-`
Times ($*$)	`*`	Division	`/`
Modulo	`%`	Power	`^`
Negation ($\neg$)	`not`		

Figure 6.1: ASCII Symbol List for Symbolic Mathematics.

examples shown below are parsed by the host system MathNat. See figure 6.1 for the symbol list of ASCIIMath for the formal grammar.

```
If `a = 2*c`, then `4*c^2 = 2*b^2`.
If a = 2*c, then 4*c^2 = 2*b^2.
```

As we described in §2.5.1 on page 18, we define the symbolic parts of mathematics outside GF, as Labelled BNF grammar [Forsberg & Ranta 2004] (shown in figure 6.2). Defining a formal grammar in GF is definitely possible. However, it comes with a slight penalty in efficiency for parsing[2]. It is because GF is general enough to cover both natural language grammars and formal language grammars. Of course this generality has a price in terms of efficiency.

Second, the Labelled BNF grammar we define for symbolic mathematics is used "as it is" for the MathAbs' ⟨*Formula*⟩ (cf. §4.3 on page 60), which saves us translating from the Labelled BNF ⟨*Formula*⟩ (shown as `Formula` in figure 6.2) to the MathAbs ⟨*Formula*⟩.

Also, a specialized parser for symbolic mathematics has better and wider support for various symbols.

Notwithstanding this, it is rather a minor technical issue, in which the current solution has only a slight advantage over the solution of defining the grammar for symbolic mathematics directly in GF. The price we pay over these advantages is the loss of auto completion support in parsing for symbolic mathematics.

The ⟨*Formula*⟩ consists of the symbolic expression `Exp` and the equation `Eq`. However, the Labelled BNF grammar for symbolic mathematics shown in figure 6.2, does not make any distinction between them. This distinction is enforced in the host system as semantic checks (cf. §8.2.7 on page 189).

[2]This penalty in efficiency not huge now. It is because this observation is taken in 2008 and since then GF is improved a lot.

We decide to do so because the symbolic expression `Exp` is the subset of the equation `Eq` in our grammar (cf. figure 6.2, more details in subsequent paragraphs). Therefore, managing it as a single grammar turns out to be simpler and easier than having separate grammars.

We now explain the Labelled BNF grammar shown in figure 6.2. Apart from the labels (for instance, `EType` on line 1 of figure 6.2), a rule in LBNF is an ordinary BNF rule, where the terminal symbols are enclosed in double quotes and nonterminals are written without quotes. Precedence decreases from top to bottom. For the same precedence, formulas are interpreted from right to left (i.e. left associative). See [Forsberg & Ranta 2005] for a detailed technical report on Labelled BNF grammar.

Symbolic Expression in LBNF The symbolic expression `Exp` is mentioned in the LBNF grammar by `Formula8`, `Formula9`, ..., `Formula13` (lines 27–47 in figure 6.2). We give examples for each rule from bottom to top. For instance,

- `Formula13` allows positive and negative variable, integer and decimals; booleans (true and false); and negation:
  - Rules on lines 45–48 in figure allow positive and negative numbers including decimals such as $1, 2.3, 3, -(4), -(5.1)$, etc.
  - Rules on lines 43–44 allow positive and negative variables such as $x, -(x)$, etc.
  - Rules on line 42 allows negation of `Formula13` such as '$\neg A$', '$\neg$ divides(m, n)', etc.
- `Formula12` allows predicates and functions:
  - Rule on line 39 allows predicates and functions such as even(x) for "x is even", multiple$(a, 2)$ for "a is a multiple of 2", gcd(a, b) for "the greatest common divisor of a and b" divides(x, y) for "x divides y", etc.
  - Rule on line 40 is mostly used by MathAbs to describe predicates and functions such as 'element_of$([a, b], y, z)$' for "a and b are elements y and z", etc.
- `Formula8-11` allows terms:
  - Rule on line 37 allows power, e.g. x^2, $(x * y)^2$, $(x/y)^{1/2}$, etc.
  - Rules on lines 33–35. allows multiplication, division and remainder, e.g. $x * y/y$, etc. Because multiplication and division have the same precedence, we interpret them from left to right. For example, $x * y/y$ is interpreted as $(x * y)/y$.
  - Rules on lines 30–31 allow addition and subtraction. Agian, because they have the same precedence, we interpret them from left to right. For example, "$x + y - z$, is interpreted as: $(x + y) - z$", "$x^2 - y + z$ is interpreted as: $((x^2) - y) + z$", etc. We can also use this subtraction symbol for negative numbers. For example: "$x + (-y) + z$".

    These symbols can also be used for sets, $+$ for disjoint union and $-$ for the difference of two sets (see the last paragraph of the section on page 110 and bullet 14 on page 7 respectively for some discussion).

  - Rules on lines 27–28 allow union and intersection for sets, e.g. "$A \cap B \cup C$, interpreted as: $(A \cap B) \cup C$".

Symbolic Equation in LBNF The symbolic equation `Eq` is mentioned in the LBNF grammar by `Formula5`– `Formula7` (lines 11–25 in figure 6.2) and by `Formula` (lines 1–2 in figure 6.2). Note that the term `Eq` is rather broad, covering equalities, inequalities and some notions for sets (such as membership, subset, etc). We give examples for each rule from bottom to top. For instance,

- Rules on lines 20–25 allow miscellaneous operators for sets such as membership, subset, subset or equal, etc. e.g. "$A \subseteq B \cup C$, interpreted as: $A \subseteq (B \cup C)$", "$x \in A$", "$x \in A \cup B \cap C$, interpreted as: $x \in (A \cup (B \cap C))$", etc.

- Rules on lines 15–18 allows inequalities, e.g.
  "$x > y > z$", "$x < y < z$", "$x \leq y$", etc.

- Rules on lines 11–13 allows equalities and predicate "divides", e.g.
  "$x + y = 2 * a + 2 * b = 2 * (a + b)$", "$x \neq y \neq z$", "$x|y$", etc.

- Rules on lines 01–02 allows assignment and type membership, e.g.
  $x := y$, "$x := y^{1/33} + 3 * 2$, interpreted as: $x := (y^{1/33} + (3 * 2))$",
  $x : \mathbb{Z}$, $x, y, z : \mathbb{N}$, $A, B : \mathbb{S}\mathrm{et}$, etc.
  In line 01, `Type` stands for linguistic types given in §6.3.1.5 on page 117.

Miscellaneous Symbolic Formulas in LBNF The miscellaneous symbolic formulas are those which are only used by MathAbs currently. We mention them as `Formula1`–`Formula4` (lines 04–09 in figure 6.2). We give examples for each rule from bottom to top. For instance,

- Rules on lines 08–09 allows Universal and existential quantifiers. e.g.
  $\forall x, y(x, y \in \mathbb{N} \Rightarrow x * y > x + y)$,
  $\exists u, v(u, v \in \mathbb{Z} \wedge u * n + v * m = 1)$, etc.
  Note that `BVars` in a list of variables (bound).

- Rule on line 06 allows conjunction, e.g. "$A \wedge B$", "$x = y \wedge x = z$", etc.

- Rule on line 05 allows disjunction, e.g.
  "$A \vee B \vee C$", interpreted as: "$((A \vee B) \vee C)$", etc.

- Rule on line 04 allows implication, e.g.
  "$A \Rightarrow B \Rightarrow C$, interpreted as: $((A \Rightarrow B) \Rightarrow C)$",
  "$x \in A \wedge x \in B \Rightarrow x \in A \cap B$, interpreted as: $((x \in A) \wedge (x \in B)) \Rightarrow (x \in (A \cap B))$",
  "$A \wedge B \Rightarrow C \Rightarrow D \wedge E$, interpreted as: "$(((A \wedge B) \Rightarrow C) \Rightarrow (D \wedge E))$", etc.

Note that this description seems to suggest that it is possible to parse the expressions which are semantically ill-formed. For instance, "$\neg(A \cap B + x)$", "$((A : \mathbb{Z}) : \mathbb{Z}) : \mathbb{Z}$", etc. It is true on the level of syntax, but in future, we will reject some of these expressions with semantic checks as we do for various semantically motivated conditions in §8.2

```
1  EType.   Formula ::= Formula1 ":" Type ;
2  EAssign. Formula ::= Ident ":=" Formula1 ;
3
4  EImpl. Formula1  ::= Formula1 "=>"   Formula2;
5  EDisj. Formula2  ::= Formula2 "lor"  Formula3 ;
6  EConj. Formula3  ::= Formula3 "land" Formula4 ;
7
8  LUniv.  Formula4 ::= "forall" BVars "(" Formula ")" ;
9  LExist. Formula4 ::= "exists" BVars "(" Formula ")" ;
10
11 EEq.      Formula5 ::= Formula5 "="  Formula6 ;
12 ENotEq.   Formula5 ::= Formula5 "!=" Formula6 ;
13 EDivides. Formula5 ::= Formula5 "|"  Formula6 ;
14
15 ELthen.   Formula6 ::= Formula6 "<"  Formula7 ;
16 EGthen.   Formula6 ::= Formula6 ">"  Formula7 ;
17 ELthenEq. Formula6 ::= Formula6 "<=" Formula7 ;
18 EGthenEq. Formula6 ::= Formula6 ">=" Formula7 ;
19
20 SBelong.      Formula7 ::= Formula8 "in"    Formula8 ;
21 SNotBelong.   Formula7 ::= Formula8 "!in"   Formula8 ;
22 SSubsetEq.    Formula7 ::= Formula8 "sube"  Formula8 ;
23 SNotSubsetEq. Formula7 ::= Formula8 "!sube" Formula8 ;
24 SSubset.      Formula7 ::= Formula8 "sub"   Formula8 ;
25 SNotSubset.   Formula7 ::= Formula8 "!sub"  Formula8 ;
26
27 SUnion.    Formula8 ::= Formula8 "uu" Formula9 ;
28 SIntersec. Formula8 ::= Formula8 "nn" Formula9 ;
29
30 EPlus.  Formula9 ::= Formula9 "+" Formula10 ;
31 EMinus. Formula9 ::= Formula9 "-" Formula10 ;
32
33 ETimes.   Formula10 ::= Formula10 "*" Formula11 ;
34 EDiv.     Formula10 ::= Formula10 "/" Formula11 ;
35 EMod.     Formula10 ::= Formula10 "%" Formula11 ;
36
37 EPower.   Formula11 ::= Formula11 "^" Formula12 ;
38
39 EFun.  Formula12 ::= Ident "(" [Formula] ")" ;
40 EFun1. Formula12 ::= Ident "(" "[" [Formula] "]" "," "[" [Formula] "]" ")" ;
41
42 LTrue.   Formula13 ::= "true" ;
43 LFalse.  Formula13 ::= "false" ;
44 LNot.    Formula13 ::= "not" Formula13 ;
45 Var.     Formula13 ::= Ident ;
46 NegVar.  Formula13 ::= "(" "-" Ident ")";
47 EInt.    Formula13 ::= Integer ;
48 EDec.    Formula13 ::= Double ;
49 ENegInt. Formula13 ::= "(" "-" Integer ")" ;
50 ENegDec. Formula13 ::= "(" "-" Double ")" ;
```

Figure 6.2: Labelled BNF grammar for symbolic mathematics.

on page 182). However, as a worst case, if some of such ill-formed formulas remain undetected by semantic check, they will surely be detected at the proof checking stage.

As we saw in §3.3.1.2 on page 41, conventions are heavily used in symbolic mathematics. The most prominent ones are the notational collision and precedence.

The most evident form of notational collision occurs when we omit an operator as a convention (as we do in case of multiplication) and when we use the same operator for different operations (as we do for "=", which according to the context could represent "equality" or an "assignment"). Therefore, we use following conventions:

1. "$*$" for multiplication, e.g. $x * y$.
2. "=" for equality, e.g. $x = x * y$, $x = y + 1$, etc.
3. ":=" for assignment, e.g. $x := y = 3$ (boolean), $x := y^2/33 + 3 * 2$, etc.

As regards the precedence, it is not possible to define the precedence globally. Therefore we only support the precedence legal for number theory, and somewhat for analysis and set theory (as evident from figure 6.2). Precedence for other domains is missing and the user must use brackets.

Finally, the grammar for `Exp` and `Eq` covers only those symbols and constructs that we have encountered so far in the parsed example set (Appendix A on page 239). However, if a symbol or notation is not supported by the grammar, it could be written using the syntax of the function, allowed by CLM. For instance, the absolute value of a variable x could be written as abs(x), $x+y$ could be written as plus(x, y), etc. However, there will be certain limitations. Most of the semantic procedures described in Chapter 8 on page 179 (such as semantics checks, context building, anaphoric resolution) will not work for such notations.

Before we proceed to the next section, we must describe another limitation. We cannot differentiate the same operator which may have different semantic meaning. For instance, the operator "+" in "$x + y$ for numbers" and "$A + B$ for sets" (a notation for disjoint union), are treated as "plus_symbol(x, y)" and "plus_symbol(A, B)" respectively. The function "plus_symbol" is syntactic rather semantic, which means that, we postpone its semantic analysis[3] until we type-check its MathAbs in future.

6.2.1 Symbolic Mathematics in GF

In contrast to symbolic mathematics in LBNF, the grammar of symbolic mathematics in GF is treated as a string. This string is parsed by the BNF tool separately. We start by describing the grammar implementation of symbolic expressions: We define category `Exp` for it (line 1 below), which simply takes a string and return `Exp` (line 2 below):

```
1   cat Exp ;
2   fun MkExp : String -> Exp ;
```

In its concrete syntax shown below, the first line defines the linearization of category `Exp` as a record containing a variable `s` of type string (i.e. `s : Str`). In the second line, we define the linearization of function `MkExp`. It takes a variable `exp` of type `String` and returns a record for `Exp` which is (`{s : Str}`) . We select the string value of `exp` with

[3] Which could be "plus(x, y)" and "disjoint_union(A, B)" respectively.

(`exp.s`) and concatenate it with quotes (`"‘"`) using operator (`++`), to fill this record. Note that `s` on line 1 and `s` on line 2 after ({) are the same; In line 1, we assign it a type and later on line 2, we assign it a value of that type.

```
1   lincat Exp = {s : Str} ;
2   lin MkExp exp = {s = "‘" ++ exp.s ++ "‘" } ;
```

A List of Symbolic Expressions For instance, "a, b and c" in a statement such as: "we suppose that a, b and c are positive integers". To define it, we have to treat the following three cases:

1. When there is only one element in the list, such as 'x', '$x + y$', '$x * y + x$', etc.
2. When there are two elements separated by a conjunction[4], such as 'x and y', '$x+y$ and $y + z$', etc.
3. When there are more than two elements, such as: "$\underbrace{x,\ y}$ and $\underbrace{\ z\ }$", "$\underbrace{x_1,\ x_2,\ x_3,\ x_4}$ and $\underbrace{\ x_5\ }$", etc. From these two examples, we can separate the list in two parts. The first part contains all the elements which are separated by comma ',' (i.e. the whole list except the last element). The second part contains the last element which is attached to the rest of the list with a conjunction ('and').

Note that the conventional and usual list in the functional programming languages, does capture this richness. For instance, using `[Exp]`, we can produce one of the following two patterns, but cannot combine them together:

- a, b, c.
- a and b and c.

GF overcomes it with special *list* formed in the GF resource grammar library (RGL) [Ranta 2009a] using *discontinuous constituents*. For instance, among others RGL provides `[AP]` and `[NP]` for the lists of adjectival phrases and noun phrases respectively. Like others, the list `[NP]` is rather a shortcut for defining a category `ListNP`, and at least two functions `BaseNP` and `ConsNP`.

At the time of implementing lists for CLM, we were unaware of these lists provided by RGL. Also our implementation for such lists is different than RGL because we do not use discontinuous constituents. We now define our implementation for the list of expressions below.

In GF, we can define our required list category (`Exps`) with the help of another category (`PExps`) as defined in the first line of the following abstract syntax. The category `PExps` stands for a partial or an intermediate list of expressions. The function `BaseExps` on line 5, covers the first case shown above (bullet 1). It takes an expression and makes it a list (`Exps`).

In contrast, the functions `BasePExp` and `ConsExps` cover the second case (bullet 2). First, `BasePExp` (which is the base case of an intermediate list) takes two `Exp` as

[4] We have restricted `Exps` to contain only a conjunction. A disjunction ('or') may occur in some other lists.

parameter and returns an intermediate list `PExps`. In line 6, this intermediate list is then passed to function `ConsExps`, which returns the list `Exps`.

Finally, the functions `ConsPExps` and `ConsExps` cover the third case (bullet 3). First, `ConsPExps` takes an expression and a list of intermediate expressions (`PExps`). It is then given to `ConsExps` which returns the list `Exps`.

```
1  cat PExps ; Exps ;
2  fun BasePExps : Exp -> Exp   -> PExps ;
3      ConsPExps : Exp -> PExps -> PExps ;
4
5      BaseExps : Exp   -> Exps ;
6      ConsExps : PExps -> Exps ;
```

Here is the corresponding concrete syntax. Note that we can easily refactor it to save space and share code. However, the purpose for the moment is to describe the code in simplicity not compactly (But see refactoring in action on page 120).

```
1  lincat PExps = {s : Str } ;
2         Exps  = {s : Str ;   n : Number } ;
3  lin BasePExps x y  = {s = x.s ++ "and" ++  y.s } ;
4      ConsPExps x xs = {s = x.s ++ ","   ++ xs.s } ;
5
6      BaseExps x  = {s = x.s  ;     n = Sg } ;
7      ConsExps xs = {s = xs.s ;     n = Pl } ;
8
9  param Number = Sg | Pl ;
```

The second line describes the linearization of list `Exps` as a record having two values: `s` of type string (`s : Str`) and `n` of type number (`n : Number`). The `Number` defined on line 9 is an algebraic type. It has two constructors: `Sg` for singular and `Pl` for plural[5]. The string `s` contains the textual form of the expression and the number `n` contains information regarding its number (i.e. whether it is singular or plural).

The first line describes the linearization of the partial list `PExps`. It is a record having one string value. Unlike `Exps`, the type `Number` is not needed here because `PExps` is always plural.

The Equation and its List We define symbolic equation `Eq` exactly the same way as we define `Exp`. However the list `Eqs` is somewhat different than `Exps`. In the list `Exps`, only conjunction is allowed between elements, as shown below:

"x, y and z", "$x + 2, x + y$ and z", etc.

But in the list `Eqs`, both conjunction and disjunction are allowed between elements, such as:

"$x + y = 3, y > x$ and $x + y < z$",
"$x + y = 3, y > x$ or $x + y < z$", etc.

[5] In GF, `param` is a keyword for the algebraic types.

The abstract syntax of Eqs is easily constructed from the abstract syntax of the list Exps. First, we have to rename the function BasePExps with ConjPEqs, and then add a new (but) similar function DisjPEqs (for disjunction). Finally, we have to replace string "Exp" with "Eq" everywhere.

```
1  cat PEqs ; Eqs ;
2  fun ConjPEqs : Eq -> Eq   -> PEqs ;
3      DisjPEqs : Eq -> Eq   -> PEqs ;
4      ConsPEqs : Eq -> PEqs -> PEqs ;
5
6      BaseEqs : Eq   -> Eqs ;
7      ConsEqs : PEqs -> Eqs ;
```

Its corresponding concrete syntax is rather straightforward:

```
1  lincat PEqs = {s : Str } ;
2         Eqs  = {s : Str ;   n : Number } ;
3
4  lin ConjPEqs x y  = {s = x.s ++ "and" ++  y.s } ;
5      DisjPEqs x y  = {s = x.s ++ "or"  ++  y.s } ;
6      ConsPEqs x xs = {s = x.s ++ ","   ++ xs.s } ;
7
8      BaseEqs x  = {s = x.s  ;  n = Sg } ;
9      ConsEqs xs = {s = xs.s ;  n = Pl } ;
```

Note that we omit the abstract syntax defining Eq, because of its similarity with Exp). However, if Exp and Eq are exactly the same then why can't we simply use the category Exp for the equation also, especially when they are simply strings in GF? The reason is as followed:

Recall that the ⟨*Formula*⟩ in LBNF consists of the symbolic expression Exp, the equation Eq and miscellaneous symbolic formulas. Having separate categories for expressions and equations in GF and using them in other GF rules for CLM, we record these category slots and apply semantic checks to enforce, let us say an expression for category Exp and equation for category Eq.

6.3 Grammar Implementation for Textual Mathematics

6.3.1 A Few Low Level Constructs

6.3.1.1 Anaphoric Pronoun

Anaphoric Pronoun is ('it' | 'they'). In the abstract syntax, we define category Pron and functions It and They as shown below:

```
1  cat Pron ;
2  fun It   : Pron ;
3  fun They : Pron ;
```

In concrete syntax, Pron is a record, having the field s of type string and the field n of type number. Further, we define the linearization of its functions and mention the fact that It is singular and They is plural, to enable number agreement later.

```
1  lincat Pron = {s : Str ;    n : Number } ;
2  lin It    = {s = "it"   ;   n = Sg} ;
3  lin They  = {s = "they" ;   n = Pl} ;
```

6.3.1.2 Subject

Subject is (`Exps` | `Pron`). The abstract is define as shown below:

```
1  cat Subject ;
2  fun MkExpsSubj    : Exps    -> Subject ;
3      MkPronSubj    : Pron    -> Subject ;
```

In the corresponding concrete syntax, we only need to up-cast these categories to category `Subject`.

```
1  lincat Subject = {s : Str ; n : Number} ;
2  lin MkExpsSubj    exps    = exps    ;
3      MkPronSubj    pron    = pron    ;
```

As we can see, the category `Subject` is a nominal group (Noun Phrase), in which we currently support nominatives only.

Note that, for the moment, only symbolic expressions and equations are supported for object position. Anyway, in modern mathematics, natural language objects are not used frequently. Nevertheless, in future, we must support them. It is because, in general, natural language is a huge collection of less frequently used constructs. If we do not support them, we end up having insufficient coverage.

A good starting point for such subjects and objects could be the constructions used in the example below:

The conjunction of A and B implies their disjunction.
The conjunction of two propositions implies their disjunction.
...

So when we support such constructs mutually shared by subjects and objects, it might be a good idea to give the current category `Subject` some other name (May be `NominalGroup` having a field such as `c:Case` in its record with value `c = Nominative`).

6.3.1.3 Demonstrative Pronoun

Demonstrative Pronoun is (‘this’ `Typ` | ‘these’ `Typ`).
(Note that `Type` is a GF keyword. Therefore for the category for types, we use the name `Typ`. It could be ‘integer(s)’, ‘number(s)’, ‘set(s)’, etc, defined on page 117). A few examples of demonstrative pronouns could be:

“this integer”, “this number”, “these integers”, “these numbers”, etc.

In the abstract syntax, we define category `DemPron` and functions `ThisType` and `TheseType`. These functions takes `Typ` as parameter, as shown below:

```
1 cat DemPron ;
2 fun ThisType  : Typ -> DemPron ;
3 fun TheseType : Typ -> DemPron ;
```

In concrete syntax, similar to `Pron`, `DemPron` is also a record, having the field `s` of type string and the field `n` of type number. Further, we define the linearization of its functions by combining string (‘this’ | ‘these’) with `Typ`, and mention the fact that `ThisType` is singular and `TheseType` is plural, to enable the number agreement later.

```
1  lincat DemPron  = {s : Str ; n : Number} ;
2  lin ThisType  t = {s = "this"  ++ t.s!Sg ; n = Sg} ;
3  lin TheseType t = {s = "these" ++ t.s!Pl ; n = Pl} ;
```

6.3.1.4 Quantity

We have defined two different kinds of quantities. As we see below, there is a little difference between them. But it is important to make this distinction according to the needs of the statements in which we use them.

- Quantity `Quant` is [(‘a’/‘an’ | ‘two’ | ‘three’ | . . . | ‘ten’)]. Note the brackets [...] around it. It means `Quant` can be empty.
- Quantity `Quant1` is (‘no’ | ‘a’/‘an’/‘one’ | ‘two’ | ‘three’ | . . . | ‘ten’ | ‘some’). Because it contains a value ‘no’, `Quant1` cannot be used in negative statements. Also it cannot be empty. Finally, unlike `Quant`, where we have two values ‘a’ or ‘an‘ for a singular, we may have three string values: ‘a’, ‘an’, or ‘one’ in `Quant1`.

Due to the similarities between these quantities, it is desirable to share the GF code between them. So, we now define the common part of these two quantities, which both will share:

```
1 cat QuantC ;
2 fun Two, Three, Four, Five, Six, ... , Ten, Some: QuantC ;
```

The category `Quant1` requires an inherent parameter `Number` for the agreement. Therefore, in the corresponding concrete syntax, the linearization of `QuantC` is a record with fields: string and number, as shown in line 1 below:

```
1 lincat QuantC = {s : Str ; n : Number } ;
2 lin Two   = {s = "two"   ; n = Pl };
3     Three = {s = "three" ; n = Pl };
4     ...
5     Ten   = {s = "ten"   ; n = Pl };
6     Some  = {s = "some"  ; n = Pl };
```

The Quantity `Quant`, is the mostly used quantity and we define the extra functions required for it, in the abstract and concrete syntax below:

```
1  cat Quant ;
2  fun UseQuantC : QuantC -> Quant ;
3      One, Empty : Quant ;
4
5  lincat Quant = {s : Str ; n : Number };
6  lin UseQuantC qc = qc ;                  // up-cast
7      One   = {s = artIndef ; n = Sg } ;
8      Empty = {s = "" ; n = Sg } ;  // singular is meaningless here. Just filling the slot
```

It is used in statements such as following:

Let x and y be positive integers.	(the function `Empty` is used)
Let x and y be **two** positive integers.	(the function `Two` is used)
Let x be **a** positive integer.	(the function `One` is used)

We use a general function `artIndef` in the linearization of `One` on line 7 (above). It is defined in the GF RGL [Ranta 2009a], as shown below:

```
oper artIndef : Str = pre {"a"; "an"/strs {"a"; "e"; "i"; "o"}} ;
```

It is defined as an *operation* (`oper`). The `oper` is similar to `lin` (i.e. the linearization of a function `fun`) but it can be used to define general functions that may be used in other functions.

The oper `artIndef` makes an agreement for an indefinite article with the first letter of the next word. It selects the article 'an' if it is followed by a vowel. Otherwise, it selects the article 'a'. For instance, "an even number", "a positive number", etc.

The Quantity `Quant1` We define quantity `Quant1` for `AdjunctWith` (cf. §6.3.2 on page 134) and for relational statements (cf. §6.3.4 on page 140).

Again, it contains a construct for value 'no', therefore, it cannot be used in negative statements. Also, unlike `Quant`, it does not have a function `Empty`.

Note that the number feature we have saved will be used later to make number agreement. For instance, when forming adjuncts such as "with no common factor", "with two common factors", etc, the number feature of quantity (e.g. 'no', 'two') must be in agreement with a relational function[6] (e.g. 'factor' and 'factors') respectively.

We now give the abstract and concrete syntax below:

```
1  cat Quant1;
2  fun UseQuantC1 : QuantC -> Quant1 ;
3      No, OneQ1 : Quant1 ;
4
5  lincat Quant1 = {s : Str ; n : Number } ;
6  lin UseQuantC1 qc = qc ;                 // up-cast
7      No    = {s = "no" ; n = Sg } ;
8      OneQ1 = {s = variants {artIndef ; "one"} ; n = Sg };
```

The variants construct of GF on line 8, expresses free variation. It allows function `OneQ1` to have two possible string values: ('one' | 'a' or 'an' depending on the first character of the next word).

[6] This term is used as a category (§6.3.1.8 on page 123). It has nothing in common with 'relational functions' of mathematics (which can be written as the ratio of two polynomial functions).

Finally, note that for the functions `fun` in abstract syntax, it is not permitted to share them between different categories. So we cannot use the same function, let us say, `One` in place of `OneQ1` (and add the missing information such as "one").

Once the categories and functions are defined in the abstract syntax, we must define their corresponding linearizations as well (with `lincat` and `lin`). For instance, we must define linearization (`lin`) for `One` and `OneQ1` separately. However, what we can do is to define the shared `oper` functions for common functionality which could be then shared by these functions. For instance, we have used the same `oper` (`artIndef`) in functions `One` and `OneQ1`.

Note that the `lincate` type of categories `QuantC`, `Quant` and `Quant1` are same. So we can define a shared oper for them:

```
lincat QuantC, Quant, Quant1 =  rec_quant ;

oper rec_quant : Type = {s : Str ; n : Number} ;
```

6.3.1.5 Type

The name `Type` is a GF keyword. Therefore, we use another name `Typ` for types.
`Typ` is (`Prop` | `Set` | `Integer` | `Number` | `Tuple` | `Rational` | `Prime` | ...). We define the following abstract syntax for it:

```
1  cat Typ ;
2  fun Prime, Rational, Irrational, Number, Integer, Prop, Set : Typ ;
```

In the concrete syntax, `Typ` is an infection table (a finite function) from number to string (`Number => Str`), labeled as `s`, having one string value for each (singular and plural)[7]. In lines 2–6, we define the linearization of function `Prime`. We save two variants for singular ('prime' and 'prime number') and two for plural ('primes' and 'prime numbers'). We add brackets (`[ ]`) around those token which contain spaces (cf. line 3 and 4 below).

```
1  lincat Typ = {s : Number => Str } ;
2  lin Prime = {s = table {
3        Sg => variants{"prime" ; ["prime number"] } ;
4        Pl => variants{"primes"; ["prime numbers"]}
5        }
6       };
7
8      Integer = {s = table {
9        Sg => "integer" ;
10       Pl => "integers"
11       }
12      };
13     ...
```

6.3.1.6 Property

`Property` is (`Positive` | `Even` | `Odd` | `Finite` | `Coprime` | `Equal` | `Arbitrary` | ...).

[7] We may also say that in the linearization of `Typ`, it is a record containing a variable `s` which has a type `table` from number to string.

We currently do not make any distinction between one-place and two-place properties in the GF grammar for CLM. These properties usually correspond to distributive and collective reading respectively. It is the host system MathNat which makes distinction between them and accordingly treat them. It is further described in §6.3.2.1 on page 129.

We now give a reason for not making any distinction between one-place and two-place properties in the GF grammar for CLM. The properties which are considered two-place or collective, can also be used as one-place properties (but remains collective). For instance, consider the collective property 'equal', in the following example:

1. Suppose that x, y and z are positive, even and equal.

Also, the above three properties are formed by a list. If we define them separately as one-place and two-place properties, then how will we combine them in a list. Therefore, we define all the properties as 'one-place'.

Also note the properties 'distinct' and 'coprime' in the following two examples:

2. Suppose that x, y and z are positive and coprime.
3. Suppose that x, y and z are positive, distinct and coprime.

Both are collective yet inherently 'one-place' properties. For instance, we cannot say "x is coprime to y" or "x is distinct to y" as we can say "x is equal to y".

Also, in the second example when we say that 'x, y and z are coprime', it is clear that these variables are pair-wise prime. But in the the third example, when we say that 'x, y and z are distinct', it is ambiguous. Are these variables pair-wise distinct or just some of them are distinct? Currently we neglect such ambiguity and always translate such properties in MathAbs as 2-arity predicates. As a conclusion, properties require a deeper analysis and a better classification. It is yet to be done for CLM.

For the moment, defining properties in GF is quite straightforward as shown in the abstract and concrete syntax below, followed by explanation:

```
cat Property ;
fun Positive, Negative, Arbitrary, Even, Odd, Finite, Equal, Coprime : Property ;
```

```
1  lincat Property = {s : Str } ;
2  lin Positive  = {s = variants {["non zero"]; "non-zero"; "positive"} } ;
3     Negative  = {s = "negative"  } ;
4     Arbitrary = {s = "arbitrary" } ;
5     ...
```

The linearization of `Positive` on line 2 above can have three string variations: ('non zero' | 'non-zero' | 'positive'). Also note the function `Arbitrary`. It is used in statements such as:

Let x be an arbitrary positive integer.

Syntactically the word "arbitrary" is an adjective similar to the linearization of other functions (such as "positive", "negative", "even", etc). However, semantically its role is different as discussed in §6.3.2.1 on page 125.

6.3.1.7 List of Properties

As per requirements of the high level constructs, we need three kinds of lists for `Property`: `Properties1`, `Properties2` and `EProperties`.

The list `Properties1` is formed by the pattern: "[P_1 P_2 ... P_n]", where P is a property. These properties are separated by space. The brackets [ ] shows that this list can be empty. Some of its examples are shown below, in which `Properties1` are bold-faced:

"Let x be a **positive even** integer.",
"Let x be a **positive odd prime** number.",
"Let x be a number." (no property), etc.

The list `Properties2` is formed by the pattern: "P_1, P_2, ..., (and | or) P_n". As we can see the list is separated by comma and the words "and | or". This list cannot be empty. Some examples of this list is shown below, in which they are bold-faced:

"We assume that x is **positive, even (and | or) odd**.",
"x is **positive (and | or) negative**.", etc.

The list `EProperties` is formed by the pattern: "either P_1, P_2, ..., or P_n". It also cannot be empty. For instance,

"We assume that **either** x is **positive, even or odd**.",
"**Either** x is **positive or negative**.", etc.

First, we define the list `Properties1`. In GF, it is straightforward to define, as shown in the abstract syntax below:

```
1 cat Properties1 ;
2 fun BaseProperties1 : Properties1 ;
3     ConsProperties1 : Property -> Properties1 -> Properties1 ;
```

And the concrete syntax:

```
1 lincat Properties1 = {s = Str } ;
2 lin BaseProperties1 = {s = "" } ;
3     ConsProperties1 p ps = {s = p.s ++ ps.s } ;
```

The `BaseProperty` takes an empty string and forms `Properties1`. In contrast the `ConsProperty` simply joins a property with the list of property (`Properties1`), without adding any other string, forming a list of the form, let us say, 'positive even coprime', etc.

Note that, GF list such as `[Property]` is a short hand for the above category `Properties1`. It has only a very slight advantage over this one: we can skip the functions on lines 2–3 from the abstract syntax. But anyway we would have to provide their linearizations in the concrete syntax as we do on line 2–3.

Second, the list `Properties2` is very similar to the list `Exps` in its structure (§6.2.1 on page 111). Therefore, due to the similarity between `Exps`, `Properties2` and `EProperties` in structure, we now generalize such lists. It would be the parametric list similar to `[Exp]` and `[Property]` but modified to our requirements.

However, the abstract syntax for such categories remains unchanged. It is because, we need these categories and functions to recognize, let us say, conjunction from disjunction for the denotational semantics of lists in the host system MathNat. Also, intermediate categories such as `PProperties`, `PEProperties` and `PExps` has a varying number of functions. If we make a common category for them. We'll produce ill-formed lists. Therefore, we cannot remove them.

The abstract syntax for `Properties2` is shown below. It is similar to the list `Exps`, with an additional function for disjunctions in line 3:

```
1  cat Properties ; PProperties ;
2  fun BaseConjPProperties : Property -> Property    -> PProperties ;
3      BaseDisjPProperties : Property -> Property    -> PProperties ;
4      ConsPProperties     : Property -> PProperties -> PProperties ;
5
6      BaseProperties2 : Property    -> Properties2 ;
7      ConsProperties2 : PProperties -> Properties2 ;
```

Similar to `Exps` and `PExps`, the linearizations of `Properties2` and `PProperties` in the concrete syntax should be define like this:

```
lincat Properties2 = {s : Str ; n : Number} ;
       PProperties = {s : Str} ;
```

The linearization of `Properties2` becomes the following, if we apply two polymorphic records as an `oper`, general to all lists.

```
lincat Properties2 = List ;

oper List : Type = {s : Str ; n : Number} ;
     Elem : Type = {s : Str} ;
```

We now define three low level functions (i.e. `oper`), followed by some explanation:

```
1  oper oneElemList : Number -> Elem -> List = \n,x ->
2           {s = x.s ; n = n } ;
3
4       twoElemList : Str -> Elem -> Elem -> List = \s,x,y ->
5           {s = x.s ++ s ++ y.s } ;
6
7       consList : Str -> Elem -> List -> List = \s,x,xs ->
8           {s = x.s ++ s ++ xs.s } ;
```

The function `oneElemList` takes `Number` and `Elem` (i.e. `Number -> Elem` on line 1 above) and returns the `oper` record `List`. After the equal sign (=) in line 1, we assign

two variable names `n` and `x` to the `Number` and `Elem` mentioned above. These variables are assigned to the number and string fields of the `List` record respectively.

In contrast, the function `twoElemList` takes a string and two elements (`Str -> Elem -> Elem`) and returns the record `{s:Str}`.

The functions belonging to the list `Properties2` and `PProperties` should be defined like this:

```
lin BaseConjPProperties x y = {s = x.s ++ "and" ++ y.s } ;
    BaseDisjPProperties x y = {s = x.s ++ "or"  ++ y.s } ;
    ConsPProperties    p ps = {s = p.s ++ ","   ++ ps.s} ;

    BaseProperties2 x  = {s = x.s  ; n = Sg} ;
    ConsProperties2 xs = {s = xs.s ; n = Pl} ;
```

It becomes following, with newly defined `oper` functions:

```
lin BaseConjPProperties x y = twoElemList "and" x y  ;
    BaseDisjPProperties x y = twoElemList "or"  x y  ;
    ConsPProperties    p ps = consList    ","   p ps ;

    BaseProperties2 x  = oneElemList Sg x   ;
    ConsProperties2 xs = oneElemList Pl xs  ;
```

Or even more concise using partial application:

```
1 lin BaseConjPProperties = twoElemList "and" ;
2     BaseDisjPProperties = twoElemList "or"  ;
3     ....
```

Third, we form the list `EProperties`. Recall that it must have two elements at minimum and it allows to write the following pattern:

"either Property$_1$, Property$_2$, or Property$_n$".

Unlike `Properties1` and `Properties2` which are used in both positive and negative statements (cf. high level constructs on page 124), we only support the list `EProperties` for the positive statements. As far as our observation, its negation is not commonly found in the mathematical texts. Therefore, it not worth the trouble to define the following variations of `EProperties` for the negative statements:

x is either not even or odd. (negation is only applied on property 'even')
x is neither even nor odd. (both properties does not hold for x)
It is not the case that either x is even or odd.
(the negation of whole the statement)

Now, at minimum, we can build `EProperties` in two ways:

1. We form it by reusing `PProperties` in a function `MkEProperties`:

```
cat EProperties ;
fun MkEProperties : PProperties -> EProperties ;
```

Because it takes `PProperties` as a parameter, only the constructions having following patterns are possible.

"Property_1 (and | or) Property_2",
"Property_1, Property_2, ..., (and | or) Property_n".

It means that the list cannot have a single element and it also cannot be empty; we need both conditions for `EProperties`.

In its concrete syntax below, we define `EProperties` similar to other lists for `Property`, as shown on line 1 below. In the linearization of function `MkEProperties`, we simply add a string "either" on the head of the list, as shown on line 2. The second part of this record (`n = Number`) gets the plural value, because this list cannot have a single element or empty.

```
1    lincat EProperties   = List ;
2    lin MkEProperties ps = oneElemList ("either" ++ ps.s) Pl ;
```

However, one problem still remains. It allows to write incorrect statements of the form given below:

"***either** Property_1 **and** Property_2",
"***either** Property_1, Property_2, ..., **and** Property_n".

We have to reject them with the semantic checks (cf. §8.2). **We use this solution in the current implementation** to reuse the code of the category `PProperties`, both in GF and in the host system MathNat (the code written for the denotational semantics).

2. However, there is another possibility. It is cleaner in a sense that we do not have to apply semantic checks in the host system MathNat for it. But we need to do a bit more work both in GF and in the host system MathNat (i.e. translation for denotational semantics).

   Similar to the category `Properties2`, we could create a new category `EProperties` in the following way. As usual, `PEProperties` is an intermediate list. Base and Cons functions of `PEProperties` (lines 2–3, below), makes the following two patterns:

   "Property_1 or Property_2"
   "Property_1, Property_2, or Property_n"

   It is the function `MkEProperties` which adds the clue word 'either' at the head of these patterns.

```
cat PEProperties ; EProperties ;
fun BasePEProperties : Property -> Property     -> PEProperties ;
    ConsPEProperties : Property -> EProperties  -> PEProperties ;

    MkEProperties : PEProperties -> EProperties ;
```

6.3.1.8 Relational Function

Relational function[8] is (`Element` | `Factor` | `Square` | `Multiple` | `Divisor` | ...). Due to the morphological and lexical differences between them, we divide it in the following three classes:

1. Relational function (`Rel1`) such as "square" or "factor". Syntactically, these relational functions belong to the class of **one-place predicates** (or unary function). For instance, we can say "squaring the equation" but cannot say "squaring the equation by 2". Similarly, we can say "factoring y" but cannot say "factoring y by 2".

2. Relation function (`Rel2`) such as "multiple", "divisor". Syntactically, these relational functions belong to the class of **two-place predicates** (or binary function). For instance, we can say "multiplying the equation by 2", etc.

3. Relation function (or predicate) (`Rel3`) such as "element". It is different than the above relational functions in a sense that we cannot say "elementing" as we do for others ("squaring", "multiplying", etc). For instance, we have to say "taking element from both sides of the equation".

These are mostly used in `Operation` (cf. §6.3.9.1 on page 148). In the abstract syntax, we define three categories corresponding to the above three classes:

```
1  cat Rel1 ; Rel2 ; Rel3 ;
2  fun Factor, Square, ...    : Rel1 ;
3      Multiple, Divisor, ... : Rel2 ;
4      Element, ...           : Rel3 ;
```

In concrete syntax, `Rel1` and `Rel2` are records, having a table from number to string (`Number => Str`) and a string field for the participle (`part : Str`) (such as "multipling", "squaring", etc). In contrast, `Rel3` does not have the string participle ("taking element" instead of "elementing").

In the code below, we first define `oper` structure `rel` for `Rel3` (on lines 1 and 5). Then we reuse it and extend its structure with participle (on line 2 using **) for `Rel1` and `Rel2` using the `oper` structure `rel12` (line 4).

```
1  oper rel   : Type = {s : Number => Str } ;
2       rel12 : Type = rel ** {part : Str } ;
3
4  lincat Rel1, Rel2 = rel12 ;
5         Rel3       = rel  ;
```

For the linearization of functions, we define below two reusable `oper` functions. Note that the first function is then also used in the second.

```
1 oper mkNumStrRec : Str -> Str -> rel = \sg,pl ->
2          {s = table {
```

[8] This term is used as a category (§6.3.1.8 on page 123). It has nothing in common with 'relational functions' of mathematics (which can be written as the ratio of two polynomial functions).

```
3               Sg => sg ;
4               Pl => pl
5            }
6        } ;
7    mkRel12 : Str -> Str -> Str -> rel12 = \sg,pl,part ->
8        {s = (mkNumStrRec sg pl).s ; part = part } ;
```

And use them as shown below:

```
1 lin Factor  = mkRel12 "factor" "factors" "factoring" ;
2     ...
3     Divisor = mkRel12 "divisor" "divisors" "dividing" ;
4     ...
5     Element = mkNumStrRec "element" "elements" ;
6     ...
```

We can only find a single function for `Rel3`. However it seems reasonable to define it separately with a hope to find other candidates in future. These categories also stresses the irregularities of the natural languages as well as the controlled or domain specific languages.

However, sometimes we do not need the above mentioned distinction between these relational functions. So for that we combine them in a category `Relation` as shown in the abstract syntax below.

```
1 cat Relation ;
2 fun MkRel1 : Rel1 -> Relation ;
3     MkRel2 : Rel2 -> Relation ;
4     MkRel3 : Rel3 -> Relation ;
```

In the concrete syntax shown below, we do not store the field for the string participle for the linearization of `Relation` and reuse oper structure `rel`. In the linearization of function MkRel1 and MkRel2, we simply reuse the `oper` function `mkNumStrRec`. As its parameters we select the needed string value from the table with selection operator (`!`). For instance, on lines 2–3 below, we retrieve the string value associated as singular from the relational function table (`r.s!Sg`). In the linearization of the function `MkRel3`, because the structure of record field of `Rel3` and `Relation` is same, we simply pass the whole record field as it is.

```
1 lincat Relation =  rel ;
2 lin MkRel1 r = mkNumStrRec r.s!Sg r.s!Pl ;
3     MkRel2 r = mkNumStrRec r.s!Sg r.s!Pl ;
4     MkRel3 r = r ;
```

High Level Constructs

6.3.2 Propositions

We now proceed to build propositions (i.e. category `Proposition`) which we support for positive and negative simple statements. These propositions are used to form theorem statements, proof statements, axiom statements and definition statements, with minor modifications. We define below several rules that construct the proposition.

6.3.2.1 The First Rule for Propositions: Subjects, Types and Properties

It is formed by `Subject`, `Quant`[9], `Properties1` and `Typ`.
A few examples are given below. Note that the second example in each line below is not valid alone. These must be prefixed by "let" to be linguistically as well as mathematically correct[10]. Also note that the negative statements are not supported with "let". So the statement with * in examples below is rejected as semantic check.

"x is a positive even integer" or "x be a positive even integer",
"it is an arbitrary positive integer" or "it be an arbitrary positive integer",
"x and y are two integers" or "x and y be two integers",
"$\sqrt{2}$ is not an integer" or "*$\sqrt{2}$ be not an integer (*not supported*)", etc.

Its abstract syntax given below is self explanatory:

```
1 cat Proposition ;
2 fun MkPosProp1 : Subject -> Quant -> Properties1 -> Typ -> Proposition ;
```

In its concrete syntax, proposition is defined as a table of string values which depends on the type of the statement (`StmntType => Str`), as shown below on line 1. The `StmntType` differentiate the statement containing "let" and the rest, as defined on line 2.

In the linearization of `MkPosProp1` on lines 4–9 below, we build a table containing two string values for a proposition, one for the 'let' statement (`LetStmnt`) and the other for the rest of statements. We simply do not want to repeat ourselves on line 5–6, and therefore, we use variable `obj`. It is defined on line 8 and it simply combines quantity, scope, list of properties and type. Furthermore, we make the number agreement between subject and type using selection operator (`!`) as (`tp.s!sbj.n`).

In line 6, we use wild card (_), that matches anything (although there remains only one case. i.e. `RestStmnt`). In this table field, we combine the string values of subject, an `oper` function `be` and object `obj`. For the function `be` (which is defined on line 10–11, using another function defined on page 123), we select its string value and make it in agreement with the number of the subject using (`be.s!sbj.n`).

```
1  lincat Proposition = {s : StmntType => Str} ;
2  param StmntType = LetStmnt | RestStmnt ;
3
4  lin MkPosProp1 sbj qnt ps tp = {s = table {
5                LetStmnt => sbj.s ++ be.inf     ++ obj ;
6                _        => sbj.s ++ be.s!sbj.n ++ obj
7               }
8           where {obj = qnt.s ++ ps.s ++ tp.s!sbj.n}
9       };
10 oper be : {s : Number => Str ; inf : Str } =
11          {s = (mkNumStrRec "is" "are").s ; inf = "be"} ;
```

As stated in §2.5, we do not reject ill-formed statements of the following form in GF. Instead we reject them in the host system MathNat (cf. §8.2.1 on page 184).

[9] As per definition of `Quant` in §6.3.1.4 on page 115, strings: "no", "few" and "all" are not allowed.
[10] An example of 'let' statement would be "let x be an even integer".

*Assume that x and y are three positive numbers.

One point worth emphasizing is: it is indeed possible to apply such constraints in GF, but we prefer not to do so. The reason is already explained in the introduction section of Chapter §5.2 on page 93. That is: we want to return a personalized error messages.

Furthermore, there is a function `Empty` that belongs to the category `Quant` (cf. §6.3.1.4 on page 115). It allows to write valid statements such as:

Let x and y be positive numbers.

But, as shown below, we can also write ill-formed statements in which an indefinite article (a | an) is missing.

*Let x be positive number.

This could be seen as the problem of agreement between `Subj` and `Quant`. (These ill-formed statements have the required agreement between subject and type). A few more examples:

*Assume that it is **three** positive number.
*Assume that they are **a** positive numbers.
*Assume that x and y are **a** positive numbers.

The main problem is: any function of category `Quant` (i.e. `One, Two, ..., Empty`) may appear with any function of category `Subj` (i.e. `It`, `They` and `Exps`). Indeed, we can define some `oper` functions which reject such ill-formed sentences. But it would be rather a bit lengthy, and without proper error messages. Currently, we reject then in the host system MathNat as a semantic check (cf. §8.2.2 on page 185), which in return allows to report a better error message.

In future, another possibility to explore would be to define an extra field in the list `Exps`, which stores its length:

```
lincat Exps = {s : Str ; num : Num} ;
```

This way, the `Quant` is not a separate argument, but the list `Exps` ‘remembers’ its size in the `num` field. However, it is more difficult to have a precise `num` for arbitrary size of lists. Also, it is not clear how we will handle quantities such as ‘no’, ‘some’, etc.

The Role of Quantity and Property Arbitrary in Semantics: Let us name the property `Arbitrary` as `Scope`. The role of quantity `Quant` and `Scope` in the above rule is somewhat limited. Let us use the following examples to explain it.

Example set 1:
Let x and y be two arbitrary positive integers.
Let x and y be two positive integers.
Let x and y be positive integers.

The MathAbs translation of these three sentences is the same, as shown below:

let $x, y \in \mathbb{Z}$ assume positive$(x) \wedge$ positive(y) •

It is translated as a formula in first order logic, as shown below:

$$\forall x, y(\underbrace{\underbrace{\underbrace{\text{in}(x,\mathbb{Z}) \wedge \text{in}(x,\mathbb{Z})} \wedge \text{positive}(x)} \wedge \text{positive}(y)})$$

The keyword 'arbitrary' stresses the fact that x and y are universally quantified. This intention is successfully conveyed even when the word "arbitrary" is not present in the second and third sentence. It is because every variable which is not quantified existentially is in fact, universally quantified (also noted by [Peters & Westerstahl 2006, p. 34]). Now consider the following sentence:

Let x be an integer. .... Assume that x is an arbitrary integer. ....
let $x : \mathbb{Z}$ •... let $x : \mathbb{Z}$ •...

The variable x is universally quantified with 'let statement' first, and then, it is again universally quantified with the keyword 'arbitrary', later in some statement. Note that we introduce a new x in the second statement which is different from the variable x in the first sentence. The CLM does not reject such duplication in the current implementation because MathAbs is not yet proof checked and we let the theorem prover to decide if it is a mistake or a valid step.

Another point worth mentioning regarding the *example set 1* is the use of quantity `Quant` in these sentences. For it, we do not need to add a predicate 'two(x,y)' in the MathAbs translation of the above main example. It is because we take care of the number agreement between the variables and the quantity `Quant` in the host system MathNat (§8.2.1 on page 184), before the MathAbs translation phase. Furthermore, from mathematician's perspective, such statements are used to describe the variable having some properties and their types. These are not at all intended for a comparison between the variables and their quantity. This holds for `Scope` also. It means that we do not need predicates such as: "arbitrary(x)", "arbitrary(y)" (for the first statement in above example set) in MathAbs.

It is also worth mentioning that property 'arbitrary' does not make sense in goal (i.e. 'prove' statements) and in deductions (i.e. 'conclusion' statements). This observation is backed by the analysis of various mathematical texts mentioned in Chapter 3. See §6.3.3.1 for more details.

Returning to our main topic, the grammar rules we have defined so far, only allow to parse positive statements. To parse the negative statements, we have the following two options:

1. We could introduce a parameter `Polarity` with two values: `Positive` and `Negative`. Then we could define `Proposition` as table of table: (`Polarity => StmntType => Str`). But note that the parameter is a low level construct in GF which is only visible in the concrete syntax and therefore, it will be lost in the host system MathNat (as we get only the abstract syntax there).

2. To overcome the above shortcoming, we could define `Polarity` as a category having two functions `Positive` and `Negative`, and add it to the abstract syntax as shown below (after renaming the function `MkPosProp1` to `MkProp1`):

```
cat Pol ;
fun Pos, Neg : Polarity ;
fun MkProp1 : Subject -> Pol -> Quant -> Properties1 -> Typ -> Proposition ;
```

The corresponding concrete syntax would be following:

```
lincat Pol = {s : Str} ;
lin Pos = {s = ""} ;
    Neg = {s = "not" } ;

lin MkProp1 sbj pol qnt ps tp = {s = table {
              LetStmnt => sbj.s ++ pol.s ++ be.inf     ++ obj ;
              _        => sbj.s ++ be.s!sbj.n ++ pol.s ++ obj
             }
          where {obj = qnt.s ++ ps.s ++ tp.s!sbj.n}
       };
```

3. **Alternatively, we can add a new function, let us say it MkNegProp1.** It must be similar to its positive twin (i.e. `MkPosProp1`) at the level of abstract syntax but slightly different in the concrete syntax, as show below:

```
fun MkNegProp1 : Subject -> Quant -> Properties1 -> Typ -> Proposition ;

lin MkNegProp1 sbj qnt ps tp = {s = table {
        LetStmnt => []    // nothing is produced for 'let' when it is negative
        _        => sbj.s ++ be.s!sbj.n ++ "not" ++ obj
               }
        where {obj = qnt.s!sbj.n ++ ps.s ++ tp.s!sbj.n}
      };
```

In our current implementation, we take the third approach with the following motivation. As we'll see in §6.3.2 on page 134, we support the attachment of the optional noun adjunct, only to the positive statements (see examples below). With this approach it become easier to do so. It is because we can simply add the category for noun adjunct to the function defined for positive statements (i.e. `MkPosProp1`) and let the function `MkNegProp1` without it.

But why would we add noun adjunct to the positive statements only? Because the interpretation of negative statements with noun adjuncts become quite difficult and unnatural. The main problem is to define the scope of negation and its effect on noun adjuncts. Here are some example sets with noun adjuncts to demonstrate it (the asterisk * means that we do not support them):

Let x, y and z be even integers **with some common elements**.
*Let x, y and z be not even integers **with some common elements**.

x, y and z are three even integers **with some common elements**.
*x, y and z are not three even integers **with some common elements**.

We can assume that $\sqrt{2}$ and $\frac{a}{b}$ are even where a and b are non-zero integers **with no common factor**.
*We can assume that $\sqrt{2}$ and $\frac{a}{b}$ are not even where a and b are non-zero integers **with no common factor**.

Interpretation of Negation for Simple Propositions To show how we interpret the negative propositions, we use the following superficial example and its MathAbs:

Assume that x and y are two arbitrary positive even integers.
let $x, y : \mathbb{Z}$ assume $(\text{positive}(x) \wedge \text{positive}(y) \wedge \text{even}(x) \wedge \text{even}(y))$

Now consider its negation:

Assume that x and y are not two arbitrary positive even integers.

With respect to the scope of negation, there are many interpretations possible. However, as we discuss on page 126, the role of quantity `Quant` and `Scope` is somewhat limited. Therefore, we exclude both from the scope of negation, and it is only applied on the properties and type. We translate this statement in MathAbs as shown below:

If x, y are already declared before the above example statement:
assume $\neg((x, y : \mathbb{Z}) \wedge \text{positive}(x) \wedge \text{positive}(y) \wedge \text{even}(x) \wedge \text{even}(y))$

Otherwise:
let x, y : `NoType` assume $\neg((x, y : \mathbb{Z}) \wedge \text{positive}(x) \wedge \text{positive}(y) \wedge \text{even}(x) \wedge \text{even}(y))$

Let us consider the same example as goal:

Prove that x and y are two arbitrary positive even integers.
let x, y : `NoType` show $(x, y : \mathbb{Z} \wedge \text{positive}(x) \wedge \text{positive}(y) \wedge \text{even}(x) \wedge \text{even}(y))$

Now its negation and MathAbs (which is straightforward):

Prove that x and y are not two arbitrary positive even integers.
let x, y : `NoType` show $\neg(x, y : \mathbb{Z} \wedge \text{positive}(x) \wedge \text{positive}(y) \wedge \text{even}(x) \wedge \text{even}(y))$

Distributive and Collective Readings: One problem still remains. How are we going to differentiate between distributive and collective readings? We again have two options:

First option: We do not make any distinction between collective and distributive readings in the syntax. It should be then taken cared in the host system MathNat as a semantic check §8.2.3 on page 185.

We could effectively make this distinction in GF, but we prefer doing it in the host system to be able to report a personalized error message in a situation when a singular subject is given to the collective property (as shown in the following example). Of course, the error reporting system of GF produces general messages. Whereas, we would like to have rather personalized error messages at every occasion.

Collective properties being an n-predicate cannot be applied on a single variable (or pronoun ‘it’), and therefore, we reject such sentences in the host system MathNat. Two examples of such statement are:

*Let x be equal.
*Assume that it is equal.

However, the problem with this approach is that we have to tag every property being distributive or collective manually, in the MathNat code (as shown in §8.2.3 on page 185). For instance, the property 'equal' is hard coded as collective, but 'even' as distributive. **We currently support this solution in the implementation.**

Second option: We could have followed a convention of considering a property to be collective by default. In case of a distributive property, a user must mention it with the keyword 'each' as shown below in the example:

Let x, y and z**, each** be positive.

Here is the shortcoming: it forces user to write "each" every time she encounters a distributive property.

To implement it in the above rules (cf. `MkPosProp1` and `MkNegProp1`), we would need a category, let us say it `Reading`, and two functions defined on it as shown below:

```
cat Reading ;
fun Distributive : Reading ;
    Collective   : Reading ;
```

The linearization of the above rules in the concrete syntax would be as shown bellow. In it, the third line means that the linearization of function `Collective` is empty.

```
lincat Reading = {s : Str};
lin Distributive = {s = [", each"]} ;
    Collective   = {s = ""} ;
```

Now both functions (`MkPosProp1` and `MkNegProp1`) would take six categories as parameters. For instance consider the abstract syntax rule for function `MkPosProp1` given bellow:

```
fun MkPosProp1 : Subject -> Reading -> Quant -> Properties1 -> Typ -> Proposition ;
```

And the linearization in the concrete syntax:

```
lin MkPosProp1 sbj reading qnt ps tp = {s = table {
         LetStmnt => sbj.s ++ reading.s ++ be.inf     ++ obj ;
         _        => sbj.s ++ reading.s ++ be.s!sbj.n ++ obj
              }
         where {obj = qnt.s!sbj.n ++ ps.s ++ tp.s!sbj.n}
      };
```

6.3.2.2 The Second Rule for Propositions: Subject and Properties

In contrast to the first rule for propositions, we would like to form propositions in which we define properties on subjects without giving their type. But, `Quant` is closely associated with `Typ` linguistically. So if we omit `Typ` we must omit `Quant` also. Therefore, we form the second rule with `Subject` and `Properties2` (and there is no easy way to merge it with the first rule for the purpose of good coding practices, other than defining a new function). For instance,

"x is positive and even" or "x be positive and even",
"x and y are even or odd" or "x and y be even or odd",
"x and y are coprime" or "x and y be coprime",
"they are not equal" or "*they be not equal", etc.
(*We are aware that it is not common to find such a sentence in textbooks or published material but we do not reject for the sake of completing this construction. It should not be a problem because CLM is a controlled language.)

We define two functions: `MkPosProp2` for positive propositions and `MkNegProp2` for negative, as shown in the abstract syntax below:

```
1 fun MkPosProp2 : Subject -> Properties2 -> Proposition ;
2     MkNegProp2 : Subject -> Properties2 -> Proposition ;
```

Its concrete syntax is also straightforward:

```
1  lin MkPosProp2 subj ps2 = {s = table {
2          LetStmnt => subj.s ++ be.inf      ++ ps2.s ;
3          _        => subj.s ++ be.s!subj.n ++ ps2.s
4               }
5       } ;
6
7      MkNegProp2 subj ps2 = {s = table {
8          LetStmnt => []  // nothing is produced for 'let' when it is negative
9          _        => subj.s ++ be.s!subj.n ++ "not" ++ ps2.s
10              }
11      } ;
```

Interpretation of Negation The negation is applied on all properties as shown in the MathAbs of this example[11]:

We conclude that x and y are not positive, even and coprime.
deduce $\neg(\text{positive}(x) \wedge \text{positive}(y) \wedge \text{even}(x) \wedge \text{even}(y) \wedge \text{coprime}(x, y)$

6.3.2.3 The Third Rule for Propositions: Subject and Properties for 'either'

It is very similar to the previous rule (on page 131). It is formed by the `Subject` and `EProperties`. Recall that `EProperties` cannot have less than two properties, Also recall that it is only defined for the positive statements (cf. page 121 for reasons). For instance,

[11] We assume that x and y are already declared in the text before this statement.

"x and y are either even, odd or positive" or
"x and y be either even, odd or positive", etc.

We define a function: `MkPosProp3` for positive propositions as shown in the abstract syntax below:

```
fun MkPosProp3 : Subject -> EProperties -> Proposition ;
```

Its corresponding concrete syntax is shown below:

```
1 lin MkPosProp3 subj ps2 = {s = table {
2         LetStmnt => subj.s ++ be.inf      ++ ps2.s ;
3         _        => subj.s ++ be.s!subj.n ++ ps2.s
4              }
5       } ;
```

6.3.2.4 The Fourth Rule for Propositions: Subject, Relational Function and Expression

We form the fourth rule with `Subject`, `Relation` and `Exp`. For instance,

"x is a divisor of y" or "x be a divisor of y",
"x is not a multiple of y" or "x be not a multiple of y",
"x, y and z are elements of $x * y * z$" or "x, y and z be elements of $x * y * z$",
"x is a square of $\sqrt{x}$" or "x be a square of $\sqrt{x}$", etc.

Of course the use of `Quant` is also possible in this rule producing the following kind of sentences (`Quant` is bold-faced), but we simply not use it without any specific reason.

x, y and z are **three** factors of $x * y * z$

We define two functions: `MkPosProp4` for positive propositions and `MkNegProp4` for negative, as shown in the abstract syntax below:

```
1 fun MkPosProp4 : Subject -> Relation -> Exp -> Proposition ;
2     MkNegProp4 : Subject -> Relation -> Exp -> Proposition ;
```

Both rules are quite similar. Therefore, we only give the concrete syntax for `MkPosProp4` below.

```
fun MkPosProp4 subj rel exp = {s = table {
       LetStmnt  => subj.s ++ be.inf      ++ rest ;
       RestStmnt => subj.s ++ be.s!subj.n ++ rest
      } where {rest = rel.s!subj.n ++ "of" ++ exp.s}
    } ;
```

6.3.2.5 The Fifth Rule for Propositions: Subject, Relational Function and Expression

We cannot say the following statements with the above rule:

"x divides y",
"x does not multiply y",
"x, y and z divide $x * y * z$", etc.

Therefore, we form the fifth rule by `Subject`, `Rel2` and `Exp` producing the above examples. Note that these statements does not make sense if we add `Rel1` and `Rel3` along with `Rel2`.

We define two functions: `MkPosProp5` for positive propositions and `MkNegProp5` for negative, as shown in the abstract syntax below:

```
fun MkPosProp5 : Subject -> Rel2 -> Exp -> Proposition ;
    MkNegProp5 : Subject -> Rel2 -> Exp -> Proposition ;
```

In these rules, we need the values of third person singular (for instance, 'divides') and plural (for instance, 'divide') for `Rel2`. But if we recall the concrete syntax of `Rel2` from §6.3.1.8 on page 123, it is an `oper` record `rel12`, containing a table `s` (having two string values: one for singular and the other for plural) and a participle `part`, as shown below.

```
oper rel12 : Type = rel ** {part : Str} ;
(Recall that it means: oper rel12 : Type = {s : Number => Str ; part : Str})
```

For instance, we had implemented the function `Divisor`, in such a way that it produces only three values: "divisor", "divisors" and "dividing", with the following linearization:

```
lin Divisor = mkRel12 "divisor" "divisors" "dividing" ;
```

But we need two more inflected forms: the third person singular "divides" and third person plural "divide". Therefore, in the linearization of the category `Rel2`, we extend `rel12` with another table field named as `p3` for the third person, as shown below:

```
lincat Rel2 = rel12 ** {p3 : Number => Str} ;
```

Then the linearization of the function `Divisor` becomes (of course we could have defined an `oper` function for it):

```
lin Divisor = {
     s    = table {Sg => "divisor" ; Pl => "divisors" } ;
     part = "dividing" ;
     p3   = table {Sg => "divides" ; Pl => "divide" }
    } ;
```

In the concrete syntax of MkPosProp5 and MkNegProp5, we access these values with (rel2.p3!subj.n) as shown below. We also use the third person singular and third person plural values of verb 'do' (defined as oper do on lines 10–14 below), in the linearization of function MkNegProp5. Further, the underscore (_) on line 2 and on line 7 is a wild card keeping the same values for LetStmnt and RestStmnt.

```
1 lin MkPosProp5 subj rel2 exp = { s = table {
2        _  =>  subj.s ++ rel2.p3!subj.n ++ exp.s
3       }
4      } ;
5     MkNegProp5 subj rel2 exp = { s = table {
6        _  =>  subj.s ++ do.p3!subj.n ++ "not" ++ rel2.p3!subj.n ++ exp.s
7       }
8      } ;
9 oper do : {p3 : Number => Str} = {p3 = table {
10              Sg => "does" ;
11              Pl => "do"
12             }
13           } ;
```

Similar to the oper function do, we need to define functions for other verbs as well (for instance, "to exist" used in §6.3.3.1 on 136). Therefore, we define the following general function to save the values of third person singular and third person plural.

```
1  oper mkP3 : Str -> Str -> {p3 : Number => Str} = \p3sg, p3pl ->
2               {p3 = table {
3                   Sg => p3sg ;
4                   Pl => p3pl
5                  }
6               };
```

Now we can redefine the verb "do" as an oper function and can reuse this function for other verbs as well, as shown below:

```
1  oper do    : {p3 : Number => Str} = mkP3 "does"   "do"    ;
2       exist : {p3 : Number => Str} = mkP3 "exists" "exist" ;
3       ...
```

6.3.2.6 Optional Noun Adjuncts

With the above **positive simple statements**, an optional noun adjunct AdjunctWith may appear. To be more precise, **we add noun adjunct to the first three rules only (cf. §6.3.2.1 on page 125, §6.3.2.2 on page 131 and §6.3.2.3 on page 131)**, because it does not make sense with the fourth and fifth rule. As already stated before, for these three rules we add noun adjunct to only positive propositions (i.e. MkPosProp1, MkPosProp2 and MkPosProp3).

For the moment, we construct noun adjuncts with the following three rules:

1. It is formed by Quant1 and Relation, resulting adjuncts such as "(with | having) (a | one | two | some | ...) common element(s)".

An example of a positive statement with subordinate would be: “a and b are non-zero integers with no common factor”.

In the abstract syntax, we define a category `AdjunctWith` (which is simply a string) and function `MkCmnRelPWith` as shown below:

```
cat AdjunctWith ;
fun MkCmnRelAWith : Quant1 -> Relation -> AdjunctWith ;
```

And the concrete syntax:

```
lincat AdjunctWith = {s : Str} ;
lin MkCmnRelAWith q r = {s =
     variants{"with"; "having"} ++ q.s!Pl ++ "common" ++ r.s!q.n
    } ;
```

Of course, this rule only makes sense in the propositions which have at least two expressions or a plural pronoun as a subject. It is because it contains the word ‘common’, as shown in the code above. See the end of this subsection for more details.

2. Another rule is formed by `Eq`, forming adjuncts such as “(with | having) $x > 0$”. We define the abstract and concrete syntax as shown below:

```
fun MkEqAWith : Eq -> AdjunctWith ;
lin MkEqAWith eq = {s = variants{"with"; "having"} ++ eq.s} ;
```

3. Finally to make the noun adjunct optional, we define `EmptyAdj` function as shown below:

```
fun EmptyAdj : AdjunctWith ;
lin EmptyAdj = {s = "" };
```

Adding Noun Adjuncts to the Propositions

How `AdjunctWith` is added to the proposition? We demonstrate it by redefining the abstract syntax of the first rule for propositions (§6.3.2.1 on page 125) below:

```
fun MkPosProp1 : Subject -> Quant -> Properties1 -> Typ -> AdjunctWith -> Proposition ;
```

Its concrete syntax as shown below, remains the same except that we add a variable `adjwth` for `AdjunctWith`. In line 5, we access its string value from record with (`.s`) and concatenate it with the rest of the string:

```
1  lin MkPosProp1 sbj q scp ps tp adjwth = {s = table {
2          LetStmnt => sbj.s ++ be.inf     ++ obj ;
3          _        => sbj.s ++ be.s!sbj.n ++ obj
4         }
5      where {obj = q.s!sbj.n ++ scp.s ++ ps.s ++ tp.s!sbj.n ++ adjwth.s}
6    };
```

Some example statements are following:

a and b are non-zero integers with no common factor,
a and b are non-zero integers with $a \neq b$, etc.

Also note that such a noun adjunct which contains "common" (i.e. the function in bullet 1 of §6.3.2.6 on page 135). It only makes sense to the propositions having at least two elements in the expression list `Exps` as a subject. Similarly, it only makes sense for pronoun `They`. It is demonstrated by the following ill-formed statements:

*a is a non-zero integer with no **common** factor,
*It is a non-zero integer with three **common** elements, etc.

We reject such statements in the host system MathNat (cf. §8.2.5 on page 188).

Again, we add noun adjunct to the first three rules only (cf. §6.3.2.1 on page 125, §6.3.2.2 on page 131 and §6.3.2.3 on page 131), because adding it to the fourth and fifth rules results into ill-formed sentences.

6.3.3 Existential Statements

We now proceed to build the positive and negative existential statements (`PropExist`). As mentioned in §3.3.3.6 on page 46 and in §4.6 on page 72, the variables in these statements are quantified on the information whether they appear as assumption, deduction or goal (to see how we make these proof statements, we specifically refer to §7.2.1). Nevertheless, we name them existential statements to be able to distinctly see them in the host system MathNat and apply the appropriate quantification.

6.3.3.1 The First Rule for Existential Statements

It is formed by `Quant`, `Properties1`, `Typ`, `Exps` and `Equations`. For instance:

"there (exists | is) a positive number n such that $n * a > b$",
"there (do not exist | are no) even integers x, y and z such that $x * a > b$, $y * a > b$ (and | or) $z * a > b$", etc.

The rule seems similar to the rule 6.3.2.1 given on page 125. For instance, consider the following statement formed by this rule:

Assume that x is a positive even integer.

With this rule, we can rephrase it to its logically equivalent statement:

Assume that there is an integer x such that $x > 0$ and even(x).

In the abstract syntax, we define category `PropExist` and two functions `MkPosExist1` (for positive existential statements) and `MkNegExist1` (for negative existential statements), as shown below.

```
cat PropExist ;
fun MkPosExist1 : Quant -> Properties1 -> Typ -> Exps -> Equations -> PropExist ;
```

We omit the implementation of function `MkNegExist1` because it is very similar to the implementation of function `MkPosExist1`.

The linearization of category `PropExist` is simply a string record. However the linearization of function `MkPosExist1` is straightforward but a bit lengthy. For instance, on line 4, we make number agreement between verbs `exist` (defined on page 134 as an `oper` function) and `be` (defined on page 125 as an `oper` function) with quantity (`exist.p3!q.n` and `be.s!q.n`). The use of `variants` produces two string values: "there exist(s)" and "there (is|are)". We further combine them with quantity (`q`), properties (`props`), type (`t`) and so on. For the type we make its number to be in agreement with quantity.

```
1  lincat PropExist ;
2  fun MkPosExist1 q props t exps exp = {s =
3       "there" ++ variants {
4                            exist.p3!q.n ;
5                            be.s!q.n
6                          } ++
7       q.s ++ props.s ++ t.s!q.n ++ exps.s ++ optComma ++
8       ["such that"] ++ exp.s
9     } ;
10  oper optComma : Str = variants{","; ""} ;
```

Note the `oper` function `optComma` which returns an optional comma. We will use it at many places in this chapter and in the next chapter.

A Note on Its Interpretation:

Consider the following example:

There are x and y such that $xa + yp = 1$ **or** $xab + ypb = b$.

How should we interpret it? For instance, both interpretations shown below seems fine:

1. There are x and y such that $\underbrace{xa + yp = 1 \text{ or } xab + ypb = b}$.
2. $\underbrace{\text{There are } x \text{ and } y \text{ such that } xa + yp = 1}$ **or** $xab + ypb = b$.

So as a convention, we consider existential statement having more precedence than conjunctions and disjunctions appearing inside. Therefore, we interpret it as the first one:

There are x and y such that $\underbrace{xa + yp = 1 \text{ or } xab + ypb = b}$.

However, if we would like to express the second interpretation, we should add a comma before the disjunction "or", as shown below (See §7.5 on page 176 to for a general account of these conventions).

$\underbrace{\text{There are } x \text{ and } y \text{ such that } xa + yp = 1}$ **, or** $xab + ypb = b$.

A Note on Property "arbitrary" in Existential Statements:

The property "arbitrary" is mostly illegal with existential statements. Consider the following existential statement with the property "arbitrary":

> *There exists an **arbitrary** integer x such that $x > 0$.

It seems strange, or even contradictory due to the use of "arbitrary" with existentially quantified statement. Because, "arbitrary" can only fundamentally correspond to the introduction of a fresh variable, which basically arise in two situations (cf. §4.6 on page 72):

1. When we want to prove a proposition with a universal quantifier (i.e. show $\forall x P(x)$ in MathAbs for a proposition $P(x)$).

2. When we want to use the fact that a proposition with an existential quantifier is true (i.e. assume $\exists x P(x)$ in MathAbs for a proposition $P(x)$).

But let us see some other examples before drawing any quick conclusions. Consider the two statements below. They are made from two different rules yet seem to have the same meaning. The first example uses the above statement.

> Example set 1:
> Assume that there **exists an arbitrary** integer x such that $x > 0$.
> Assume that x is an **arbitrary** positive integer.

In contrast, consider the same statements but slightly modified. Are they similar in meaning? Are they even correct logically?

> Example set 2:
> *We conclude that there **exists an arbitrary** integer x such that $x > 0$.
> *There exists an **arbitrary** integer x such that $x > 0$.
> *We conclude that x is an arbitrary positive integer.
> *Prove that x is an arbitrary positive integer.

If we try to look for such statements where property "arbitrary" appears in existential statements in the mathematical books, they are usually not found. In a similar way, we do not find goals, deduction, conclusion containing the property "arbitrary" for simple propositions (cf. §6.3.2 on page 124). i.e. statements similar to the last one in the *example set 2*.

However, even if they do appear together (very rare), we find them highly difficult to interpret. Therefore, we do not allow the property "arbitrary" when it appears in the statements similar to the examples given in *example set 2* (i.e. when they occur as goals or deductions in existential and simple propositions), and reject them in the host system MathNat (cf. 8.2.5 on page 188). In contrast, we allow the property "arbitrary" when it appears in the statements similar to examples given in *example set 1*.

Interpretation of Negation in Existential Statements:

To show how we interpret the negative existential statements, we use the following false example:

Assume that there exists two integers x and y such that $x > y$ and $x \neq y$.

Its MathAbs is:

let $x, y : \mathbb{Z}$ assume $(x = y \wedge x \neq y)$ •

Now consider its negation and MathAbs, followed by some explanation:

Assume that there does not exist two integers x and y such that $x = y$ and $x \neq y$.
Logical Formula: $\neg(\exists\, x, y : \mathbb{Z}\ (x = y \wedge x \neq y))$
By De Morgan's law: $\forall\, x, y : \mathbb{Z}\ \neg(x = y \wedge x \neq y)$
MathAbs: assume $\forall\, x, y : \mathbb{Z}\ \neg(x = y \wedge x \neq y)$ •

Instead of using the actual logical formula in MathAbs, we utilize its factorized version, in which negation is removed from the quantifier. It is because, we need the scope of these variables to support linguistic features such as anaphora of variables and references (cf. §2.5.2 on page 19).

Also, as we discuss on page 126, the role of category quantity `Quant` and category `Scope` is limited. Therefore, we exclude both categories from the scope of negation as we have already done for simpler propositions (see on page 129).

Now consider the same false example and its MathAbs as goal:

Prove that there exists two integers x and y such that $x = y$ and $x \neq y$.
show $\exists\, x, y : \mathbb{Z}\ (x = y \wedge x \neq y)$

And now its negation:

Prove that there are no integers x and y such that $x = y$ and $x \neq y$.

Note that it is equivalent to $\neg(\exists\, x, y : \mathbb{Z}\ (x = y \wedge x \neq y))$ which by factorizing becomes: $\forall\, x, y : \mathbb{Z}\ \neg(x = y \wedge x \neq y)$. It becomes following in MathAbs:

let $x, y : \mathbb{Z}$ show $\neg(x = y \wedge x \neq y)$ •

Again, the reason for this transformation is to be able to extract the correct quantification. Also, we might need these variables in the proof, if they are not declared again. However, if they are declared again in the proof, then that declaration will simply appear on top of this declaration and be used in the rest of the proof.

6.3.3.2 The Second Rule for Existential Statements

We would like to define a rule similar to the first one but without `Typ`. If `Typ` does not appear in the statement, `Quant` also does not make sense as demonstrated by the following statement:

*There exist three x, y and z such that $x * a > b$, $y * a > b$ and $z * a > b$.
(It should be "three `Typ` x, y and z", see the next rule)

So we cannot re-define it inside rule 1. Therefore, we form the second rule by `Properties1`, `Exps` and `Equations`. For instance,

"there (exists | is) positive even n such that $n * a > b$",
"there (does not exist | is no) n such that $na > b$",
"there (exist | are) positive x, y and z such that $x * a > b$ and $y * a > b$", etc.

We define two functions `MkPosExist2` (for positive existential statements) and `MkNegExist2` (for negative existential statements). In the abstract and concrete syntax shown below, we give below only `MkNegExist2` because both are quite similar.

```
fun MkNegExist2 : Properties1 -> Exps -> Equations -> PropExist ;

lin MkNegExist2 ps exps eqs =
   {s = "there" ++
        variants{ do.s!exps.n ++ ["not exist"] ;
                  be.s!exps.n ++ "no"
                } ++
        ps.s ++ exps.s ++ optComma ++ ["such that"] ++ eqs.s
   } ;
```

6.3.4 Relational Statements

6.3.4.1 The First Rule for Relational Statements

A rather limited role for its coverage, we give the following statements, which are formed by `Subject`, `QuantRel` and `Relation`. For instance,

"x, y and z have a common factor",
"they have some common multiples",
"x, y and z have no common divisor", etc.

In the abstract and concrete syntax shown below, we define category `PropRel` and a function `MkPropRel1`. Note that `Quant1` defined on page 116 contains 'no'. Therefore, we do not need a function for negative statements.

```
1 cat PropRel ;
2 fun MkPropRel1 : Subj -> Quant1 -> Relation -> PropRel ;
```

```
1 lincat PropRel = { s : Str } ;
2 lin MkRelProp1 subj qnt rel = {
3      s = subj.s ++ has.s!subj.n ++ qnt.s ++ "common" ++ rel.s!qnt.n
4     };
5 oper has : {s : Number => Str} = mkNumStrRec "has" "have" ;
```

Note that on line 3 above, we select the value of relational function in accordance with the quantification (`rel.s!qnt.n`).

But can we modify the above rule in a way that an optional `Exp` could be added at the end forming the following example statements:

"n and m have a common divisor d",
"they have no common factor 2", etc.

Yes we can. But let us upgrade it for the following statements as well:

"n and m have two common divisors d and e",
"n and m have two common divisors",
"n and m have common divisors d and e",
"they have no common factors 2 and 3", etc.

Now we should not. We explain the reason in subsequent paragraphs. Now we need `Subject`, `Quant1`, `Relation` and a list of expressions (which allows an empty list). The `Quant1` must have an agreement with `Relation`. (So that we produce "two common divisors", but not "two common divisor", etc.)

But the following three examples, reveal that the rule becomes very complex if we insist on using a list for expressions which allows empty list:

n and m have two common divisors.
n and m have a common divisor d.
n and m have two common divisors d and e.

Such a list must have three possibilities, as shown in the above example respectively:

1. Empty (the names of these common divisors are not given)
2. One element (singular, d)
3. More than one elements (plural, d, e)

The category `Quant1` can easily make agreement with the first case or the last two cases; but not both at the same time. Therefore, instead of taking the pain of defining a rule which becomes complex not only at GF level but also in the host system MathNat for semantics, we form the following two rules:

1. Formed by `Subject`, `Quant1` and `Relation`. (Already covered with the function `MkPropRel1`.)
2. Formed by `Subject`, `Quant1`, `Relation` and `Exps`. (Defined in the next section.)

6.3.4.2 The Second Rule for Relational Statements

For the second rule in the above list, we combine `Subject`, `Quant1`, `Relation` and `Exps` as shown below:

```
1  fun MkPropRel2 : Subj -> Quant1 -> Relation -> Exps -> PropRel ;
2
3  lin MkPropRel2 subj qnt rel exps = { s = subj.s ++ has.s!subj.n ++
4           qnt.s ++ "common" ++ rel.s!qnt.n ++ exps.s } ;
```

Note that on lines 3–4 above, we select the value of verb `has` in accordance with subject (`has.s!subj.n`) and the value of relational function (`Relation`) in accordance with quantity (`rel.s!qnt.n`). However the following examples demonstrate that it is still a partial agreement.

*n and m have **three** common divisors d and e.
The quantity (three) must agree with the number of expressions (d, e).

*n has **three** common **divisor** d and e.
The word "common" only applies to plural subjects, but n is singular.
The subject must agree with quantity (three), as well as with the number of expressions (d, e).

In summary, two conditions must be fulfilled: (1) the agreement between `Quant1` and `Exps` and (2) `Exps` must have more than one elements. These conditions are applied in the host system MathNat as semantic check (cf. §8.2.6 on page 188).

Note that the list `Exps` does not allow to have its elements (i.e. `Exp`) separated by disjunction. Therefore, we do not have to consider the possibility of following ill-formed statements:

*n and m have two common divisors d or e.

6.3.4.3 Two Generalized Rules for Relational Statements

The above two rules hard-code the word “common” for some statements that appears in textual proofs. We define the same functions now without it. For instance,

“n and m have a divisor d”,
“they have no factor 2”, etc.

or

“n and m have a divisor”,
“they have three factors”, etc.

Again note that we cannot use `variants` in the concrete syntax for this variation. Because we will unable to see this information in the host system. Therefore, we form two rules:

1. Formed by `Subject`, `Quant1` and `Relation`.
2. Formed by `Subject`, `Quant1`, `Relation` and `Exps`.

We define two functions `MkGenPropRel1` and `MkGenPropRel2`. We only define `MkGenPropRel1` below.

```
fun MkGenPropRel1 : Subj -> Quant1 -> Relation -> PropRel ;
lin MkGenPropRel1 subj qnt rel = {
      s = subj.s ++ has.s!subj.n ++ qnt.s ++ rel.s!qnt.n
     } ;
```

However note that, we need to define somewhat similar semantic check for the function `MkGenPropRel2` as we define for function `MkPropRel2` (cf. §6.3.4.2 on page 141) in §8.2.6 on page 188. We skip the implementation of its semantic check as well.

6.3.5 Equation with a Reference

Formed by `Equation` and `Reference`. For instance,

“$x^2 + y^2 = (2 * a + 1)^2 + (2 * b + 1)^2$ – (1)”,
“$x^2 + y^2 = (2 * a + 1)^2 + (2 * b + 1)^2$ – i”,
“$x^2 = (2 * a + 1)^2$”, etc.

We define a category `EqWithRef` and a function `MkEqWithRef`. This function actually combines equation and an optional reference, as shown below:

```
cat EqWithRef ;
fun MkEqWithRef : Eq -> Ref -> EqWithRef ;

lincat EqWithRef = {s : Str} ;
fun MkEqWithRef eq ref = {s = eq.s ++ ref.s} ;
```

But we have not yet defined `Ref`. It is simply an optional `String` enclosed with optional brackets, as shown in the following abstract and concrete syntax. Note that the word optional is translated as a function `NoRef`.

```
cat Ref ;
fun MkRef : String -> Ref ;
    NoRef : Ref ;

lincat Ref = {s : Str} ;
lin MkRef ref = {s = variants { ref.s ; "(" ++ ref.s ++ ")"} } ;
    NoRef     = {s = ""};
```

6.3.6 Statements

We now combine propositions, equations (with references), relational and existential statements into one category labeled as `Statement`. The abstract and concrete syntax is given below followed by some explanation:

```
cat Statement ;
fun MkPropStmnt  : Proposition -> Statement ;
    MkExistStmnt : PropExist   -> Statement ;
    MkRelStmnt   : PropRel     -> Statement ;
    MkEqRefStmnt : EqWithRef   -> Statement ;
```

```
1 lincat Statement = { s : Str } ;
2 lin MkPropStmnt   prop   = { s = prop.s!RestStmnt } ;
3     MkExistStmnt  pexst  = pexst ;
4     MkRelStmnt    prel   = prel  ;
5     MkEqRefStmnt  eqRef  = eqRef ;
```

As shown above, `Statement` is simply a string record. Other than `Proposition`, the remaining categories are also string records and therefore, directly up-cast to the `Statement`. We choose `Statement` to be used as a general category in the macro level grammar (cf. §7). It means that we'll use its list (see §6.3.6.1) for most of the proof statements such as assumptions ("we suppose that . . . ", etc), deductions ("we conclude that . . . ", etc), goals ("prove that . . . ", etc), etc. It also explains why we select the string value associated with `RestStmnt` and ignore the field `LetStmnt` (which is used only with 'let' statements given in the next paragraph) from the category `Proposition` on line 2.

For the 'let' statements, we define a slightly different category labeled as `LetStatement`. It is defined as shown below:

```
cat LetStatement ;
fun MkPropLetStmnt  : Proposition -> LetStatement ;
    MkEqRefLetStmnt : EqRef       -> LetStatement ;

lincat LetStatement = { s : Str } ;
lin MkPropLetStmnt  prop  = { s = prop.s!LetStmnt } ;
    MkEqRefLetStmnt eqref = eqref ;
```

Note that the categories `PropExist` and `PropRel` would produce linguistically ill-formed sentences if combined with 'let'. Therefore, they are not available in `LetStatement`. Here are two examples of this category when it is used in a proof or theorem statements:

"let A and B be two sets",
"let $(x + y)^2 = x^2 + y^2 + 2 * x * y$", etc.

The List of Statements

6.3.6.1 Statements

The category `Statements` is a non-empty list of `Statement`. It supports the following pattern:

$\texttt{Statement}_1[, \ldots, \texttt{Statement}_{n-1}$ (, and |, or) $\texttt{Statement}_n]$

As we have seen in §2.4, we resolve the coordination ambiguity in above statements by interpreting them as following:

$$\underbrace{\texttt{Statement}_1\ \underbrace{[, \ldots, \underbrace{\texttt{Statement}_{n-1}\ \underbrace{(\text{, and}|\text{, or})\ \texttt{Statement}_n]}}}}$$

Note the use of comma in "(, and |, or)" to resolve the ambiguity with the conjunction and disjunction in different kinds of statements that can appear inside $\texttt{Statement}_i$. An elaborative example is the following in which we repeat the same statement three times:

There are x and y such that $xa + yp = 1$ or $xab + ypb = b$ **,**
there exists x_1 and y_1 such that $x_1a + y_1p = 1$ or $x_1ab + y_1pb = b$ **, or**
there exists x_2 and y_2 such that $x_2a + y_2p = 1$ or $x_2ab + y_2pb = b$.

As already stated, this list is given to the various blocks, for instance, in proof block to form proof statement (assumption, deduction, goal, proof by cases, etc), as shown below for the above example (after some modification):

We conclude that there are x and y such that $xa + yp = 1$ or $xab + ypb = b$ **,**
x_1 and y_1 are positive even integers **, and** x_1 divides y_1.

or

Suppose that x is even**,** y is odd **, and** z is rational.

In terms of implementation, it is similar to the pattern of the category `Exps` and `Properties2` given on pages 111 and 119 respectively, and therefore, we define it in a similar way:

```
1  cat Statements ; PStatements ;
2  fun BaseAndPStatements : Statement -> Statement   -> PStatements ;
3      BaseOrPStatements  : Statement -> Statement   -> PStatements ;
4      ConsPStatements    : Statement -> PStatements -> PStatements ;
5
6      BaseStatements : Statement   -> Statements ;
7      ConsStatements : PStatements -> Statements ;
```

We now give its concrete syntax, which seems quite straightforward to follow:

```
1  lincat Statements  = List ;
2         PStatements = {s : Str } ;
3
4  lin BaseAndPStatements = twoElemList [", and"] ;
5      BaseOrPStatements  = twoElemList [", or"]  ;
6      ConsPStatements    = consList    ","       ;
7
8      BaseStatements = oneElemList Sg ;
9      ConsStatements = oneElemList Pl ;
```

The function `BaseStatements` above, covers the case when we have only one element in the list. In contrast, the `ConsStatements` covers the following four patterns:

1. `Statement`$_1$ **, and** `Statement`$_2$ (using function `BaseAndPStatements`).
2. `Statement`$_1$ **, or** `Statement`$_2$ (using function `BaseOrPStatements`).
3. `Statement`$_1$, ..., `Statement`$_{n-1}$ **, and** `Statement`$_n$
   (using functions: `ConsPStatements` and `BaseAndPStatements`).
4. `Statement`$_1$, ..., `Statement`$_{n-1}$ **, or** `Statement`$_n$
   (using functions: `ConsPStatements` and `BaseOrPStatements`).

6.3.6.2 EStatements

The category `EStatements` is a list of `Statement` having at least two elements. It supports the following pattern:

Either `Statement`$_1$, ..., `Statement`$_{n-1}$ or `Statement`$_n$.

It is similar to the pattern of category `EProperties` given on page 121 and therefore we define it in a similar way. For that we make use of category `PStatements` by defining a function `MkEStatements` that takes it as a parameter as shown below in the abstract syntax:

```
cat EStatements ;
fun MkEStatements : PStatements -> EStatements ;
```

In its concrete syntax below, we simply add the clue word ‘either’ in front:

```
lincat EStatements = {s = Str ; n = Number } ;
lin MkEStatements xs = {s = "either" ++ xs.s ; n = Pl } ;
```

(Of course we can easily re-factor it in a way that we share the linearizations with `EProperties` and other similar categories using `oper` functions.)

Either $\texttt{Statement}_1$, or $\texttt{Statement}_2$.
Either $\texttt{Statement}_1$, $\texttt{Statement}_2$, ..., or $\texttt{Statement}_n$.

It fulfills one of our requirement: i.e. the list must have two elements at minimum. We need this requirement because the clue word 'either' does not make sense when the list is empty or has only one element. However, it allows to write statements of the form given below, which we reject with the semantic checks (cf. §8.2).

*Either $\texttt{Statement}_1$, and $\texttt{Statement}_2$.
*Either $\texttt{Statement}_1$, $\texttt{Statement}_2$, ..., and $\texttt{Statement}_n$.

6.3.6.3 LetStatements

Note that in `Statements` and `EStatements`, propositions for 'let' statements cannot occur (defined in §6.3.2 on page 124). For that we define a non-empty list labeled as `LetStatements` in which only propositions defined in §6.3.2 on page 124 are allowed. It will allow us to write, let us say, proof statement of the form:

Let x be a positive integer **, and** y be a negative integer.
Let $x = y$ **, and** $y = z$.
Let $x = y$ **, or** $x = z$.

Because of its similarity with the category `Statements`, we omit its implementation details.

6.3.7 Conditional Statement

The category `IfthenStmnt` is formed by two lists (`Statements`), as shown the following pattern:

if $\texttt{Statement}_1$[, $\texttt{Statement}_2$, ..., $\texttt{Statement}_{n-1}$ (, and | , or) $\texttt{Statement}_n$] then $\texttt{Statement}_i$[, $\texttt{Statement}_{ii}$, ..., $\texttt{Statement}_{k-i}$ (, and | , or) $\texttt{Statement}_k$].

For instance,

"if a and b are even then they have a common factor",
"if m and r have a common divisor d then it divides n",
"if $a = 2 * c$, and $4 * c^2 = 2 * b^2$, and $2 * c^2 = b^2$ then b is even"
and a superficial conditional:
"if a is positive, b is negative, and c is even then x is even, y is odd, and $x * y * z = 10$", etc.

Its abstract and concrete syntax is rather straightforward as shown below:

```
cat PropIfthen ;
fun MkPropIfthen : Statements -> Statements -> PropIfthen ;

lincat PropIfthen = {s : Str} ;
lin MkPropIfthen st1 st2 = {s =
      "if" ++ st1.s ++ optComma ++ "then" ++ st2.s} ;
```

Again, as given in §2.4, we resolve the coordination ambiguity in `IfthenStmnt` by interpreting them as following:

$$\textbf{If } \underbrace{P_1, \underbrace{P_2, \ldots, \underbrace{\underbrace{P_{n-1} (\text{and}|\text{or})\ P_n}}}}\ \textbf{then}\ \underbrace{Q_1, \underbrace{Q_2, \ldots, \underbrace{\underbrace{Q_{n-1} (\text{and}|\text{or})\ Q_n}}}}.$$

6.3.8 Take Statement

The category `TakeStmnt` is formed by the list (`Equations`), as shown the following pattern:

we [can] (choose | take | select) `Equations`.

For instance,

"we can choose $x := 10$",
"we take $a := x + 1$, $b := y + 1$ and $c := z + 1$", etc.

Its implementation is rather straightforward and therefore we skip it.

6.3.9 Justifications

As we see in Chapter 2 and 4, mathematical assertions are sometimes aided with justifications. Therefore, for justifications we define a category `Justification`. It is formed by (`Statement` | `Operation` | `Definition` | `Anaphora` | `DefReference`) (the last four are defined in the subsequent subsections), as shown below:

```
cat Justification ;
fun MkStmntJust  : Statement    -> Justification ;
    MkOperJust   : Operation    -> Justification ;
    MkAnphrJust  : Anaphora     -> Justification ;
    MkDefRefJust : DefReference -> Justification ;
```

In the corresponding concrete syntax shown below, the `Justification` is a record having two values: `s` of type `Str` and `t` of type `JustifType`. In line 6, we define algebraic data type `JustifType` with two constructors: `JstStmnt` for justifications formed from `Statement` and `JstRest` for the rest. We use this information in proof statements, for instance, see §7.2.1.3 on page 160.

```
1 lincat Justification = {s : Str ; t : JustifType} ;
2
3 lin MkStmntJust st   = {s = st.s   ; t = JstStmnt} ;
4     MkOperJust  oper = {s = oper.s ; t = JstRest } ;
5     MkDefJust   df   = {s = df.s   ; t = JstRest } ;
6     ...
7 param JustifType = JstStmnt | JstRest ;
```

6.3.9.1 Operation as Justifications

An operation (`Operation`) which is needed for `Justification` (§6.3.9 on page 147). The coverage of these operations is very limited yet for the moment highly desirable with respect to the textual proof of elementary number theory. It is formed in the following ways:

The First Rule for Operational Justifications is formed by `Rel1` (cf. given on page 123). For instance,

"(factoring | squaring | ...) [at] both sides",
"taking (factor | square | ...) (at | from) both sides".

Its abstract and concrete syntax is as shown below:

```
cat Operation ;
fun MkOper1 : Rel1 -> Operation ;
```

```
1  lincat Operation = { s : Str } ;
2  lin MkOper1 rel1 = { s = variants {
3        "taking" ++ rel1.s!Sg ++ variants{"at"; "from"} ++ ["both sides"] ;
4        rel1.part ++ variants{"";"at"} ++ ["both sides"]
5        }
6     } ;
```

Note that `Rel1` contains a string field for participles (`part : Str`), which we access on line 4 above.

The Second Rule for Operational Justifications is formed by `Rel2` and `Exp`. For instance,

"(multiplying | dividing | ...) both sides by 2",
"multiplying the equation by x", etc.

Its abstract and concrete syntax is as shown below:

```
fun MkOper2 : Rel2 -> Exp -> Operation ;
```

```
1  lin MkOper2 rel2 exp = { s = variants {
2        rel2.part ++ variants{"";"at"} ++ ["both sides by"] ++ exp.s ;
3        rel2.part ++ ["the equation by"] ++ exp.s;
4        }
5     } ;
```

The Third Rule for Operational Justifications is formed by `Rel2`, `EqAnaphor` and `Exp`. For instance,

"multiplying the last equation on both sides by 2",
multiplying the last equation by 2 at both sides",
"dividing our first equation by x", etc.

The category EqAnaphor is not yet defined but from the above examples, we can guess that it is the string: "the (last|first) equation", defined in §6.3.9.2 on page 150.

Returning to our main topic, we define the abstract and concrete syntax this operation as shown below:

```
fun MkOper3 : Rel2 -> EqAnaphor -> Exp -> Operation ;

lin MkOper3 r2 eqa exp = { s = variants{
  r2.part ++ eqa.s ++ variants{"on"; "at"} ++ ["both sides by"] ++ exp.s;
  r2.part ++ eqa.s ++ "by" ++ exp.s;
  r2.part ++ eqa.s ++ "by" ++exp.s ++ variants{"on"; "at"} ++ ["both sides"]
  }} ;
```

The Fourth Rule for Operational Justifications is formed by Rel3. It is similar to the first rule but still we record their difference in the abstract syntax. For instance,

"taking factor from both sides".

It literally produces only the above sentence, because the category Rel3 has only one function Factor for the moment. We define function MkOper4 as shown below:

```
fun MkOper4 : Rel3 -> Operation ;
lin MkOper4 rel3 = { s = "taking" ++ rel3.s!Sg ++ ["from both sides"]};
```

The Fifth Rule for Operational Justifications is formed by Exp and (Equation | Reference).
For instance,

"substituting [the value of] x in [equation] $x = 2 * b + 1$",
"substituting [the value of] x into [equation] (i)", etc.

We define two functions MkOper5 and MkOper6 as shown below:

```
fun MkOper5 : Exp -> Eq        -> Operation ;
    MkOper6 : Exp -> Reference -> Operation ;

lin MkOper5 exp eq = {
      s = "substituting" ++ variants{""; ["the value of"]} ++ exp.s ++
          variants{"in"; "into"} ++ variants{""; "equation"} ++ eq.s
     } ;

    MkOper6 exp ref = {
      s = "substituting" ++ variants{""; ["the value of"]} ++ exp.s ++
          variants{"in"; "into"} ++ variants{""; "equation"} ++ ref.s
     } ;
```

6.3.9.2 Anaphoric Reference

As demonstrated in §3.3.5, both implicit and explicit anaphoric references are quite common in mathematical texts. We define anaphoric reference (Anaphor) for statement, equation and theorem. Typical patterns are following:

1. (the | our) (first | last) (‘statement’ | ‘hypothesis’ | ‘deduction’ | ‘equation’ | ‘theorem’).
2. ‘theorem 24’, ‘theorem’, etc.

The category `Anaphor` combines `EqAnaphor`, `StAnaphor` and `ThmAnaphor` (defined in subsequent subsections), as shown below:

```
cat Anaphor ;
fun MkAnaphor1 : EqAnaphor  -> Anaphor ;
    MkAnaphor2 : StAnaphor  -> Anaphor ;
    MkAnaphor3 : ThmAnaphor -> Anaphor ;

lincat Anaphor = {s : Str} ;
lin MkAnaphor1 eqA  = eqA  ;
    ...
```

Again, note that we need this information in the host system MathNat to solve anaphoric pronouns and references, and therefore, must register it as categories and functions to make it visible. We now define categories `EqAnaphor`, `StAnaphor` and `ThmAnaphor` below.

But before we proceed, we could have defined the functions of these categories directly as the functions of category `Anaphor`. Of course both solutions are technically the same and it is more like a matter of taste.

EqAnaphor The category `EqAnaphor` could be better explained by the following pattern:

“(the | our) (first | last) equation”

It is defined as shown below:

```
cat EqAnaphor ;
fun MkFirstEq, MkLastEq : EqAnaphor ;

lincat EqAnaphor = { s : Str } ;
lin MkFirstEq = { s = variants {
                   ["the first equation"] ;
                   ["our first equation"]
                  }
                } ;
    MkLastEq = ...
```

StAnaphor The category `StAnaphor` could be better explained by the following pattern:

“(the | our) (first | last) (statement | hypothesis | deduction)”

We define it in abstract syntax as shown below, but omit its concrete syntax.

```
cat StAnaphor ;
fun MkFirstSt,      MkLastSt      : StAnaphor ;
    MkFirstHypoth,  MkLastHypoth  : StAnaphor ;
    MkFirstDeductn, MkLastDeductn : StAnaphor ;
```

ThmAnaphor The category `ThmAnaphor` could be better explained by the following pattern:

"theorem 24", "theorem", etc.

We define its abstract and (rather simple) concrete syntax as shown below:

```
1  cat ThmAnaphor ;
2  fun MkThmAnaphor1 : String -> ThmAnaphor ;
3      MkThmAnaphor2 : ThmAnaphor ;
4
5  lincat ThmAnaphor = {s : Str} ;
6  fun MkThmAnaphor1 n = {s = "theorem" ++ n.s} ;
7      MkThmAnaphor2   = {s = "theorem"} ;
```

6.3.9.3 Reference to Definitions

A reference to the definition (`DefReference`) is needed for `Justification` (on page 147) and quite common in mathematical texts. It is formed in the following ways:

The First Rule for Reference to Definition is formed by an optional `PropertyOpt` and `Typ`. For instance,

"the definition of [positive] integer[s]",
"the definition of [even] number[s]",
"the definition of prime(s)", etc.

We first define `PropertyOpt` which has two functions:

```
cat PropertyOpt ;
fun MkPropOpt     : Property -> PropertyOpt ;
    MkEmptPropOpt : PropertyOpt ;

lincat PropertyOpt = { s : Str } ;
lin MkPropOpt p   = p ;
    MkEmptPropOpt = {s = ""} ;
```

And the function `MkDef1` in the following abstract and concrete syntax which is self explanatory.

```
cat DefReference ;
fun MkDef1 : PropertyOpt -> Typ -> DefReference ;

lincat DefReference = { s : Str } ;
lin MkDef1 p t = {s = ["the definition of"] ++ p.s ++ variants{t.s!Sg; t.s!Pl}} ;
```

The Second Rule for Reference to Definition is formed by literals: (`Euclidean_Division` | `Induction_Hypothesis` | ...). These literals may have more than one string values. For instance, the linearization of function `Euclidean_Division` is both 'Euclidean division' and 'the definition of Euclidean division'. We define them in the following way:

```
fun Euclidean_Division, Induction_Hypothesis, ... : DefReference ;

lin Euclidean_Division = {
      s = variants{ ["euclidean division"] ;
                    ["the definition of euclidean division"]
                  }
     } ;
    Induction_Hypothesis = {
      s = variants{ ["induction hypothesis"] ;
                    ["the definition of induction hypothesis"]
                  }
     } ;
```

The List of Justifications

The category `Justifications` is a non-empty list of `Justification`. It supports the following pattern:

- (because | since |...) $\texttt{Justification}_1$[, ..., (because | since |...) $\texttt{Justification}_{n-1}$ and (because|since|...) $\texttt{Justification}_n$]

The category `Justifications` is used in some proof statements (cf. §7.2.1.3 on page 160 and §7.2.1.5 162). A few examples are following (the part in brackets {...} does not belong to `Justifications`):

"**by** the definition of positive numbers **and by** euclidean division, {we (conclude | assume) that ...}",
"**by** the last statement **and by** substituting x in $y = x + 10$, {...}",
"**by** the first statement **and because** it is even", {...}
"**by** the last statement**, by** substituting x in $y = z{*}x{+}10$ **and because** z is positive, {...}", etc.

The key phrases "(because | since |...)" belongs to an `oper` function `JustifPhr`, defined as shown below and followed by explanation:

```
1  oper JustifPhr : JustifType -> Str = \t -> case t of {
2         JstStmnt => variants {["by the fact that"]; "since"; "because"}
3         JstRest  => "by" ;
4       } ;
```

In line 1, we use an algebraic data type `JustifType` in the operation `JustifPhr`. It is defined in §6.3.9 on page 147. `JustifType` has two constructors: `JstStmnt` for justifications formed by `Statement` and `JstRest` for the rest. The operation `JustifPhr` takes `JustifType` as parameter and returns a string, as shown on line 1. In case of value `JstStmnt` we return strings: "by the fact that", "since" and "because"; otherwise we return string "by", as shown on lines 2–3 above.

It simply means that any one of three key phrases ("by the fact that", "since" and "because") may appear with the `Justification` formed by the `Statement` as shown below.

"**because** x is positive",
"**since** there is an x such that $x > 0$",
"**by the fact that** $x > 0$", etc.

In contrast, justifications formed by the rest of categories, the key phrase "by" is added as shown below:

"**by** the last statement",
"**by** substituting the value of x in equation the $y = 2 * x + 10$",
"**by** the definition of rationals", etc.

In structure, the list `Justifications` is similar to the the categories `Exps` and `Properties2` given on pages 111 and 119 respectively, and therefore, we define the abstract syntax in the similar way:

```
1  cat Justifications ; PJustifications ;
2  fun BasePJustifications : Justification -> Justification   -> PJustifications ;
3      ConsPJustifications : Justification -> PJustifications -> PJustifications ;
4
5      BaseJustifications : Justification   -> Justifications ;
6      ConsJustifications : PJustifications -> Justifications ;
```

However, the linearizations in the concrete syntax are a little different, as shown below, and followed by explanation:

```
1  lincat Justifications = List ;
2         PJustifications = {s : Str}
3
4  lin BasePJustifications x y  = { s = (JustifPhr x.t) ++ x.s ++ "and" ++  y.s };
5      ConsPJustifications x xs = { s = (JustifPhr x.t) ++ x.s ++ ","   ++ xs.s };
6
7      BaseJustifications x  = {s = (JustifPhr x.t)  ++ x.s  ; n = Sg} ;
8      ConsJustifications xs = {s = (JustifPhr xs.t) ++ xs.s ; n = Pl} ;
```

Note the use of `JustifPhr` in the above code. For instance, in the code (`JustifPhr x.t`) given on line 4 and others, we get the appropriate key phrase[s] according to the `JustifType` of `x` with (`x.t`).

Chapter 7

Macro Level CLM Grammar

Contents

7.1 Introduction **155**

7.2 Theorem and its Proof **155**

7.2.1 Proof and its Statements 156

7.2.2 Theorem and its Statements 170

7.3 Axiom **173**

7.4 Definition **174**

7.5 Conventions **176**

7.5.1 Ambiguity 177

7.1 Introduction

As a whole, a CLM document is a collection of axioms, definitions, propositions (theorems, lemma) and proofs, structured by specific keywords (Axiom, Definition, Lemma, Theorem and Proof). The text following these keywords is a list of sentences obeying different grammars (one for axioms, one for definition, etc). These grammars are not completely independent and share a lot of common rules. For instance, we use the micro level grammar (cf. Chapter 6) in almost all structural blocks with slight modifications.

7.2 Theorem and its Proof

A theorem and its proof are formed by two categories `ThmStmnts` and `PrfStmnts`, which are lists of the theorem statement `ThmStmnt` and the proof statement `PrfStmnt` respectively (given in subsequent sections).

We define a category `ThmPrf` and function `MkThmPrf` as shown below:

```
cat ThmPrf ;
fun MkThmPrf : ThmStmnts -> PrfStmnts -> ThmPrf ;
```

In its concrete syntax, we simply glue `ThmStmnts` and `PrfStmnts` together along with the keywords “Theorem” and “Proof”, as shown below:

```
1 lincat ThmPrf = { s : Str } ;
2 lin MkThmPrf thm prf = {
3           s = "Theorem" ++ thm.s ++ "Proof" ++ prf.s
4          } ;
```

7.2.1 Proof and its Statements

An introduction of proof block and its statements with respect to the mathematical texts is given in §3.2.4 on page 35. In contrast, its semantics is given in §4.3 (on page 61) and in definition 6 on page 66.

In CLM, a proof is a non-empty list of proof statements (`PrfStmnts`). It simply allows to write proof statements separated by full-stop. We describe it by the following pattern.

$\texttt{PrfStmnt}_1.$ $[\texttt{PrfStmnt}_2. \dots \texttt{PrfStmnt}_n.]$

Each proof statement in the above pattern (i.e. $\texttt{PrfStmnt}_1$, $\texttt{PrfStmnt}_2$, etc. Let us call it $\texttt{PrfStmnt}_k$) allows to write complex proof statements of the following pattern:

$\texttt{PrfStmnt}_{k_1}$; and $\texttt{PrfStmnt}_{k_2}$; and ... ; and $\texttt{PrfStmnt}_{k_n}$.

Note the use of semi-colon in "; and" to resolve the ambiguity between the `Statements` and the proof statement ($\texttt{PrfStmnt}_k$) on the use of conjunction. We interpret it as following:

$$\underbrace{\texttt{PrfStmnt}_{k_1}\text{; and } \underbrace{\texttt{PrfStmnt}_{k_2}\text{; and } \underbrace{\dots\text{; and } \texttt{PrfStmnt}_{k_n}}}}.$$

Two elaborative examples of such proof statement are following:

If $p \nmid a$ then $\texttt{gcd}(a, p) = 1$ **; and** therefore, by theorem 24, there are x and y such that $x * a + y * p = 1$ or $x * a * b + y * p * b = b$.

The following example is rather superficial but explains the pattern well. The conjunction **"; and"** may appear between different proof statements, the conjunction **", and"** (also disjunction ", or") may appear between different statements (i.e. `Statements`) and the conjunction **"and"** (also disjunction "or") may appear inside the category `Statement`.

We suppose that x **and** y are positive integers **, and** z is an even integer **; and**
we assume that $x > y$ **, and** $x = 1$ **; and**
therefore, we conclude that $y = 0$.

To implement it in the GF grammar, we can very well combine both patterns as shown below:

$\texttt{PrfStmnt}_1.$ $[(\texttt{PrfStmnt}_{2_1}$; and ... ; and $\texttt{PrfStmnt}_{2_n}). \dots \texttt{PrfStmnt}_n.]$

We define categories `PrfStmnts` and `PrfStmnt` simply as string records as shown in the following concrete syntax:

```
lincat PrfStmnt, PrfStmnts = { s : Str } ;
```

To make the non empty list `PrfStmnts`, we define three functions in the abstract syntax, as shown below.

```
fun BasePrfStmnts : PrfStmnt -> PrfStmnts ;
    ConsPrfStmnts : PrfStmnt -> PrfStmnts -> PrfStmnts ;
    ConjPrfStmnts : PrfStmnt -> PrfStmnts -> PrfStmnts ;
```

The function `BasePrfStmnts` takes a proof statement and returns the list `PrfStmnts`, covering the case when we have only one element in the list. The functions `ConsPrfStmnts` and `ConjPrfStmnts` add the proof statement in the list `PrfStmnts`. The concrete syntax given below reveals that the linearization of `ConsPrfStmnts` adds a full stop and linearization of `ConjPrfStmnts` adds a semi-colon and a conjunction:

```
lin BasePrfStmnts prf      = { s = prf.s ++ "."} ;
    ConsPrfStmnts prf prfs = { s = prf.s ++ "." ++ prfs.s } ;
    ConjPrfStmnts prf prfs = { s = prf.s ++ ";" ++ "and" ++ prfs.s } ;
```

The Proof Statement

A proof statement `PrfStmnt` is defined as followed:

7.2.1.1 Restatements

As described in §3.2.4 on page 35, we sometimes restate a goal or its logical equivalent, which we ought to prove. We form restatements in three ways:

1. By some key phrases (labeled as `ShallProveThat`), `Statements` and `Subordinate`.
2. By key phrases `ShallProveThat`, `EStatements` and `Subordinate`.
3. By key phrases `ShallProveThat`, `IfthenStmnt` and `Subordinate` (slightly different).

Instead of making another category which combines the `Statements`, `EStatements` and `IfthenStmnt`, we prefer to define three separate rules in the abstract syntax. Because these categories already combine and reuse many other categories, it seems unnecessarily complicated to combine them again for Haskell in the host system MathNat. However, we share their code as much as possible in the concrete syntax. **This argument holds for most of the statements defined for theorems, proofs, axioms and definitions.**

The First Rule for Restatement in Proof Statement is formed by key phrases `ShallProveThat`, `Statements` and `Subordinate`. For instance:

> "we have to prove that $x > y$, $x = y$ or $x < y$, where y is an integer",
> "it is sufficient to prove that $x > y$", etc.

We must define the operation function `ShallProveThat` and the `Subordinate` before. The function `ShallProveThat` is a collection of some key phrases as shown below:

> "we [(will | shall | have to)] (prove | show) that", "let us (prove | show) that", "it is sufficient to (prove | show) that".

Defining it is quite simple:

```
1 oper shallProveThat : Str = variants{
2       "we" ++ variants{""; "shall"; "will"; ["have to"]} ++
3                                       variants{"prove"; "show"} ++ "that" ;
4       ["it is sufficient to"]++ variants{"prove"; "show"}++ "that" ;
5       ["let us"] ++ variants{"prove"; "show"} ++ "that" ;
6     };
```

In contrast, the category `Subordinate` is simply the list `Statements`, attached with a bunch of key phrases. Its use is optional, as shown below with brackets ([...]).

[[,] (where | if | when | with the condition that | provided that) `Stmnts`.]

We define it as shown below:

```
cat Subordinate ;
fun MkSubord    : Statements -> Subordinate ;
    EmptySubord :               Subordinate ;
```

Without surprise, the corresponding concrete syntax is:

```
lincat Subordinate = {s = Str } ;
fun MkSubord stmnts = { s = optComma ++ variants{
      "where"; "when"; "if"; ["provided that"]; ["with the condition that"]
      } ++ stmnts.s
    } ;
    EmptySubord = {s = ""} ;
```

Note that the function `EmptySubord` is an empty string making `Subordinate` optional. Also recall that `optComma` is an `oper` function which adds an optional comma (defined on page 137).

Returning back to our main rule, we define it as shown below:

```
cat PrfStmnt ;
fun MkRestmnt : Statements -> Subordinate -> PrfStmnt ;
```

In the corresponding concrete syntax, we simply have to glue `Statements` and `Subordinate` together along with `ShallProveThat` key phrases. Also note that, we do not need to access the function `ShallProveThat` with (`.s`) (on line 2 below), because it is already a string (i.e. `Str`) instead of the string record (i.e. `{s:Str}`)

```
lincat PrfStmnt = {s = Str} ;
fun MkRestmnt stmnts sbrd = {s = ShallProveThat ++ stmnts ++ sbrd.s} ;
```

The Second Rule for Restatement in proof Statement is formed by `ShallProveThat` key phrases, `EStatements` and `Subordinate`.
For instance:

"we have to prove that either $x > y$, $x = y$ or $x < y$, where y is an integer",
"it is sufficient to prove that either $x > y$ or $y > z$", etc.

Its implementation is similar to the first rule for restatements above (on page 157). We just need to replace the category `Statements` with `EStatements`, as shown below in the abstract syntax. We omit the concrete syntax because of their similarity.

```
fun MkERestmnt : EStatements -> Subordinate -> PrfStmnt ;
```

The Third Rule for Restatement in Proof Statement is formed by `ShallProveThat` key phrases, `IfthenStmnt` and `SubordinateIfThen`.
For instance:

"it is sufficient to prove that if $x \in A$ then $x \in B$ where $A \subseteq B$" , etc.

Again, the corresponding implementation is similar to the first rule for restatements on page 157 and we only need to replace the category `Statements` with `IfthenStmnt`. Because of that we omit its implementation.

Note that the category `SubordinateIfThen` is the same as the category `Subordinate` (defined on page 158). Recall that `Subordinate` is simply `Statements`, attached with a bunch of key phrases. One of these key phrases is "if", which may produce syntactically ill-formed sentences in the context of this rule (as shown in the example below). Therefore we omit it from the definition of `SubordinateIfThen`.

*We shall prove that if $x \in A$ then $x \in B$ **if** $A \subseteq B$.

7.2.1.2 Assumptions

In a proof it is common to assume facts needed to prove the theorem (cf. 3.2.4 on page 36). These are also called hypotheses. An assumption or hypothesis is formed in many ways as shown below:

The First Rule for Assumptions in Proof Statement is formed by the 'let' keyword, `LetStatements` and `Subordinate`. For instance,

"let A and B be two sets, and $A \subseteq B$",
"let m and n be relatively prime integers",
"let $m = n$, and $n = r$", etc.

We define a function `MkLet` as shown below:

```
fun MkLet : LetStatements -> Subordinate -> PrfStmnt ;
lin MkLet stmnts sbrd = {s = "let" ++ stmnts.s ++ sbrd.s } ;
```

The Second Rule for Assumptions in Proof Statement is formed by `AsmThat` key phrases, `Statements` and `Subordinate`. For instance,

"suppose that A and B are two sets",
"assume that m and n are relatively prime integers",
"we can suppose that there are two integers u and v such that $u * n + v * m = 1$",
"we suppose that $\sqrt{2} = \frac{a}{b}$, where a and b are non zero integers with no common factor",
"we assume that $x > a$, $x > b$**, and** $x > c$, where a, b and c are integers", etc.

Its implementation is similar to the rule for let statements above (on page 159). We just need to replace string "let" with `AsmThat` key phrases in the concrete syntax as shown below.

```
fun MkAssume : Statements -> Subordinate -> PrfStmnt ;
lin MkAssume stmnts sbrd = {s = AsmThat ++ stmnts.s ++ sbrd.s} ;
```

Where, the operation function `AsmThat` is a collection of some key phrases as shown below:
"(let | [we [can]] (suppose | assume) that | we can (write | say) that)".

Defining it is also quite simple:

```
oper AsmThat : Str = variants{
         variants{""; "we"} ++ variants{""; "can"} ++
                              variants{"suppose"; "assume"} ++ "that" ;
         ["we can"] ++ variants{"say"; "write"} ++ "that"
     } ;
```

The Third Rule for Assumptions in Proof Statement Similar to the second rule, it is formed by `AsmThat` key phrases, `EStatements` and `Subordinate`. For instance,

> "suppose that either A and B are two sets **, or** A and B are positive integers",
> "we assume that either m and n are relatively prime**, or** they are not relatively prime", etc.

We omit its implementation because of its similarity with the second rule above (on page 159). Of course we can combine it the second rule. But as already said, we gain nothing in doing so, except unnecessarily complicating the abstract syntax.

7.2.1.3 Assumptions with Justifications

Although it is rare but we may have to give justifications (cf. §3.2.4 on page 36) for an assumption to make sense. It is formed by some key phrases, `Statements`, `Justifications` and `Subordinate` in the following two patterns:

1. (we (suppose | assume) that | ...) [,] `Statements Justifications`[1] [,] [(where | ...) `Statements`].

2. `Justifications`[1] [,] (we (suppose | assume) that | ...) `Statements` [,] [(where | ...) `Statements`].

Some examples of *Assumptions with justifications* are following:

> "we suppose that $\sqrt{2} = \dfrac{a}{b}$ by the definition of rational number",

[1]Recall that the key phrases: (because | since | by the fact that | by) are added in each element of the list `Justifications`. See page 152.

"we can write that $2 * b^2 = (2 * c)^2 = 4 * c^2$ by substituting the value of a into equation (i)",
"by the definition of rational numbers, we suppose that $\sqrt{2} = \frac{a}{b}$",
"by substituting the value of a into equation (i), we can write that $2*b^2 = (2*c)^2 = 4 * c^2$ ",
"we can write that $a = 2 * c$ by the definition of even numbers, where c is an integer.", etc.

Regarding the implementation of these patterns in the GF grammar, can we build it in one function? Let us try the following abstract syntax:

```
fun MkAssumeJustif : Statements -> Justifications -> Subordinate -> PrfStmnt ;
```

Then the corresponding concrete syntax could use `variants` for both patterns as shown below:

```
1   lin MkAssumeJustif stmnts jstif sbrd = {s = variants {
2        Linearization of pattern 1 showing stmnts before jstif and sbrd;
3        Linearization of pattern 2 showing jstif before stmnts and sbrd
4        }
5      } ;
```

However there is a problem with this approach. When the syntactical part is done, this grammar is called by the host system MathNat. In it we can access only he abstract syntax representing the ontology. The abstract syntax of function `MkAssumeJustif` currently cannot indicate which pattern is actually used in the linearization. Therefore, in case of appearing any of these patterns, we always see that the `Statements` part appears first, and followed by `Justifications` part, which is then followed by the `Subordinate` (i.e. `Statements -> Justifications -> Subordinate`).

However, the order in which these categories appear in these patterns are important for the anaphoric resolution (cf. §2.5.2 and §8.3.1.1). We explain it with the help of the following superficial examples:

Pattern 1: We assume that $x * y > 0$ because $\underbrace{\text{it is positive}}_{\texttt{Justifications}}$.

Pattern 2: Because $\underbrace{\text{it is positive}}_{\texttt{Justifications}}$, we assume that $x * y > 0$.

In the first example, pronoun "it" refers to $x * y$, whereas in the second example it refers to some object before this sentence. Therefore, we cannot merge these patterns in one function.

Consequently, we define two functions for these two patterns in the following abstract syntax:

```
fun MkAssumeJustif1 : Statements -> Justifications -> Subordinate -> PrfStmnt ;
    MkAssumeJustif2 : Justifications -> Statements -> Subordinate -> PrfStmnt ;
```

And the corresponding concrete syntax is straight forward:

```
1 lin MkAssumeJustif1 stmnts jstf sbrd = {s =
2      AssumeThat ++ stmnts.s ++ jstf.s ++ sbrd.s};
3
4    MkAssumeJustif2 jstf stmnts sbrd = {s =
5      jstf.s ++ optComma ++ AssumeThat ++ stmnts.s ++ sbrd.s };
```

In line 2 and 4, we give the `JustifType` of justification in question to `JustifPhr` to get the appropriate string value.

7.2.1.4 Deductions

As described in §3.2.4 on page 36, in a proof, deductions are the *results* which we deduce from existing premises and known knowledge. We form deduction in two ways as shown below:

The first rule for deduction in proof statement is formed by the `ConcludeThat` key phrases, `Statements` and `Subordinate`. Some of the examples are following:

> "we conclude that $\sqrt{2}$ is an irrational number",
> "it implies that m and r are coprime **, and** $r < m$",
> "it implies that $\underbrace{\text{there exist } m \text{ and } n \text{ such that } m * a + nb = r \text{ or } 2 * (m * a + nb) = 2 * r}$ **, and** $n < m$", etc.

> Note that the anaphora of propositions (i.e. pronoun 'it' above) is not resolved. It is because each step in the proof is a consequence of the previous step(s).

The operation function `ConcludeThat` is a collection of key phrases as shown below: "we (get | conclude | deduce) that", "it is (clear | obvious | trivial | evident | easy to see) that", "it (implies | means) that".

We omit its implementation because of its technical similarity with other operation functions.

The corresponding abstract and concrete syntax is:

```
fun MkDeduce : Statements -> Subordinate -> PrfStmnt ;
lin MkDeduce stmnts sbrd = {s = ConcludeThat ++ stmnts.s ++ sbrd.s} ;
```

The second rule for deduction in proof statement Formed by the `ConcludeThat` key phrases, `EStatements` and `Subordinate`.
For instance:

> "we conclude that either $\sqrt{2}$ is an irrational number or $\sqrt{2}$ is not an irrational number", etc.

We omit its implementation because of the technical similarity with the first rule for deduction given above.

7.2.1.5 Deductions with Justifications

It is common to give justification(s) for deduced facts. It is formed in the following ways:

The first rule for deduction with justifications in proof statement is formed by `ConcludeThat` key phrases, `Statements`, `Justifications` and `Subordinate`, producing the following pattern:

- [(we (conclude | deduce) that | ...)] `Statements` [,] `Justifications`[2] [,] [(where | ...) `Statements`]

For example,

> "a^2 is even because it is a multiple of 2",
> "we get that $2 * b^2 = 4 * c^2$ by substituting the value of a into equation (1)",
> "there exist q and r such that $n = m * q + r$ by euclidean division",
> "q divides r because $r = n - m * q$", etc.

We implement it as following:

```
fun MkDeduceJustif1 : Statements -> Justifications -> Subordinate -> PrfStmnt ;
lin MkDeduceJustif1 : stmnts jstfs sbrd = {
  s = variants{ConcludeThat; ""} ++ stmnts.s ++ jstfs.s ++ sbrd.s} ;
```

The second rule for deduction with justifications in proof statement is formed by `Justifications`, `Statements`, optional key phrases `ConcludeThat` and `Subordinate`, producing the following pattern:

- (because | since | ...) `Justifications` [,] [(we (conclude | deduce) that | ...)] `Statements` [,] [(where | ...) `Statements`].

Examples of above pattern are:

> "because b is positive and it is a multiple of 2, we conclude that b^2 is even",
> "because $p|a * b$ and $p|p * b$ it is clear that $p|b$",
> "by induction hypothesis, there are u' and v' such that $u' * m + v' * r = 1$", etc.

Because these patterns have the same structure (i.e. the categories will have the same order in the abstract syntax), we can merge them to one rule as shown below.

```
fun MkDeduceJustif2 : Justifications -> Statements -> Subordinate -> PrfStmnt ;
lin MkDeduceJustif2 : js stmnts sbrd = {s =
     js.s ++ optComma ++ variants{ConcludeThat; ""} ++ stmnts.s ++ sbrd.s
   };
```

The third rule for deduction with justifications in proof statement is formed by `Justifications'`, `ShowThat`[3] key phrases, `Statements` and an optional `Subordinate` forming the following patterns.

1. `Justifications'` [,] (shows that | establishes that | ...) `Statements` [,] [(where | ...) `Statements`].

[2] Recall that the key phrases: (because | since | by the fact that | by) are added in each element of the list `Justifications`. See page 152.

[3] `ShowThat` key phrases are: (shows | establishes | yields | proves | returns | gives) (that | the (result | fact) that).

For example,

"squaring both sides shows that $2 * b^2 = a^2 - (1)$",
"dividing both sides by 2 and the fact that x is even, yields the result that $b^2 = 2 * c^2$",
"the fact that x is even and $y = x$ yields the result that y is even",
"the fact that x is even yields the result that 2 divides x", etc.

By observing the above examples, we can guess that for each element in the list `Justifications'`, if it is formed by `Statement`, we add a phrase "the fact that", otherwise we add nothing.

In the implementation shown below, it is very similar to the implementation of `Justifications` (on page 152). Therefore, instead of implementing another category we reuse the category `Justifications`. But we need to modify its concrete syntax to make it compatible to our new needs. Let us first recall its abstract and concrete syntax:

```
cat Justifications ; PJustifications ;
fun BasePJustifications : Justification  -> Justification   -> PJustifications ;
    ConsPJustifications : Justification  -> PJustifications -> PJustifications ;
    BaseJustifications  : Justification  -> Justifications ;
    ConsJustifications  : PJustification -> Justifications ;

lincat Justifications, PJustifications = List ;
lin    BasePJustifications x y = {
           s = (JustifPhr x.t) ++ x.s ++ "and" y.s ; n =  Pl
          };
       ConsPJustifications x xs = {
           s = (JustifPhr x.t) ++ x.s ++ "," ++ xs.s ; n = Pl
          };
      BaseJustifications x  = {s = (JustifPhr x.t)  ++ x.s  ; n = Sg} ;
      ConsJustifications xs = {s = (JustifPhr xs.t) ++ xs.s ; n = Pl} ;
```

In the linearization of `Justifications` and `PJustifications`, instead of defining them as `oper` record `List` (defined as: `{s : Str ; n : Number}`), we define them as shown below:

```
oper   JustifRec : Type = {s : JustPhrType => Str ; n : Number} ;
lincat Justifications, PJustifications = JustifRec ;
```

`JustPhrType` used above, is an algebraic data type for different types of Justifications based on their usage in different rules. For instance, in the code below, the constructor `JustPhrs1` represents the linearization of the Justifications we have used so far[4]. In contrast, we use the constructor `JustPhrs2` in the current rule.

```
param JustPhrType = JustPhrs1 | JustPhrs2 ;
```

Similar to the `oper` function `JustifPhr`, we need another function needed for this rule. But first we rename `JustifPhr` to `JustifPhr1` for harmony in their names. Then we define another function with the name `JustifPhr2` as shown below:

[4] In the "Assumptions with Justifications", and in the first and second rule of "Deductions with Justifications".

```
oper JustifPhr1 : JustifType -> Str = \t -> JustifPhr t ;

     JustifPhr2 : JustifType -> Str = \t -> case t of {
       JstStmnt => ["the fact that"];
       JstRest  => "" } ;
```

Now we redefine the linearization of functions that belong to categories `Justifications`, `PJustifications`. For that we first define an `oper` function `mkJustif` use it the linearizations of all functions:

```
1   oper mkJustif : JustifType -> Number -> Str -> Str -> Str -> JustifRec =
2     \t,n,x,s,y -> {s = table {
3             JustPhrs1 => (JustifPhr1 t) ++ x.s ++ s ++ y.s ;
4             JustPhrs2 => (JustifPhr2 t) ++ x.s ++ s ++ y.s ;
5           } ;
6          n = n
7         } ;
8   lin BasePJustifications x y  = mkJustif x.t Pl x.s "and" y.s ;
9       ConsPJustifications x xs = mkJustif x.t Pl x.s ","  xs.s ;
10
11      BaseJustifications x  = mkJustif  x.t Sg  x.s "" "" ;
12      ConsJustifications xs = mkJustif xs.t Pl xs.s "" "" ;
```

Note the use of empty strings (`""`) on lines 11–12 above. It just saves us from defining one more `oper` function for these two functions.

Returning to our main topic, we now define the implementation of rule three. Note the use of `js.s!JustPhrs2` below to access the second type of justifications:

```
fun MkDeduceJustif3 : Justifications -> Statements -> Subordinate -> PrfStmnt ;
lin MkDeduceJustif3 : js stmnts sbrd = {
     s = js.s!JustPhrs2 ++ optComma ++ ShowThat ++ stmnts.s ++ sbrd.s
     } ;
```

A closer look to the rules two and three reveals that these two patterns have the same structure. We mean that in these rules the categories have the same order in the abstract syntax. Because only the abstract syntax is visible in the host system MathNat, merging them in one rule does not raise any issue for the anaphoric resolution. So instead of defining `MkDeduceJustif3`, we add its concrete syntax to the function `MkDeduceJustif2` as variants:

```
fun MkDeduceJustif2 : Justifications -> Statements -> Subordinate -> PrfStmnt ;
lin MkDeduceJustif2 : js stmnts sbrd = {s = variants {
     js.s!JustPhrs2 ++ optComma ++ variants{ConcludeThat; ""} ++ stmnts.s ++ sbrd.s ;
     js.s!JustPhrs2 ++ optComma ++ ShowThat ++ stmnts.s ++ sbrd.s
    };
```

The fourth rule for deduction with justifications in proof statement is formed by `Statements` (as justifications), `Statements` (as conclusions) and an optional `Subordinate` forming the following patterns.

- since `Statements`[,] then `Statements` [,] [(where | ...) `Statements`].

For instance,

"since $A \subseteq A \cup B$, then $x \in A \cap B$",
"since $A \subseteq A \cup B$ and $B \subseteq A \cup B$ then $A = B$", etc.

Its abstract and concrete is shown below:

```
fun MkDeduceJustif3 : Statements -> Statements -> Subordinate -> PrfStmnt ;
lin MkDeduceJustif3 : js stmnts sbrd = {
      s = "since" ++ js.s ++ optComma ++ "then" ++ stmnts.s ++ sbrd.s ;
    };
```

7.2.1.6 Miscellaneous Proof Statements

1. The statement formed by `TakeStmnt` (on page 147) without adding anything further. Therefore, its abstract and concrete syntax is very simple:

```
fun Take : TakeStmnt -> PrfStmnt ;
lin Take take = take ;
```

2. Key phrases such as (then | thus | so | therefore | now | in this case [,]) could be added to any `PrfStmnt` with an exception that it cannot be the first statement in the proof.

```
fun ThereforePrfSt : PrfStmnt -> PrfStmnt ;
lin ThereforePrfSt prf = {s = therefore ++ prf.s} ;

oper therefore : Str = (variants{"then"; "thus"; "so"; "therefore";
             "hence"; "now"; ["in this case"]} ++ optComma) ;
```

   For instance,
   "therefore , we conclude that $\sqrt{2}$ is an irrational number", etc.

3. Key phrases such as (it is trivial | this (ends | concludes) the proof | QED) **terminates** a proof. However it is not always necessary to terminate a proof with such key phrases as the system MathNat can figure out the end of a proof in many cases.

4. Key phrase "we (will | shall) prove it later" terminates a proof as **unfinished**.

5. Key phrases such as (it is (a contradiction | impossible) | it cannot be possible) or (it is contrary to (the | our) hypothesis), to trigger a **contradiction**.

6. The following pattern to trigger a **contradiction with some justification(s)**.
   [(the (fact | result) that)] `Statements` ((implies | is) a contradiction | is (impossible | not possible) | cannot be possible).

   For instance,
   "the fact that x is positive implies a contradiction",

7. There are some other statements as well which we cover in §7.2.1.7.

7.2.1.7 Proof by Cases

'Proof by case' or 'proof by exhaustion' is an important proof method. As demonstrated in Chapter 3 (§3.2.4.1, page 37), there are numerous patterns which are used in the mathematical texts for proof by case method. However, CLM currently supports only two with strict formating. We present below these patterns followed by some examples.

Pattern 1: We proceed by case analysis.

Case: *Condition* `PrfStmnts` [This ends the case.]
Case: *Condition* `PrfStmnts` [This ends the case.]
......
Case: *Condition* `PrfStmnts` [This ends the case.]
[This was the last case.]

Consider the same theorem and its proof in this pattern:

Theorem. Assume that m and n are coprime integers. Assume that either $m \neq 0$ or $n \neq 0$. Prove that there exist two integers u and v such that $u * n + v * m = 1$.
Proof. We proceed by case analysis.
Case: $n = 0$. In this case $m = 1$ because m and n are coprime. Therefore we can choose $u := 0$ and $v := 1$. This ends the case.

Case: $m = 0$ and $n = 1$. In this case we can choose $u := 1$ and $v := 0$.

Case: $n \neq 0$ and $m \neq 0$. By euclidean division there are r and q such that $n = m * q + r$. It is trivial that m and r are coprime and $r < m$. So there are u' and v' such that $u' * m + v' * r = 1$. It implies that $u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$. So we can choose $u := v'$ and $v := u' - v' * q$.

Pattern 2:

If *condition* then `PrfStmnts`
Otherwise if *condition* then `PrfStmnts`
Otherwise if *condition* then `PrfStmnts`
......
Otherwise `PrfStmnts`

Consider the same theorem and its proof in this pattern:

Theorem. Assume that m and n are relatively prime integers. Assume that either $m \neq 0$ or $n \neq 0$. Prove that there exist two integers u and v such that $u * n + v * m = 1$.
Proof. If $n = 0$ then $m = 1$ because m and n are coprime. We can choose $u := 0$ and $v := 1$.
Otherwise if $m = 0$ and $n = 1$ then we can choose $u := 1$ and $v := 0$.
Otherwise there exist r and q such that $n = m * q + r$ by euclidean division. [...] So we can choose $u := v'$ and $v := u' - v' * q$.

These patterns may have nested cases, and we can even intermix them in a proof. The first pattern is simpler than the second one. In it, although the markers such as: "This ends the case" and "This was the last case" are optional and there is a possibility for such proofs to be ambiguous. But the marker such as "Case:" still allow us to decide whether a proof branch is already completed or not. In a similar way when we have nested cases, the marker "We proceed by case analysis" helps to disambiguate between the nested cases.

Now let us consider the pattern two. In it the markers 'if' and 'otherwise if' allows us to define the boundaries of a branch. But what should we do when another conditional

appears inside a branch? It is better explained with the (superficial) example as shown bellow:

Proof. Let x and y be integers.
If $x * y > 0$ then it is positive. **If** there is an integer z such that it is equal to $x * y$ then $z > 0$.
Otherwise if $x * y = 0$ then $x = 0$ or $y = 0$.
Otherwise $x * y < 0$.

Which 'if' marks the boundary of this case? There is no way to disambiguate on the basis of syntax and we need a very deep semantic analysis for it (which is a hard problem). Because our disambiguation process is solely based on syntax, we are forced to return two interpretations.

The first interpretation makes a new branch of the proof on first conditional. The second conditional appears as a statement of this branch. The second interpretation takes the first conditional as a normal proof statement. It is then the second conditional that makes a first branch of proof by case. The situation gets more interesting when we intermix both patterns (but necessarily ambiguous). See §8.3.11 on page 221 for the disambiguation algorithm.

In the GF grammar, the specific statements for cases (i.e. case markers) are simply treated as proof statements. It means that the result given by GF, after parsing a proof (containing cases), is also a list of proof statements. We re-build case structure in the host system MathNat by working on the abstract syntax trees of the proof statements. Agian, see §8.3.11 on page 221 for the algorithm.

This approach is better because of various reasons, some of them are given below:

- Like bnf grammar, attribute grammar is also insufficient to capture the richness of the proof by case method.

- If we still insist on defining an attribute grammar, we would not be able to parse incomplete proofs.

- Using the approach mentioned above, incomplete proofs could be parsed and we can return the error messages more precisely because we know the context at each stage.

- We can reduce ambiguity with the semantics of the proof at runtime (if we plug-in MathAbs to some proof assistant in the future).

We now define the specific statements for cases, as shown below:

1. Case markers for pattern 1:

   - The function `CaseAnls` for "we proceed by [the] case analysis"
   - The function `Case` that takes `Statements` to form *condition*:

   ```
   fun Case : Stmnts -> PrfStmnt ;
   lin Case conds = {s= "case" ++ variants{":"; "."} ++ conds.s};
   ```

   - The function `EndCase` for "this ends the case"
   - The function `EndLastCase` for "it was the last case"

2. The statement formed by conditional `IfthenStmnt` (on page 146) and `Subordinate`. It may also be used as a general proof statement. In fact, it is the context which decides whether we consider it a start of a new branch in the proof or just a simple statement. Its implementation is straight forward:

```
fun MkIfthen1 : IfthenStmnt -> Subordinate -> PrfStmnt ;
lin MkIfthen1 ifthen sbrd = {s = ifthen.s ++ sbrd.s} ;
```

3. The conditional statement formed by `Statements` and `TakeStmnt` and `Subordinate`, as shown below in the abstract syntax:

```
fun MkIfthen2 : Stmnts -> TakeStmnt -> Subordinate -> PrfStmnt ;
```

For instance,

"if $m = 0$ and $n = 1$ then we can choose $u := 1$ and $v := 0$", etc.

4. The conditional statement formed by `IfthenStmnt`, `Statements` and `Subordinate`:

```
fun MkIfthen3 : PropIfthen -> Stmnts -> PrfStmnt ;
```

For instance,

"if $n = 0$ then $\underbrace{m = 1 \text{ because } m \text{ and } n \text{ are coprime}}$", etc.

5. The *otherwise* statements:

- Formed by `PrfStmnt`. It is quite similar to the function `ThereforePrfSt` in its structure:

```
fun Otherwise1 : PrfStmnt -> PrfStmnt ;
lin Otherwise1 prfStmnt = {s =
     "otherwise" ++ optComma ++ prfStmnt.s} ;
```

For instance,

"otherwise we assume that x is positive", etc.

- Formed by (`Statements` | `EStatements`) and `Subordinate`:

```
fun Otherwise2 : Statements -> Subordinate -> PrfStmnt ;
fun Otherwise3 : Statements -> Subordinate -> PrfStmnt ;

lin Otherwise2 stmnts sbord = {s =
      "otherwise" ++ optComma ++ stmnts.s ++ sbord.s} ;
lin Otherwise3 estmnts sbord = {s =
      "otherwise" ++ optComma ++ estmnts.s ++ sbord.s} ;
```

These statements are not allowed next to each other and with `ThereforePrfSt`.

7.2.2 Theorem and its Statements

An introduction of theorem block and its statements with respect to the mathematical texts is given in §3.2.3 on page 33. In contrast, its semantics is given in §4.3 (on page 61) and in definition 7 on page 67.

In CLM, a theorem is a non-empty list of theorem statements (`ThmStmnts`). It could be formed by the following pattern, in which it simply allows to write theorem statements separated by full-stop:

$\texttt{ThmStmnt}_1$. [$\texttt{ThmStmnt}_2$. ... $\texttt{ThmStmnt}_n$.]

(It is the host system MathNat which enforce the semantics of theorem. See §8.3.8 on page 213 for more details.)

We define categories `ThmStmnts` and `ThmStmnt` simply as string records. Further, to make the non empty list `ThmStmnts`, we define two functions in the abstract syntax, as shown below.

```
lincat ThmStmnt, ThmStmnts = {s : Str} ;
fun BaseThmStmnts : ThmStmnt -> ThmStmnts ;
    ConsThmStmnts : ThmStmnt -> ThmStmnts -> ThmStmnts ;
```

And the corresponding concrete syntax is very simple:

```
1 lin BaseThmStmnts thm      = { s = thm.s ++ "."} ;
2     ConsThmStmnts thm thms = { s = thm.s ++ "." ++ thms.s } ;
```

The Theorem Statement

The theorem statement `ThmStmnt` is defined as follows:

7.2.2.1 Statement to Prove

The main part of the theorem is the proposition or statement that that we have to prove. It is called the *goal* which we have to prove in the *proof*. It should be the final statement in the theorem.

The first rule for goal in theorem The most basic way in which the theorem could be written is symbolic, using ⟨*Formula*⟩ directly (given in figure 6.2 on page 109). However, there is a condition that ⟨*Formula*⟩ must start with universal or existential quantifier. In the GF grammar, it is simply defined as shown below:

```
fun MkThmSymb     : SymbFormula -> ThmStmnt ;
fun MkSymbFormula : String      -> SymbFormula ;
```

We can see that `SymbFormula` is a string. However, by defining this new category (`SymbFormula`), we enforce it to be different from `Exp` and `Eq`. Therefore, the semantic conditions can be applied distinctly in the host system MathNat. A few examples are following:

The formula: $\forall(x, y : \mathbb{Z})\,(\text{even}(x) \wedge \text{even}(y) \Rightarrow \text{even}(x+y))$, for statement:
"if x and y are two even integers then $x+y$ is even"

The formula: $\forall(A, B : \mathbb{S}\text{et})\,((A \cup B = A \cap B) \Rightarrow A \subseteq B)$, for statements:
"for all sets A and B, if $A \cup B = A \cap B$ then $A \subseteq B$"

The formula: $\forall x : \mathbb{Z}\ (\exists\, y : \mathbb{Z}(x+1=y))$, for statement:
"for all integers x, there is an integer y such that $x+1=y$", etc.

The second and third rules for goal in theorem These are formed by miscellaneous key phrases (given below), (`Statements` | `EStatements`) and `Subordinate`, and forms the following pattern:

- [(prove | show) that] (`Statements` | `EStatements`) [,] `Subordinate`

Some example statements are:

"prove that there exists two integers u and v such that $u * n + v * m = 1$",
"prove that either $x > y$, $x = y$ or $x < y$, where y is an integer",
"show that $x+y$ is even", etc.

Due to the similarity between these two rules, we give concrete syntax only for one rule:

```
fun ThmProves  : Statements  -> Subordinate -> ThmStmnt ;
    ThmEProves : EStatements -> Subordinate -> ThmStmnt ;

lin ThmProves stmnt sbrd = {s = ProveThat ++ stmnt.s ++ optComma ++ sbrd.s} ;

oper ProveThat : Str = variants{["prove that"]; ["show that"] ; ""} ;
```

The fourth rule for goal in theorem It is formed by miscellaneous key phrases (given below), `IfthenStmnt` and `SubordinateIfThen`, and forms the following pattern:

- [(prove | show) that] `IfthenStmnt` [,] `SubordinateIfThen`

Where, the category `SubordinateIfThen` is defined in §7.2.1.1 on page 159. Some example statements are:

"prove that if $x \in A$ then $x \in B$",
"show that if $A \cup B = A \cap B$ then $A \subseteq B$",
"if a is integer then there is no integer x such that $a < x < a+1$", etc.

It implementation is very similar to the above rules:

```
fun ThmProvesIfThen : IfthenStmnt -> SubordinateIfThen -> ThmStmnt ;
lin ThmProvesIfThen stmnt sbrd = {s = ProveThat ++ stmnt.s ++ optComma ++ sbrd.s} ;
```

The fifth, sixth and seventh rule for goal in theorem These are formed by miscellaneous key phrases (given below), an optional `Typ`, `Exps`, (`Statements` | `EStatements` | `IfthenStmnt`) and `Subordinate`, and forms the following pattern:

- [(prove | show) that] for (all | every) [`Typ`] `Exps` , (`Statements` | `EStatements` | `IfthenStmnt`) [,] `Subordinate`

Some example statements are:

"prove that for (all | every | arbitrary) integer[s] x, x is positive and even",
"for (all | every | arbitrary) number[s] x and y, x is an element of y",
"for (all | every | arbitrary) x and y, x does not multiply y",
"show that for (all | every | arbitrary) a and b, there is n such that $n * a > b$",
"for (all | every | arbitrary) x and y, $(x + y)^2 = x^2 + y^2 + 2 * x * y$",
"for (all | every | arbitrary) a if it is an integer then there is no integer x such that $a < x < a + 1$", etc.

Of course we could have merged these rules with the second, third and fourth rule respectively, but we prefer not to do so. Because there are no big gains. In this way we avoid unnecessary complicated categories and make it simple. Anyway we share concrete syntax between them as much as possible.

For GF, implementation we first need a new category for an optional `Typ`, labeled as `TypeOpt`, as defined below:

```
cat TypeOpt ;
fun EmptyTOpt : TypeOpt ;
    MkTOpt    : Typ -> TypeOpt ;
```

In its concrete syntax the category `TypeOpt` is similar to `Typ`. In the linearization of the function `EmptyTOpt`, both singular and plural values of its record table are empty. In contrast the function `MkTOpt` simply resides `Typ` in it without any change.

```
lincat TypeOpt = {s : Number => Str } ;
fun EmptyTOpt   = { s = table {_ => "" }} ;
    MkTOpt type = type ;
```

We define below the fifth rule, which combines some key phrases, `TypeOpt`, `Exps`, `Statements` and `Subordinate`.

```
1  fun ThmProvesR5 : TypeOpt -> Exps -> Statements -> Subordinate -> ThmStmnt ;
2  lin ThmProvesR5 t vars stmnt = {s = ProveThat ++
3       variants{
4       ["for all"]   ++ t.s!Pl         ++ vars.s ++ optComma ++ stmnt.s ;
5       ["for every"] ++ t.s!Sg         ++ vars.s ++ optComma ++ stmnt.s ;
6       "for" ++ artIndef!vars.n ++ "arbitrary" ++ t.s!vars.n ++ vars.s ++ optComma ++ stmnt.s
7       } };
```

Note the lines 3–5 above. The phrase "for all" always needs the plural form of `TypeOpt`. In contrast, the phrase "for every" always needs the singular form of `TypeOpt`. However, in line 7, we need number agreement with `TypeOpt`. Also if the variable is singular then we need an article between "for" and "arbitrary". For instance:

"for an arbitrary integer x, ... ",
"for arbitrary integers x and y, ... "

We apply a semantic condition that variables mentioned in the first part must be used in the statement.

7.2.2.2 Assumptions in theorem

We construct the assumptions in the same way as we do for assumptions in the proof (cf. §7.2.1.2 on page 159). Therefore, we only give below its abstract syntax:

Assumptions in theorem provides a starting point to the author for writing its proof. Assumptions cannot be the last statement of the theorem.

```
fun MkThmLet     : LetStatements  -> Subordinate -> ThmStmnt ;
    MkThmAssume  : Statements     -> Subordinate -> ThmStmnt ;
    MkThmEAssume : EStatements    -> Subordinate -> ThmStmnt ;
```

This abstract syntax indicates that all of the categories (such as `Statements`, `EStatements`, `LetStatements`, `Subordinate`) are reused again. In a similar way the corresponding concrete syntax is also reused from the assumptions given as proof statement (show below after this paragraph). However, we need to define these functions to record these theorem statements for denotational semantics and to linguistic features to apply in the host system MathNat.

```
fun MkThmLet     st sbord = {s = "let"      ++ st.s ++ optComma ++ sbord.s } ;
    MkThmAssume  st sbord = {s = AssumeThat ++ st.s ++ optComma ++ sbord.s } ;
    MkThmEAssume st sbord = {s = AssumeThat ++ st.s ++ optComma ++ sbord.s } ;
```

7.2.2.3 Miscellaneous Key Phrases for theorem statement

Key phrases such as (then | thus | so | therefore | now) could be added to any `ThmStmnt` with an exception that it cannot be the first statement in the theorem. We define its abstract syntax as shown below:

```
  fun ThereforeThmSt : ThmStmnt -> ThmStmnt ;
```

7.3 Axiom

As described in §3.2.1, an axiom consists of a statement or a few, expressing a proposition or fact that is considered to be true without a proof. In CLM, we reuse all the statements from theorem for axiom after removing the key phrases "(prove | show) that").

We would like to reuse the code written for theorem statements. One solution among many would be to define a category `AxThmGoalStmnt` to save the statements given as goal in theorem. But it will not contain the optional key phrases: "(prove | show) that"). It is because we do not share these phrases in axiom.

As an example, we redefine the function `ThmProvesR2` for the second rule given in §7.2.2.1 on page 171, by removing the `oper` function `ProveThat`, as shown below.

```
cat AxThmGoalStmnt ;
fun AxThmProvesR2 : Statements -> Subordinate -> AxThmGoalStmnt ;
lin AxThmProvesR2 stmnt sbrd = {s = stmnt.s ++ sbrd.s} ;
```

Similarly, for function ThmProvesR5 defined in §7.2.2.1 on page 172, we simply need to change its output category from ThmStmnt to AxThmGoalStmnt and remove the use of oper ProveThat from its concrete syntax:

```
1 fun AxThmProvesR5 : TypeOpt -> Exps -> Statements -> Subordinate -> AxThmGoalStmnt ;
2 lin AxThmProvesR5 t vars stmnt = {s = variants{
3      ["for all"]   ++ t.s!Pl         ++ vars.s ++ optComma ++ stmnt.s ;
4      ["for every"] ++ t.s!Sg         ++ vars.s ++ optComma ++ stmnt.s ;
5      ["for arbitrary"] ++ t.s!vars.n ++ vars.s ++ optComma ++ stmnt.s
6      } };
```

Now from the category AxThmGoalStmnt, we can make the theorem statement as shown below:

```
fun ThmProves : AxThmGoalStmnt -> ThmStmnt ;
lin ThmProves thm = {s = ProveThat ++ thm.s } ;
```

Note that, the assumptions in theorem (§7.2.2.2 on page 173) and the rest of theorem statements (§7.2.2.3 on page 173) are usable as axiom statements without any changes. But we must make an intermediate category, let us name it AxThmRestStmnt. So for all the functions defined in §7.2.2.2 and §7.2.2.3, we must change the output category from ThmStmnt to AxThmRestStmnt. Once it is done, we define function which convert category AxThmRestStmnt to the category ThmStmnt for theorem statements.

For axiom statements we define functions which convert categories AxThmGoalStmnt and AxThmRestStmnt to the category AxStmnt as shown below:

```
cat AxiomStmnt ;
fun MkAxiom1 : AxThmGoalStmnt -> AxStmnt ;
fun MkAxiom2 : AxThmRestStmnt -> AxStmnt ;

lincat AxStmnt = {s : Str };
lin MkAxiom1 ax = ax ;
lin MkAxiom2 ax = ax ;
```

As already stated, in CLM axiom block is exactly the same as theorem block. So there implementations are also similar. Of course with an obvious difference: axiom block starts with “Axiom” but theorem block starts with “Theorem”.

7.4 Definition

In CLM, a definition is somehow similar to axiom and theorem. In the GF grammar, a definition is simply a list of definition statements DefStmnts which are separated by full-stop as shown bellow:

$\texttt{DefStmnt}_1$. [$\texttt{DefStmnt}_2$. ... $\texttt{DefStmnt}_n$.]

As usual it is defined as shown below:

```
cat Definition ; DefStmnt ; DefStmnts
fun MKDef : DefStmnts -> Definition ;
lin MkDef defst = {s = "Definition." ++ defst.s };
```

Where the list `DefStmnts` is formed by the category `DefStmnt` (definition statement). It is is defined as followed:

Conditional Statements For conditional statement we support following patterns:

1. "if $\texttt{Statement}_1$[, $\texttt{Statement}_2$, ... , (and | or) $\texttt{Statement}_n$] then [we define] `Statement`"

   It is similar to the conditional statement (`IfthenStmnt`) (cf. §6.3.7 on page 146) with an exception of having only one `Statement` after the key phrase "then [we define]".

   Following are a few examples,

   "if $x > 0$ and $y > 0$, then we define x and y to be positive",
   "if $x > 0$ then x is positive",
   "if an integer n is divisible by 2 then it is even", etc.
   (the similar examples as the last two can be seen in figure 3.2 on page 33.)

   We define it as followed:

```
fun MkDefCond1 : Statements -> Statement -> DefStmnt ;
lin MkDefCond1 conds consq = {s = "if" ++ conds.s ++ optComma ++
                   "then" ++ variants{["we define"]; ""} ++ consq.s} ;
```

2. If we flip categories of the above categories:
   [we define] `Statement` if $\texttt{Statement}_1$[, $\texttt{Statement}_2$, ... , (and | or) $\texttt{Statement}_n$].

   For example,

   "we define x and y to be positive if $x > 0$ and $y > 0$",
   "x is positive if $x > 0$",
   "we define an integer n to be even if it is divisible by 2", etc.

   We only give its abstract syntax below:

```
fun MkDefCond2 : Statement -> Statements -> DefStmnt ;
```

   Recall that it is important to define this flip of categories as a new rule. It is because we need this order in the host system MathNat to solve the anaphoric resolution.

3. [we define] `Statement` (iff | only if | if and only if | if) $\texttt{Statement}_1$[, $\texttt{Statement}_2$, . . . , (and | or) $\texttt{Statement}_n$].

   For example,

   "we define x and y to be positive iff $x > 0$ and $y > 0$",
   "x is positive only if $x > 0$",
   "an integer n is even if and only if it is divisible by 2", etc.

We define it as followed:

```
fun MkDefCond3 : Statement -> Statements -> DefStmnt ;
lin MkDefCond3 consq conds = {s = ["we define"] ++ consq.s ++ iff ++ conds.s} ;
oper iff : Str = variants{"iff"; ["if and only if"]; "only if"} ;
```

Assumptions Assumptions in definitions are the same as in theorems and proofs (for instance, see §7.2.1.2 on page 159). A few examples are given in the next heading.

Define Statement It is formed by the `Statement` and the optional key phrase "we define".

A few examples containing assumptions and define statements are following:

"let x be an integer. assume that $x > 0$. we define x to be positive.",
"let m and n be arbitrary integers with a condition that $m > 0$. Then n is said to be divisible by m if there is a number q, such that $n = q * m$.",
"let n be an even integer. Then we define it to be divisible by 2", etc.

It is important to note that the last statement in a definition must be a conditional or a define statement.

7.5 Conventions

A controlled language simplifies the interpretation of its linguistics features by adopting some conventions. In the course of the last few chapters we have seen that CLM is no exception. It follows conventions for resolving anaphora and differentiating between collective and distributive readings. We refer to §2.4, §2.5.2 and §8.3 for an account.

We have also seen some of the conventions that we define to remove ambiguity. In the course of describing CLM grammar at macro and micro level, we have already discussed some of these conventions. we now simply combined them below.

However, we consider these conventions as temporary hacks until we connect MathNat to the proof assistant(s), to get type information (or boolean feedback in case of untyped proof assistant such as Mizar[Trybulec *et al.* 1973]). More details are given in §9.2.1 on page 234.

7.5.1 Ambiguity

We combine different kinds of propositions and statements in §6.3.6 by forming the category `Statement`. Many categories which form the category `Statement` allows conjunctions and disjunctions. These are three examples:

x is arbitrary, positive and even.
y is a multiple of 2, 4 and 8.
There are x and y such that $(x * a + y * p = 1)$ or $(x * a * b + y * p * b = b)$.

Then we define a category `Statements` which is a non-empty list of `Statement` in §6.3.6.1. It allows to write the following pattern for a mathematical sentence to solve the ambiguity propositions and statements:

Statement_1, Statement_2, ... (, and | , or) Statement_n.

Which we interpret as following:

$$\underbrace{\text{Statement}_1,\ \underbrace{\text{Statement}_2,\ \underbrace{\ldots\ (\text{ , and } \mid \text{ , or) Statement}_n.}}}$$

An elaborate example is the following in which we repeat the same statement three times:

x is arbitrary, positive and even **,**
y is a multiple of 2, 4 and 8 **, and**
there are x and y such that $(x * a + y * p = 1)$ or $(x * a * b + y * p * b = b)$.
('or' is interpreted as implication as discussed in §3.3.3.3 on page 45)

When we use the list `Statements` in a theorem statement (`ThmStmnt`) or in a proof statement (`PrfStmnt`, let us say as an assumption, our example becomes the following:

We assume that x is arbitrary, positive and even **,**
y is a multiple of 2, 4 and 8 **, and**
there are x and y such that $(x * a + y * p = 1)$ or $(x * a * b + y * p * b = b)$.

But `ThmStmnt` in theorem block and `PrfStmnt` in proof block, may be a complex statement as shown below:

$\texttt{ThmStmnt}_k \equiv \texttt{ThmStmnt}_{k_1}$; and $\texttt{ThmStmnt}_{k_2}$; and ... ; and $\texttt{Thmtmnt}_{k_n}$
$\texttt{PrfStmnt}_k \equiv \texttt{PrfStmnt}_{k_1}$; and $\texttt{PrfStmnt}_{k_2}$; and ... ; and $\texttt{Prftmnt}_{k_n}$

With the following interpretation:

$$\underbrace{\texttt{PrfStmnt}_{k_1};\text{ and }\underbrace{\texttt{PrfStmnt}_{k_2};\text{ and }\underbrace{\ldots;\text{ and }\texttt{PrfStmnt}_{k_n}.}}}$$

Now our example become the following, if we do repetition:

We assume that x is arbitrary, positive and even **,**
y is a multiple of 2, 4 and 8 , and
there are x and y such that $(x * a + y * p = 1)$ or $(x * a * b + y * p * b = b)$ **; and therefore, we conclude that** x is arbitrary, positive and even **,**
y is a multiple of 2, 4 and 8 , and
there are x and y such that $(x * a + y * p = 1)$ or $(x * a * b + y * p * b = b)$.

Consider now the real example, which is ambiguous but with conventions we are able to diffuse the ambiguity:

> **If** $\neg p|a$ then gcd$(a,p) = 1$**; and** therefore, by theorem 24, there are x and y such that $xa + yp = 1$ or $xab + ypb = b$.

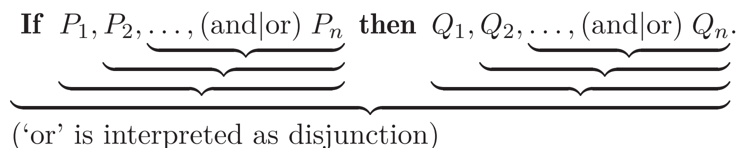
And now the same example but having different interpretation because of (**, or**) at the end:

> **If** $\neg p|a$ then gcd$(a,p) = 1$**; and** therefore, by theorem 24, there are x and y such that $xa + yp = 1$ **, or** $xab + ypb = b$.

Also recall that the conditionals of the form:

If $P_1, P_2, \ldots,$ (and|or) P_n **then** $Q_1, Q_2, \ldots,$ (and|or) Q_n.

are always interpreted as:

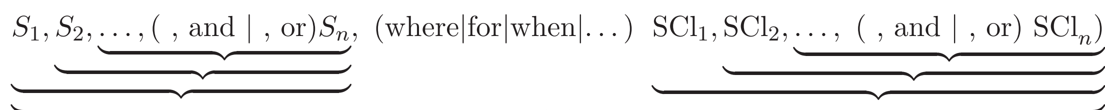


(‘or’ is interpreted as disjunction)

Finally, as regards the attachments (subordinate clauses on page 158, abbreviated as `SCl` below), we support the following pattern for a statement:

$S_1, S_2, \ldots,$ (, and | , or)S_n, (where|for|when|...) SCl$_1$, SCl$_2, \ldots,$ (, and | , or) SCl$_n$.

As usual, we interpret it as follows. This convention applies to all lists, the ones we have mentioned and those we have not.



Chapter 8

The Host System MathNat

Contents

8.1 Introduction . . . **179**
8.1.1 The Overall Picture . . . 180
8.1.2 The CLM Grammar in Haskell . . . 181
8.2 Semantic Checks . . . **182**
8.2.1 List of expressions and Quantity . . . 184
8.2.2 Pronoun and Quantity . . . 185
8.2.3 Collective properties . . . 185
8.2.4 List of expressions or Pronouns with Noun Adjunct . . . 187
8.2.5 Statements with Property Arbitrary . . . 188
8.2.6 Quantity and List of Expressions . . . 188
8.2.7 Expression, Equation and Symbolic Formula . . . 189
8.3 Discourse, Linguistic Features and Others . . . **189**
8.3.1 Propositions . . . 191
8.3.2 Equation with Reference . . . 200
8.3.3 Existential Statements . . . 202
8.3.4 Statement and its lists . . . 207
8.3.5 Conditional Statement . . . 208
8.3.6 Justifications . . . 210
8.3.7 The Macro Level Grammar . . . 213
8.3.8 Theorem . . . 213
8.3.9 Proof . . . 217
8.3.10 Theorem and its Proof . . . 220
8.3.11 Case Analysis for Proof by Cases . . . 221
8.3.12 The rest of blocks, categories, statements and rules . . . 232

8.1 Introduction

In the course of this thesis, we have referred to the host system MathNat several times. In this chapter we will describe it in detail.

GF grammar can be compiled into the low-level code called **P**ortable **G**rammar **F**ormat (PGF). The GF grammar in PGF format plays the same role as JVM byte code plays for Java. It means that the GF grammar could be used in other general purpose programming languages such as Haskell [Marlow 2010], Java, etc, with the help of GF runtime system.

GF provides an API to work on PGF grammar code (the API for Haskell is the most up-to-date). For instance, rendering functions such as `linearize` and `parse` are made available as ordinary Haskell functions. With the function `parse`, GF translates the abstract syntax trees of the grammar into Haskell data objects, where the abstract syntax is translated to an algebraic data type. It is beyond the scope of this work to give an account of Haskell. We refer to [Marlow 2010] and Haskell homepage (`http://haskell.org/`) for further details.

As obvious from the introduction, the host system MathNat is a program written in Haskell. Another reason for choosing Haskell is this: Functional languages are well suited for symbolic or algebraic computation. Haskell provides rich functionality such as higher order functions, pattern matching, type checking, etc. For instance, we need to go through the algebraic data types produced by the GF abstract syntax, filter them, apply different functionalities, and finally, build different tree or forest structures.

Regarding the LBNF grammar for symbolic mathematics, note that the BNF Converter tool is a compiler front end, which was created as a "spin-off of GF, customizing a subset of GF to combine with standard compiler tools" [Forsberg 2007, p. 51]. For it, the compiler's back-end could be written in Haskell. Therefore, along with GF grammar, we also call the grammar for symbolic mathematics as Haskell data objects in the host system MathNat.

8.1.1 The Overall Picture

On one hand we have CLM document which allows to write textual axioms, definitions, theorems and proofs using the categories `Axiom`, `Definition` and `ThmPrf`. As mentioned in Chapter 7, these categories are blocks containing the list of statements, having no axiom, definition, theorem and proof semantics. On the other hand, we have MathAbs document which allows to write formal axioms, definitions, theorems and proofs, having a clear semantics as given in §4.4.

Our overall objective is to take CLM documents and translate it into MathAbs document. In the process of doing so, we apply procedures such as:

1. Applying semantic checks on sentences
2. Building context
3. Applying linguistic features
4. Discourse building for each block (theorem, proof, etc)
5. Translating CLM document to MathAbs document

When the mathematical text from CLM grammar is parsed, a list of sentence level **A**bstract **S**yntax **T**rees (ASTs) are produced. By sentence level, we mean that an AST is produced for each sentence which is separated by full stop. We recognize each AST by pattern matching on algebraic data types and apply the above mentioned procedures one by one.

These procedures are not applied on the whole abstract syntax tree of the text at once. Instead, these are applied on each of the sentences one by one. Even in each sentence, these are applied on each statement recursively, forming a combined result (i.e. the MathAbs rules) for sentence (recall that a sentence may be formed by a list of statements). The result is updated with each sentence that appears next in the text. This procedure is terminated when all the sentences are parsed and there is nothing left.

8.1.2 The CLM Grammar in Haskell

As mentioned in Chapter 7, the categories `ThmPrf`, `Axiom`, `Definition` are blocks containing the list of statements belonging to theorem, proof, axiom and definition respectively. In CLM document, we allow to describe a list of theorems and proofs, list of axioms and list of definitions (with a simple LBNF grammar). These lists are given to the host system MathNat, where we apply specialized functions for context building, block structure building, linguistic support and the translation to MathAbs. These procedures are somewhat interleaved. Therefore, we'll explain them all in one (wherever possible) to save time, space and effort (because in terms of programing and labor, going through the abstract syntax again and again is expensive).

Let us concentrate on the theorem and proof block. It is formed by the category `ThmPrf` as shown below:

```
cat ThmPrf ;
fun MkThmPrf : ThmStmnts -> PrfStmnts -> ThmPrf ;
```

In Haskell, it becomes as following:

```
data GThmPrf = GMkThmPrf GThmStmnts GPrfStmnts
```

First, the letter '`G`' is prefixed to all constructs. The name of the category is assigned to this new data type (i.e. `GThmPrf`). In contrast, the name of the function becomes the name of its constructor (`GMkThmPrf`), where `GThmStmnts` and `GPrfStmnts` are other data types.

Similarly, the category `PrfStmnts` is defined as shown below:

```
1 cat PrfStmnts ;
2 fun BasePrfStmnts : PrfStmnt -> PrfStmnts ;
3     ConsPrfStmnts : PrfStmnt -> PrfStmnts -> PrfStmnts ;
4     ConjPrfStmnts : PrfStmnt -> PrfStmnts -> PrfStmnts ;
```

In Haskell, it becomes the following. Again, on lines 2–4 below, note that the names of the functions become the names of constructors:

```
1 data GPrfStmnts =
2    GBasePrfStmnts GPrfStmnt
3  | GConjPrfStmnts GPrfStmnt GPrfStmnts
4  | GConsPrfStmnts GPrfStmnt GPrfStmnts
```

Similarly, the category for proof statement `PrfStmnt` is formed by several functions. We give some of them:

```
1 cat PrfStmnt ;
2 fun MkRestmnt  : Statements    -> Subordinate -> PrfStmnt ;
3     MkERestmnt : EStatements   -> Subordinate -> PrfStmnt ;
4     MkLet      : LetStatements -> Subordinate -> PrfStmnt ;
5     MkAssume   : Statements    -> Subordinate -> PrfStmnt ;
6     ...
```

Of course, it becomes the following in Haskell:

```
1  data GPrfStmnt =
2    | GMkRestmnt  GStatements    GSubordinate
3    | GMkERestmnt GEStatements   GSubordinate
4    | GMkLet      GLetStatements GSubordinate
5    | GMkAssume   GStatements    GSubordinate
6    ...
```

We now take only one constructor from it (for instance, `GMkAssume` given on line 5 above) and try to go deeper in the data types `GStatements` and `GSubordinate`. As mentioned in §7.2.1.1 on page 158, the category `Subordinate` is simply the list `Statements`, attached with a bunch of key phrases, defined as shown below, followed by the corresponding Haskell data type:

```
cat Subordinate ;
fun MkSubord     : Statements -> Subordinate ;
    EmptySubord :               Subordinate ;

data GSubordinate = GMkSubord GStatements | GEmptySubord
```

The data type for `GStatements` is a sophisticated list (cf. §6.3.6.1 on page 144) having `GStatement` as elements. In Haskell, `GStatement` becomes the following (cf. §6.3.6 on page 143):

```
1  data Statement =
2     GMkPropStmnt  GProposition
3   | GMkExistStmnt GPropExist
4   | GMkRelStmnt   GPropRel
5   | GMkEqRefStmnt GEqWithRef
```

We end this section here, but we go deeper in the construct `GProposition` in the next section. We'll also use these data objects in §8.3 for discourse building, applying linguistic features and other procedures.

8.2 Semantic Checks

Most of the semantic checks are applied on various types of statements such as propositions, existential statements, relational propositions and equations (lines 2–5 above respectively). We define these semantic checks in the subsequent sections.

But first, we show how they are plugged into the algebraic data type, let us say, on proof statements (i.e. `GPrfStmnt`, defined above). We define the function `chkPrfStmnt` (and similarly `chkThmStmnt` for theorem statement, `chkAxStmnt` for axiom statement and `chkDefStmnt` for definition statement). It takes `GPrfStmnt`, the sentence number (to return it in case of an error), and returns an error or nothing. We select its constructors one by one by pattern matching:

```
1 Function name: chkPrfStmnt
2 Input:  GPrfStmnt labeled as prf, Position labeled as sentence_no
3 Output: Error or nothing
```

```
4  Procedure:
5   pattern matching on prf
6   IF prf is (GMkRestmnt stmnts subord)        //constructor of GPrfStmnt
7     THEN apply function chkStmnts stmnts
8          apply function chkStmnts (get_stmnts_from_subordinate subord)
9
10  ELSE IF prf is (GMkERestmnt estmnts subord) //constructor of GPrfStmnt
11    THEN apply function chkEStmnts stmnts
12         apply function chkStmnts (get_stmnts_from_subordinate subord)
13
14  ELSE IF ....
15  ....
```

We omit its corresponding function (i.e. `chkPrfStmnts`) for proof block. It is because it is simply made by applying the function `chkPrfStmnt` on `GPrfStmnts` using a higher order 'map' function.

Similarly, we also omit the implementation of function `chkThmStmnts`. Because, it also simply applies the function `chkThmStmnt` on `GThmStmnts` using a higher order map function. The function `chkThmStmnt`, is defined below, which of course takes `GStatement` as input:

```
1  Function name: chkStmnt
2  Input:  GStatement labeled as stmnt, Position labeled as sentence_no
3  Output: Error or nothing
4  Procedure:
5   pattern matching on stmnt
6   IF stmnt is (GMkPropStmnt prop)         //constructor of GStatement
7     THEN apply function (chkProp prop)
8
9   ELSE IF stmnt is (GMkExistStmnt exist)  //constructor of GStatement
10    THEN apply function (chExistStmnt exist)
11
12  ELSE IF ....
13    THEN ....
14
15  ELSE IF stmnt is (MkEqRefStmnt eq)
16    THEN apply semantic check (eq is an equation) //not an expression or anything else
```

For the sake of completeness, we finally define the `chkProp`, which takes `GProposition` as input. The data type `GProposition` represents simple propositions that are defined in §6.3.2 on page 124. It becomes the following data type in Haskell (we only present few for the moment, remaining will only be given if needed):

```
1  data GProposition =
2     GMkPosProp1 GSubject GQuant GProperties1 GType GAdjunctWith
3   | GMkNegProp1 GSubject GQuant GProperties1 GType
4   | GMkPosProp2 GSubject GProperties2 GAdjunctWith
5   ...
```

Whereas, `GSubject` is define as:

```
1  data GSubject = GMkExpsSubj    GExps
2                | GMkPronSubj    GPron
```

Finally, the function `chkProp`:

```
1  Function name: chkProp
2  Input:  GProposition labeled as prop, Position labeled as sentence_no
3  Output: Error or nothing
4  Procedure:
5   pattern matching on prop
6   IF prop is (GMkPosProp1 subj quant propts type adjnct) //constructor of GProposition
7    THEN
8      pattern matching on subj
9      IF subj is (GMkExpsSubj exps)
10      THEN
11       apply semantic check (all of elements in exps are expressions)
12       apply semantic check (agreement between exps and quants) //see coming subsection
13       apply all required semantic checks
14
15     ELSE IF subj is ...
16     ....
17
18  ELSE IF prop is (....)                                //constructor of GProposition
19   THEN ...
20  ....
```

All the semantic checks mentioned in Chapters 6 and 7 are plugged-in as shown above. Also note that all these semantic checks are trivial pieces of code, offering no insight except one: this shows how we actually apply some of the constraints in the host system MathNat which might be hard to apply directly in the CLM grammar.

8.2.1 List of expressions and Quantity

In most of the rules for various statements (i.e. propositions, existential statements, etc), we have to make an agreement as semantic check between number of elements in the list of expression (`GExps`) and the quantity (`GQuant`). It is specifically stated in §6.3.2.1 on page 126. This check allows us to reject statements of the following form:

> *Assume that x and y are **three** positive numbers.

The number three in above example belongs to `GQuant`, which is roughly defined as:

```
data GQuant = GOne | GTwo | GThree | ... | GEmpty
```

Let us first define two helper function:

- `quant2num` which takes `GQuant` and returns integer:
  `GSome -> -1 (a hack; see line 8 in the code below), GEmpty -> 0, GOne -> 1, GTwo -> 2, ...`
- `exps2num` takes the list `GExps` and returns its length as integer

Finally for agreement, we define function `agrmntExpsQuant`. It takes `GExps`, `GQuant`, position (i.e. sentence number from the mathematical texts) and returns nothing if they are in agreement. Otherwise it returns an error message mentioning the sentence number. It can be described with the following Haskell like pseudo-code:

```
1  Function name: agrmntExpsQuant
2  Input:  GExps as exps, GQuant as quant, Position as sentence_no
3  Output: Error or nothing
4  Procedure:
5     qn = quant2num quant
6     en = exps2num exps
7     IF (qn==en)              // length of GExps is equal to quantity
8        OR (qn==-1 and en>0)  // quantity is GSome
9        OR (qn==0  and en>=1) // quantity is GEmpty
10     THEN return nothing
11
12    ELSE return an appropriate error with sentence_no
```

8.2.2 Pronoun and Quantity

We also have to make an agreement as semantic check between pronoun (`GPron`) and the quantity (`GQuant`) (cf. §6.3.2.1 on page 126). This check allows us to reject statements as shown below. Note that pronouns are in agreement with types:

*It is **three** positive number. (‘it’ in agreement with ‘number’)
*They are **a** positive numbers. (‘they’ in agreement with ‘numbers’)

We first define the helper function `pron2num`. It takes a pronoun and returns an integer: (`GIt -> 1, GThey -> 2`).

For agreement, we define the following function. It takes a pronoun, quantity, line number and returns nothing if there is an agreement. Otherwise it returns an appropriate error message with the line number of that sentence.

```
1  Function name: agrmntPronQuant
2  Input:  GPron as pron, GQuant as quant, Position as sentence_no
3  Output: Error or nothing
4  Procedure:
5     qn = quant2num quant
6     pn = pron2num  pron
7     IF   (qn==pn)
8       OR (qn>1  and pn==2) // Quantity not (GEmpty|GOne) and pronoun is ‘They’
9       OR (qn==0 and pn>=1) // Quantity is GEmpty and any pronoun
10     THEN return nothing
11
12    ELSE return an appropriate error with sentence_no
```

8.2.3 Collective properties

As mentioned in §6.3.2.1 on page 129, we need to reject those statements which applies collective properties on a single variable (or on pronoun ‘it’), such as following:

*Let x be equal.
*Assume that it is equal.

We first need to tag each property being collective or distributive. We do it in Haskell with data type (`PropertType`) and function (`whichPropert`). Note a convention on line 6 below. That is, instead of throwing an error, we set all the properties as `Distributive` (with dots ...), which are not tagged manually in the code.

```
1  data PropertType = Collective | Distributive
2
3 Function name: whichPropert
4 Input:  GProperty as p
5 Output: PropertType
6 Procedure:
7  CASE p of
8    GEq, GCoprime                          -> Collective
9    GPositive, GNegative, GEven, GOdd, ... -> Distributive
```

We have many kinds of lists for category `Property` (to be precise: `Properties1`, `Properties2`, `EProperties`). The difference between them is syntactical. Since the syntax phase is covered with GF, here, we convert them to the usual list such as `[Property]` (of course keeping the information if it was list separated by conjunction or disjunction, etc). In the following pseudo-code, we assume that such conversion is already done.

```
Function name: chkExpsProps
Input:  GExps as exps, [GProperty] as props, Position as sentence_no
Output: Error or nothing
Procedure:
 IF the list props is empty then return nothing else do following:
 IF ((length exps)>1) then return nothing else do following:
  rest_of_props = props - first element
  proptype = whichPropert (take_first_element props)
  CASE proptype of
    Distributive -> chkExpsProps exps rest_of_props //recursive call
    Collective   -> return error with sentence_no and the property which caused it
                             (i.e. take_first_element from the list rest_of_props)
```

We need another function for pronouns:

```
Function name: chkPronProps
Input:  GPron as pron, [GProperty] as props, Position as sentence_no
Output: Error or nothing
Procedure:
 IF the list props is empty then return nothing else do following:
 IF pron is GThey then return nothing else do following:
  rest_of_props = props - first element
  proptype = whichPropert (take_first_element props)
  CASE proptype of
    Distributive -> chkPronProps pron rest_of_props //recursive call
    Collective   -> return error with sentence_no and the property which caused it
                             (i.e. take_first_element from the list rest_of_props)
```

Of course we can combine both functions for a higher category `GSubject`. Recall that is define as:

```
 data GSubject = GMkExpsSubj GExps | GMkPronSubj GPron
```

We can access `GExp` and `GPron` from it by pattern matching, as shown in the following pseudo-code:

```
Function name: chkExpsProps
Input:  GSubject as subj, [GProperty] as props, Position as sentence_no
Output: Error or nothing
Procedure:
 pattern matching on subj
 IF subj is (GMkExpsSubj exps) then chkExpsProps exps props sentence_no
 ELSE IF subj is (GMkPronSubj pron) then chkPronProps pron props sentence_no
 ELSE return nothing
```

8.2.4 List of expressions or Pronouns with Noun Adjunct

We have described noun adjuncts in §6.3.2.6 on page 134. In Haskell it becomes the following data type:

```
1  data GAdjunctWith =
2     GMkCmnRelAWith GQuant1 GRelation
3   | GMkEqAWith GEq | GEmptyAdj
```

Of course the noun adjuncts attach to various propositions, which contains a subject. The function described on line 2 corresponds to bullet 1 in §6.3.2.6, must attach only with the proposition having at least 2 expressions or a plural pronoun as a subject. This check allows to reject the ill-formed statements such as:

> *a is a non-zero integer with no **common** factor,
> *It is a non-zero integer with three **common** elements, etc.

First, we define a function `chkCmnExpsAdjnct` which takes `GExps` and noun adjunct as parameters:

```
Function name: chkCmnExpsAdjnct
Input:  GExps as exps, GAdjunctWith as adjnct, Position as sentence_no
Output: Error or nothing
Procedure:
 pattern matching on adjnct
 IF adjnct is (GMkCmnRelAWith q rel) then do following:
    IF (length exps)==1 then return an appropriate error with sentence_no
    ELSE return nothing
 ELSE return nothing
```

Now we define a similar function for pronouns:

```
Function name: chkCmnPronAdjnct
Input:  GPron as pron, GAdjunctWith as adjnct, Position as sentence_no
Output: Error or nothing
Procedure:
 pattern matching on adjnct
 IF adjnct is (GMkCmnRelAWith q rel) then do following:
    IF (pron is GIt) then return an appropriate error with sentence_no
    ELSE return nothing
 ELSE return nothing
```

8.2.5 Statements with Property Arbitrary

As described in §6.3.3.1 on page 138, when the property "arbitrary" occurs in goals and conclusions produced by the simple propositions and existential statements, we reject them. Some of these need to be rejected statement are:

> *We conclude that there exists an arbitrary integer x such that $x > 0$.
> *We conclude that x is an arbitrary positive integer.
> *Prove that x is an arbitrary positive integer.

Defining the function which does it is quite easy and straight forward, as shown in the following pseudo-code:

```
Function name: rejectArbitrary
Input:  GProperties1 as props, Position as sentence_no
Output: Error or nothing
Procedure:
 IF props is empty list then return nothing
 ELSE
  prop = take_first_element props
  pattern matching on prop
  IF prop is GArbitrary then return appropriate error with sentence_no
  ELSE recursive call to rejectArbitrary (props without prop) sentence_no
```

8.2.6 Quantity and List of Expressions

We need an agreement between quantity and the list of expressions for relational statements given in §6.3.4.2 on page 141. It is very similar to the check given in §8.2.1 on page 184, with an exception of having slightly different quantity `Quant1` (cf. §6.3.1.4 on page 115) and one extra condition. It is defined in Haskell as shown below:

```
data GQuant1 = GNo | GOneQ1 | GTwo | GThree | ... | GThen | GSome
```

There are two conditions that we must fulfill: (1) the agreement between `Quant1` and `Exps` and (2) `Exps` must have more than one elements.

So let us define a helping function `quant1toNum` which takes `GQuant1` and returns integer: `GNo -> 0, GOneQ1 -> 1, GTwo -> 2, ..., GSome -> -1`. Recall that we already have define a function `exps2num` which takes the list `GExps` and returns its length as integer.

Finally we define the main function `agrmntQuant1Exps`. It takes `GQuant1`, `GExps`, position (i.e. sentence number from the mathematical texts) and returns nothing if they are in agreement according to our conditions. Otherwise its returns an error message mentioning the line number. It can be described with the following Haskell like pseudo-code.

```
1 Function name: agrmntQuant1Exps
2 Input:  GQuant1 as quant, GExps as exps, Position as sentence_no
3 Output: Error or nothing
4 Procedure:
5    qn = quant1toNum quant
6    en = exps2num exps
7    IF (en>1)                 // length of GExps is greater than 1
```

```
8      THEN do following:
9       IF   (qn==en)           // length of GExps is equal to quantity
10        OR (qn==-1 and en>0)  // quantity is GSome
11        OR (qn==0  and en>=1) // quantity is "no"
12       THEN return nothing
13    ELSE return an appropriate error with sentence_no
```

Fundamentally, the function remains the same as the function `agrmntExpsQuant`, except one extra condition given on line 5 below. So we can re-factor it easily and use function `agrmntExpsQuant` inside it. An example of the condition applied on line 9 is the following sentence:

> n and m have no common divisors d and e.

8.2.7 Expression, Equation and Symbolic Formula

As we have seen that symbolic expressions (`Exp`), equations (`Eq`) and symbolic formulas (`SymbFormula`, cf. §7.2.2.1 on page 170) appear in many rules. These categories are simple string records in the GF grammar and therefore, any ⟨*Formula*⟩ may be given. We need to assure with semantic checks, that the symbolic mathematics given to these categories respects their specifications. We only define this check for expressions and leave the other two checks because of their similarly in the procedure.

The symbolic expression `Exp` is mentioned in the LBNF grammar by `Formula8`, `Formula9`, . . . , `Formula13` (lines 27–47 in figure 6.2 on page 109). Note that these labels (i.e. `Formula8`, ..., `Formula13`) are not visible in the abstract syntax. But we still can recognize them by pattern matching on the the constructs that use them. For instance, the constructs `SUnion` and `SIntersec` on lines 27–28 in figure 6.2 forms `Formula8`.

The data type for GExp in Haskell is given on line 1 below. We define a function `transExp`, in the rest of the code, which translates this string containing expression to ⟨*Formula*⟩ using BNFC parser.

```
1 data GExp = GMkExp String
2
3 Function name: transExp
4 Input:  GExp as exp, Position as sentence_no
5 Output: Formula
6   pattern matching on exp
7   IF exp is (GMkExp (GString s)) then apply bnfc parsing on s
8   IF parse fails then return an appropriate error
9    ELSE save the result in formula
10   pattern matching on formula recursively
11   IF formula is one of (Formula8, Formula9,..., Formula13)
12     THEN ok ELSE return an appropriate error with sentence_no
```

There are many other semantic checks which we apply on CLM (as mentioned in chapters 6 and 7. However, it does not seems necessary to define their trivial code here, and therefore, we omit them.

8.3 Discourse, Linguistic Features and Others

An informal account on how we build discourse from the mathematical texts and support linguistic features is given in §2.5.2. We now describe the procedures for discourse

building, linguistic features and others in concrete terms. The format is somewhat similar to the way we presented CLM grammar in Chapters 6 and 7. We will go through the abstract syntax of CLM as a whole and describe these procedures for every category and function. In our opinion this section must be read along with §2.5.2 on page 19. We now fix some notations:

Context: `Context` is a 3-tuple `(CV, ST, CE)` where:

`CV` is a list of 6-tuple (`Sn`, List of symbolic objects $[obj_1, ..., obj_n]$, Number (1 for singular, n for plural), Quantification, How declared, Object(s) type). In `CV`, we record necessary information such as every occurrence of symbolic expressions, left hand sides of equations, pronouns and references.

1. `Sn` or S_n is the sentence number in the mathematical texts.
2. Symbolic object `obj` is a symbolic expression which we store as anaphoric referent.
3. Quantification is defined as:

```
data VarQuant = QUniv | QExists | QLet | QNo | QUndecided
```

   Where, `QUniv` stands for Universal (i.e. $\forall x(...)$), `QExists` stands for Existential (i.e. $\exists x(...)$), `QLet` stands for the rule let (i.e. let x : ...)), `QNo` is used for number and `QUndecided` for those expressions which mix variables having `QUniv`, `QExists` and `QLet` quantification (see examples in appendix A on page 239).

4. *How declared* could have two values: `Explicit` for explicitly defined variables and `Implicit` for those which are implicitly defined variables by the system. For instance, "assume that $x+y$ is a positive integer". If x, y are not defined before (i.e. they are not in the context) then both are implicitly defined by the system (i.e. let x, y : `NoType`). Whereas, $x + y$ is explicitly defined because of this statement. It is defined as following for this chapter:

```
data HowDecl = Explicit | Implicit
```

5. Of course object type is the type of the object. For instance, x, y in the above bullet have type "`NoType`". Whereas, $x + y$ has type "Integer".

`ST` is a list of 3-tuple (`Sn`, logical formula, `StType`).
We use it to save information regarding logical formulas for each sentence, equivalent to its MathAbs. This information is useful in many ways as described in §2.5.2.2 on page 22 and bullet §1 of §4.6 on page 74.

1. `StType` is statement type (whether it is an assumption, deduction, goal or justification) as defined below:

```
data StType = Asm | Ddc | Prv | Justif
```

We need this information to be able to use the same function for different types of statements. For instance, `StType` is
`Asm` for a statement such as "suppose that x is a positive even integer",
`Prv` for a statement such as "prove that x is not a positive even integer",
`Ddc` for a statement such as "x and y are two integers" and
`Justif` for a statement such as "... because $\sqrt{2}$ is not an even integer".

`CE` is a list of 3-tuple (`Sn`, equation, reference).
We use it to save information regarding equations and references (cf. §2.5.2.3 on page 22).

Micro Level CLM Grammar

We start this section by describing procedures for "high level constructs" of the micro level grammar mentioned between §6.3.2 to §6.3.9.

8.3.1 Propositions

As already mentioned briefly on page 183, the simple propositions (i.e. category `Proposition`) that are defined in §6.3.2 on page 124 becomes `GProposition` in Haskell, and its rules become constructors, as shown below:

```
1  data GProposition =
2     GMkPosProp1 GSubject GQuant GProperties1 GType GAdjunctWith
3   | GMkNegProp1 GSubject GQuant GProperties1 GType
4   | GMkPosProp2 GSubject GProperties2 GAdjunctWith
5   | GMkNegProp2 GSubject GProperties2
6   | GMkPosProp3 GSubject GEProperties GAdjunctWith
7   | GMkNegProp3 GSubject GEProperties
8   | GMkPosProp4 GSubject GRelation GExp
9   | GMkNegProp4 GSubject GRelation GExp
10  | GMkPosProp5 GSubject GRel2 GExp
11  | GMkNegProp5 GSubject GRel2 GExp
```

Recall that:

- `GSubject` is either a list of expressions, i.e. `GExps` e.g. x, y and z, $2 * x + y$, etc, or pronoun (`GPron`) e.g. it and they. (cf. §6.3.1.2 on page 114)
- `GQuant` is one, two, three, etc. (cf. §6.3.1.4 on page 115)
- `GProperties1`, `GProperties2` and `GEProperties` are lists of properties. (cf. 6.3.1.7 on page 119). e.g.:
  `GProperties1`: "positive even", "arbitrary negative odd", etc.
  `GProperties2`: "positive, even (and | or) coprime", etc.
  `GEProperties`: "**either** x is **positive, even or coprime**", etc.
- `GType` is integer, rational, prime, etc. (cf. §6.3.1.5 on page 117)
- `GAdjunctWith` is an optional noun adjunct. We have three rules for it. e.g. "having a common element", "with $x > 0$" or *nothing*, etc. (cf. §6.3.2.6 on page 134)

- `GRelation` is element, factor, square, etc. (§6.3.1.8 on page 123)

We define a skeleton function `mkProp` for `GProposition`. It allows us to build context, apply linguistic features and translate the Haskell data objects for CLM to something which can easily be converted into MathAbs on the macro level. We give its pseudo-code below, followed explanation in subsequent subsections:

(Note that, we simplify the input of these functions by not mentioning the sentence number and statement type `StType` in input parameters. But they should be considered to be accessible with labels `sn` and `stype` throughout this chapter.)

```
1  Function name: mkProp
2  Input:  GProposition as prop, Context as cntxt
3  Output: Tuple (Context, [Assignment], [Assignment])
4  Procedure:
5   pattern matching on prop
6   IF prop is (GMkPosProp1 subj quant props1 tp adjnctW)
7    THEN (mkProp1 subj quant props1 tp adjnctW PPosit cntxt)
8
9   ELSE IF prop is (GMkNegProp1 subj quant props1 tp)
10   THEN (mkProp1 subj quant props1 tp GEmptyAdj PNeg cntxt)
11
12  ELSE IF prop is (GMkPosProp2 subj props2 adjnctW)
13   THEN (mkProp2 subj props2 adjnctW PPosit cntxt)
14
15  ....
16
17  ELSE IF prop is (GMkPosProp4 subj rel exp)
18   THEN (mkProp4 subj rel PPosit cntxt)
19
20  ELSE IF prop is (GMkNegProp4 subj rel exp)
21   THEN (mkProp4 subj rel PNeg cntxt)
22
23  ....
```

As we can see on lines 2–3 above, it takes `GProposition`, `Context` and returns a 3–tuple (`Context, [Assignment], [Assignment]`). The second and third element of this tuple are lists of `Assignment` which is a part of MathAbs definition given in §8.3.9 on page 217:

```
LetF.    Assignment ::= "let" [Ident] ":" Type ;
AssumeF. Assignment ::= "assume" Formula ;
```

As we can see, `LetF` allows us to declare variables. `Ident` used in it is simply a string. Of course `[Ident]` means the list of identifiers. `Type` corresponds to the linguistic types defined in §6.3.1.5 on page 117. Whereas, `AssumeF` allows to introduce hypothesis. The first assignment list in 3–tuple (`Context, [Assignment], [Assignment]`) stores those variables which are automatically declared. Whereas, the second assignment list stores explicitly declared variables as well as other formulas. We demonstrate its with an example:

Suppose that $x + y$ is a positive integer.
(assume that x and y are not declared, in other words, they are not in the context)
First `[Assignment]` = [let x, y : NoType]
Second `[Assignment]` = [assume $(x + y : \mathbb{Z}) \wedge \text{positive}(x + y)$]

Also note that, most of the functions in the host system MathNat returns this output. Because with it, it is easy to distinguish between variable declarations (with constructor `LetF`) and assumptions (with constructor `AssumeF`). In case when we don't need this distinction (in some cases of negative statements, prove statements, deductions and justifications), we can easily extract formula from these constructs. See this happening in macro level grammar in §8.3.7 on page 213.

Finally, we access the first constructor of `GProposition` on line 6 above by pattern matching which is given in the next sub section, where `subj` is label for `GSubject`, `quant` is label for `GQuant`, `props1` is label for `GProperties1`, `tp` is label for `GType` and `adjnctW` is label for `GAdjunctWith`.

8.3.1.1 The first rule for propositions

It is represented by the constructors given on lines 2–3 on page 191 in the definition of data type `GProposition`. Here are few examples of this rule:

Assumptions:
Suppose that x is a positive even integer.
Let x and y be positive even integers.
Goals:
Prove that x is not a positive even integer.
Deduction:
x and y are two integers.
We conclude that $\sqrt{2}$ is not an even integer.

On lines 6–10 of function `mkProp` on page 192, we use function `mkProp1` for both positive and negative propositions covered by this rule. The description of these functions will be lengthy and therefore, one has to bear with us. We define it below:

```
1  Function name: mkProp1
2  Input: GSubject as subj, GQuant as quant, GProperties1 as props, GType as tp,
3        GAdjunctWith as adjnctW, Polarity as pol, Context as cntxt
4  Output: Tuple (Context, [Assignment], [Assignment])
5  Procedure:
6   pattern matching on subj
7   IF subj is (GMkExpsSubj exps)
8    THEN do following:
9     forms = mkExps2Forms exps
10    makeExpsTypeProps forms props tp adjnctW pol cntxt
11
12  ELSE IF subj is (GMkPronSubj pron)
13   THEN makePronTypeProps props tp adjnctW pol cntxt
```

As we see above, we do pattern matching to discover two cases of subject, shown in the subsequent paragraphs.

When subject is a list of expressions We are referring to lines 7–10 above. We first define function `mkExps2Forms` used on line 9 above. It translates the list of expressions coming for GF to the list of MathAbs formulas, as shown below:

```
Function name: mkExps2Forms
Input: GExps as exps
Output: [Formula]
Procedure:
 forms = translate list exps to list of formulas using BNFC parser
 apply semantic check that each element of forms is expression
 apply semantic check that all the elements of forms are unique
 report error with sn if any of these checks fail //sn = sentence number
 return forms
```

We now define the function `makeExpsTypeProps` below which is used above in function `mkProp1` on line 13.

```
1  Function name: makeExpsTypeProps
2  Input: [Formula] as forms, GQuant as quant, GProperties1 as props, GType as tp,
3        GAdjunctWith as adjnctW, Polarity as pol, Context as cntxt
4  Output: Tuple (Context, [Assignment], [Assignment])
5  Procedure:
6   pattern matching on adjnctW
7   IF adjnctW is GEmptyAdj //no adjunct
8    THEN (makeTypeProps forms props tp pol cntxt)
9
10  ELSE IF adjnctW is (GMkCmnRelAdjWith quantA rel) //e.g. no common factor
11   THEN (makeTypePropsCmnRel forms props tp quantA rel cntxt)
12
13  ELSE IF adjnctW is (GMkFunAdjWith eqwithRef) //e.g. with x>o
14   THEN (makeTypePropsFun forms props tp eqwithRef cntxt)
```

In the above function, we mainly cover three constructors of noun adjuncts described in §6.3.2.6 on page 134, and given below.

No Noun Adjunct: The first case is given on lines 7–8 above, when no nouns adjunct is given. Let us work on it first, and therefore, we now define function `makeTypeProps` given above:

```
1  Function name: makeTypeProps
2  Input: [Formula] as forms, GQuant as quant, GProperties1 as props, GType as tp,
3        GAdjunctWith as adjnctW, Polarity as pol, Context as cntxt
4  Output: Tuple (Context, [Assignment], [Assignment])
5  Procedure:
6   pattern matching on props
7   ps = translate GF list property props to Haskell list property //i.e [Property]
8   IF ps is empty  //no property is given
9    THEN mkTypeProps forms pol typ [] cntxt
10
11  ELSE do following:
12   IF (chkCollectiveProps forms ps)==True
13      THEN mkTypeProps forms pol typ ps cntxt
14   ELSE return an error that collective property must apply on at least two expressions
```

There are two cases in the above function: the first case when we have an empty list (`[]`) for properties (i.e. the case when no property is given); and the second case when properties are given. In it, on line 12, we make sure that if any of these properties are collective then it must be applied to more than one expression. Otherwise we return an error. Now, let us define `mkTypeProps` function used on line 10 above, for such statements ("@" in `cntxt@(cv,st,ce)` is used as alias. Recall the definition of Context from page 189):

```
1  Function name: mkTypeProps
2  Input: [Formula] as forms, GType as tp, [Property] as ps, Polarity as pol,
3        Context as cntxt@(cv,st,ce)
4  Output: Tuple (Context, [Assignment], [Assignment])
5  Procedure:
6    t = transType tp // translates GType to MathAbs type
7    (cv1, autoforms) = declSubVars forms cv
8    vars = extract all those elements of forms who are variables //e.g. x, y, z, etc
9    restforms = extract all the rest //e.g. x+y, x*y, sqrt(2), 4, etc
10
11   autoLet = mkLet autoforms NoType //simply makes "let autoforms:NoType"
12   asgnLet = search vars in cv1 for those already declared
13             IF not found THEN OK ELSE return a warning
14             in both cases compute (mkLet vars t) for asgnLet
15
16   (coll_ps, dist_ps) = separate collective and distributive properties from ps
17
18   cforms = FOR EACH i (1 to length coll_ps)
19             [pi(forms) | pi:coll_ps] //[equal(x,2*y),...]
20   cform  = fold cforms with conjunction //equal(x,2*y) AND ...
21
22   dforms = FOR EACH i (1 to length dist_ps)
23             FOR EACH j (1 to length forms)
24              [pi(fj) | pi:ps, fj:forms] //[even(x),even(2*y),positive(x),positive(2*y),...]
25   dform  = fold cforms with conjunction   //even(x) AND even(2*y) AND positive(x) AND...
26
27   formula = IF pol is PPosit THEN (conjunction of cform and dform)
28              ELSE negation of (conjunction of cform and dform)
29   asgnAssume = mkAssume formula //simply makes "assume formula"
30
31   cv2 = add in cv1 (sn, forms, length forms, QLet, Explicit, t)
32   st1 = add in st  (sn, formula, stype)
33   ce  = remain unchanged because no equation in this statement
34   cntxt1 = (cv2, st1, ce)
35   return(cntxt1, [autoLet], [asgnLet]++[asgnAssume]) //++ combines two lists
```

On line 7 above, we use function `declSubVars` (defined at the end of this section). It defines sub variables found in an expression given to be declared. For instance, when we have a statement such as:

"let x be an even integer"

There is no sub variable to consider and the MathAbs would:

"let $x : \mathbb{Z}$ assume $\text{even}(x)$"

(If x is not found in the context. Otherwise we'll give a warning as well along with this MathAbs). Note the context on lines 31–33. For this statement we'll add

the following entries in `CV` and `ST`, whereas `CE` remains unchanged (suppose that this statement also appears in proof block and its the second sentence of the theorem and proof):

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
2	x	1	`QLet`	`Explicit`	Integer

`CV`

$\mathbb{S}_n$	Logical Formula	Statement Type
...	...	...
2	$x : \mathbb{Z}$	Hypothesis

`ST`

But instead of x, if the superficial expressions $a + b$ and $2 * x + y^z$ are given:

> "Suppose that $a + b$ and $2 * x + y^z$ are even integer"
> (With "let" we only allow variables, not expressions. Therefore, we rephrase it.)

Then we need to consider sub expressions first (a, b and $2, x, y, z$). Suppose that a, z are already defined in the text before and therefore, they are available in the context. But others are not already defined. Then the function **`declSubVars`**, will return a 2-tuple: (1) context variable (`CV`) in which these undefined variables are added now, and (2) the list of these newly defined variables. The function **`mkTypeProps`** will use this list to define them in `Assignment`: "`let` b, x, y : `NoType`". After that the function **`mkTypeProps`** will do the rest of the job: "`assume` $(a+b : \mathbb{Z}) \wedge (2*x+y^z : \mathbb{Z}) \wedge \text{even}(a+b) \wedge \text{even}(2*x+y^z)$". For this context will be:

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
2	b	1	`QLet`	`Implicit`	`NoType`
2	x, y	2	`QLet`	`Implicit`	`NoType`
2	$a + b, 2 * x + y^z$	2	`QLet`	`Explicit`	Integer

`CV`

$\mathbb{S}_n$	Logical Formula	Statement Type
...	...	...
2	$(b, y, z :$ `NoType`$) \wedge (a + b : \mathbb{Z}) \wedge (2 * x + y^z : \mathbb{Z}) \wedge \text{even}(a + b) \wedge \text{even}(2 * x + y^z)$	Hypothesis

`ST`

Note in the above example, we separate variables and other expressions to make "let" and "assume" respectively using the code on lines 8 and 9.

Also note the lines 18–25 above. we separate collective and distributive properties and make functions by applying them on expressions in a collective and distributive manner respectively.

Finally, again note the context on lines 31–33. As we have seen, we store the whole expression in CV table (on line 31) as a convention. It seems a bit naive but there is no established algorithm to decide which identifier an anaphoric pronoun refers to. In fact in most of the cases this convention picks up the right referent.

```
Function name: declSubVars
Input:  [Formula] as forms, CV as cv
Output: 2-Type (CV, [Formula]
Procedure:
  procedure stops when the list forms is empty
  f = take first element of list forms
  vars = get all variables from f
  IF (vars is empty) //only literals
   THEN recursive call to (declSubVars (forms-f) cv)
  ELSE IF (vars==[f]) //no sub variables so it is taken care in mkTypeProps
   THEN recursive call to (declSubVars (forms-f) cv)
  ELSE do following:
   nvars = lookup each element of vars in cv of cntxt and
           filter those variables which are not declared
   cv1 = add (sn, nvars, (length nvars), QLet, Implicit, NoType) in cv
   (final_cv, rem_nvars) = recursive call to (declSubVars (forms-f) cv1)
   return (final_cv, nvars ++ rem_nvars)
```

The second and third constructs of noun adjunct: On lines 11 to 15 in function `makeExpsTypeProps` in §8.3.1.1 on page 194, the second and third constructs of noun adjunct are given. We cover them with functions `makeTypePropsCmnRel` and `makeTypePropsFun`. These functions are essentially defined in the similar way as we have defined function `makeTypeProps`. The only difference is the treatment of extra categories (of the second and third constructs of noun adjunct) in addition. For instance, consider an example for the second rule:

$$\text{We assume that } a \text{ and } b \text{ are non-zero integers } \underbrace{\text{with one common factor}}_{\text{Noun adjunct}}.$$

So we define `makeTypePropsCmnRel` and `makeTypePropsFun` in a way that they reuse the function `makeTypeProps`. Let us demonstrate it for one function shown below. Note the use of extra categories `GQuant1` and `Relation` on line 2 and 3, to cover the second construct of noun adjunct. Note that we do not need polarity for this function. Because for noun adjuncts, negative statements are not supported (as already reported in §6.3.2.1 on page 128).

```
1  Function name: makeTypePropsCmnRel
2  Input: [Formula] as forms, GType as tp, [Property] as ps, GQuant1 as quantA,
3         Relation as rel, Context as cntxt (or (cv,st,ce))
4  Output: Tuple (Context, [Assignment], [Assignment])
5  Procedure:
6    ((cv1,st1,ce1), autoAsgn, asgn) = makeTypeProps forms tp ps stype sn cntxt
7    letAsgns = filter_let_statements from asgn (Empty if there is no ‘let’ in asgn)
8    formulas = IF (filter_assume_statements from asgn) is not empty
9                THEN remove_Assume_keyword from it
10              ELSE Empty
11
12   str_rel   = mkStrRel rel
13   str_quant = transQuant1 quantA
14   cmnRelF = (transQuant1 quantA)++"_cmn_"++(mkStrRel rel)++"("++forms++")"
15
16   final_formula = IF formulas is empty THEN cmnRelF
17                   ELSE (conjunction of formula and cmnRelF)
18
```

```
19  assumeAsgns = [mkAssume final_formula] //simply makes "assume final_formula"
20
21  cv2 = cv1 //it remains unchanged
22  st2 = add in st (sn, final_formula, stype)
23  ce2 = ce1 //remain unchanged because no equation in this statement
24  cntxt2 = (cv2, st2, ce2)
25
26  return(cntxt2, autoAsgn, letAsgns++assumeAsgns)
```

The function `mkStrRel` on line 12, takes a relation and return a string:

`GFactor -> "Factor", GSquare -> "square"`, etc.

Whereas, the function `transQuant1` on line 13 takes `Quant1` and returns string:

`GOne -> "one", GTwo -> "two"`, etc.

Then in the next line we combine both functions with some other string literals to produce string such as `one_cmn_factor(x,y)` for "x and y ... with a common factor", `two_cmn_factor(a,b)`, etc.

In the if-condition on line 17, only two cases are possible: (1) the list is empty (2) or the list has one element (i.e. `formulas==[formula]`). The only usage of the first case is to encode the non-existence of formula (i.e. when list is empty).

On line 22, note that we ignore `st1`. Because it is produced by the function `makeTypeProps` and therefore, contains half formula. To demonstrate this point, consider again the example given on previous page:

We assume that a and b are non-zero integers $\underbrace{\text{with one common factor}}_{\text{Noun adjunct}}$.

Result produced by function `makeTypeProps`:
(Context, [ ] (i.e. empty), [let $a, b : \mathbb{Z}$, assume positive(b)$\wedge$ positive(b)])

Result added by function `makeTypePropsCmnRel` to the rule assume:
$\wedge$ **one_cmn_factor**(a, b)

We omit the implementation of function `makeTypePropsFun` which covers the third construct of noun Adjunct (in the function `makeExpsTypeProps` on page 194).

When subject is a pronoun: On lines 13–14, of function `mkProp1` on page 193, we are in the case when the subject is pronoun. In this case we need to resolve the anaphora for the pronouns. The procedure is already explained in §2.5.2 on page 19. But now we define the pseudo-code (i.e. function `makePronTypeProps`) for this case:

(On line 3, note the data type `Polarity`. It could have two constructors: `PPosit` for positive and `PNeg` for negative.)

```
1 Function name: makePronTypeProps
2 Input: GPron as pron, GQuant as quant, GProperties1 as props, GType as tp,
3       GAdjunctWith as adjnctW, Polarity as pol, Context as cntxt(co,st,ce)
4 Output: Tuple (Context, [Assignment], [Assignment])
5 Procedure:
```

```
6  pattern matching on adjnctW
7  IF adjnctW is GEmptyAdj //no adjunct
8   THEN do following:
9    pattern matching on pron
10   IF pron is GIt
11   lookup a latest record in cv with conditions that object is:
12     singular (i.e. 1), declared explicitly, tp==co.record.type (if not type=NoType)
13   IF found (sn, [f], 1, VarQuant, Explicit, type)
14     THEN makeTypeProps [f] props tp pol cntxt
15   ELSE return an error with sn that anaphora is not resolve for 'it'
16
17   ELSE do following:                    //GThey
18    lookup latest plural object in cv with conditions that object is:
19      (quant2n quant)== length of object (if quant is given), declared explicitly,
20      tp==co.record.type (if not type=NoType)
21    IF found (sn, forms, length forms, VarQuant, Explicit, type)
22      THEN makeTypeProps forms props tp pol cntxt
23    ELSE return an error with sn that anaphora is not resolve for 'they'
24
25 ELSE IF adjnctW is (GMkCmnRelAdjWith quantA rel) //e.g. no common factor
26  THEN same procedure that we've define on lines 11-20 for function:
27    (makeTypePropsCmnRel forms props tp quantA rel cntxt)
28
29 ELSE IF adjnctW is (GMkFunAdjWith eqwithRef) //e.g. with x>o
30  THEN same procedure that we've define on lines 11-20 for function:
31    (makeTypePropsFun forms props tp eqwithRef cntxt)
```

On line 11, for a statement containing pronoun "it", we lookup the latest `obj` of `(sn, obj, 1, any VarQuant, Explicit, type)` from `CV`, with conditions that it is singular (i.e. 1), it is declared explicitly, it should have the same type as the current statement and the `type` should not be `NoType`. If these conditions are fulfilled, we replace "it" with that `obj`. For instance, consider the following example:

a^2 is even because it is a multiple of 2.

The pronoun "it" is replaced by a^2 and we'll get:

a^2 is even because a^2 is a multiple of 2.

In case if the type is mentioned and both types match, such as:

Let x be an integer. Assume that it is a positive integer.

The pronoun "it" is replaced by x. But we'll send a warning that x is being declared twice. However, consider the following case when the types are different:

Let A be a set. Assume that it is a positive integer.

The pronoun "it" will not be replaced by A, and we lookup in the records previous to this in the context (in `CV`).

For pronoun "they" we do following:

1. If no quantity (e.g. two, three, etc) is mentioned in the statement, we replace pronoun "they" with the latest `[obj`$_1$`,...,obj`$_n$`]` of `(sn, [obj`$_1$`, ..., obj`$_n$`], n (i.e. length of list obj), VarQuant, Explicit, type)` from `CV` with a conditions that it is plural, declared explicitly, it has the same type as the current statement and the `type` should not be `NoType`.

2. Otherwise, if there is a quantity Q mentioned in the statement then we replace this pronoun with the latest [obj$_1$, ..., obj$_n$] of `(sn, [obj`$_1$`, ..., obj`$_n$`], n, VarQuant, Explicit, type)` from `CV` with the above mentioned conditions as well as one extra condition that $Q = n$.

To avoid being too long, we omit the implementation of the remaining four rules of `GProposition` (on page 191). But see the appendix A on page 239 for these rules being in action.

8.3.2 Equation with Reference

The category `EqWithRef` given in §6.3.5 on page 142 becomes `GEqWithRef` in Haskell, and its rules become the constructors, as given below:

```
data GEqWithRef = GMkEqWithRef GEq GRef
```

Here are few examples of equations with references, where $(a), 4.1$ are references:

> We assume that $(x+y)^2 = x^2 + y^2 + 2*x*y$ – (a).
> Squaring both sides yields that $2*b^2 = a^2$ 4.1.
> We get that $2*b^2 = (2*c)^2 = 4*c^2$ by substituting the value of a into equation 4.1

We now define the function `mkEqWithRef` in pseudo-code:

```
1  Function name: mkEqWithRef
2  Input: GEq as eq, GRef as ref, Context as cntxt(cv,st,ce)
3  Output: Tuple (Context, [Assignment], [Assignment])
4  Procedure:
5   equation = parse eq with BNFC parser
6   check that an equation is given (semantic check)
7   vars = get_all_variables_uniquely equation //e.g. 2*x+y=2*x returns [x,y]
8   (cv1, autovars) = declSubVars vars cv
9   left_eq  = left side of equation
10  lookup for a latest (sn, obj, 1, VarQuant, Explicit, type) with condition that left_eq==obj
11     t = if such record found then type else NoType
12  cv2 = add (sn, [left_eq], 1, QLet, Explicit, t) in cv1
13  st1 = add (sn, equation, stype) in st
14  ce1 = add (sn, equation, transRef ref) in ce
15  cntxt1 = (cv2, st1, ce1)
16  return (cntxt1, [mkLet autovars], [mkAssume equation])
```

On line 9, the choice of taking the left side of an equation to store for anaphoric resolution is a bit naive. We follow this convention because of the reason that there is no established

algorithm to decide which identifier an anaphoric pronoun refers to (cf. §3.3.4.1 on page 51). In fact taking the left side as a convention turns out to be a good decision for many equations. Also note on line 12 that we always have a single element as a list of referents. Here is the context for two examples:

> We assume that $(x + y)^2 = x^2 + y^2 + 2 * x * y$ – (a).
> (Consider x, y already available in the context as `QLet` and it is the 4th sentence.)

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
4	$(x + y)^2$	1	`QLet`	`Explicit`	`NoType`

`CV`

But, if we consider x, y to be already available in the context, but x as `QLet` and y as, let us say `QExists`, then the quantification in `CV` would be `QUndecided`. We give more details at the end of this subsection.

$\mathbb{S}_n$	Logical Formula	Statement Type
...	...	...
4	$(x + y)^2 = x^2 + y^2 + 2 * x * y$	Hypothesis

`ST`

$\mathbb{S}_n$	Equation	Reference
...	...	...
4	$(x + y)^2 = x^2 + y^2 + 2 * x * y$	a

`CE`

And if we have a pronoun "it " after this sentence, then pronoun "it" refers to $(x+y)^2$:

> We assume that $(x + y)^2 = x^2 + y^2 + 2 * x * y$ – (a). It is even.

Discussion on Quantification (`VarQuant`): The constructor `QUndecided` does not play any role currently; the others (`QLet`, `QExists`, `QUniversal`) do. The data type `VarQuant`, allows us to see if a variable is already defined or not. If it is defined then we can see how it was quantified. This information is useful in a situation, that can be better demonstrated by a superficial example:

> Theorem. ...Then there exist two integers u and v such that ....
> Proof. ... Assume that u and v are positive. ...

The variables u, v in the theorem are existentially quantified and bound. But of course, they are available as anaphoric referents in the later statements. When u, v from proof are analyzed, we lookup for them in the context. Because they are found as existential variables, we send a warning regarding it (i.e. multiple declaration for same variable names). However, u, v will be defined as: "let u, v : `NoType` assume positive(u) $\wedge$ positive(v)". But if we found this statement instead:

> Theorem. ...Then there exist two integers u and v such that ....
> Proof. ... Assume that they are positive. ...

Again We'll send a warning but produce: "let u, v : `NoType`" assume positive(u) $\wedge$ positive(v).

8.3.3 Existential Statements

The category `PropExist` given in §6.3.3 on page 136 becomes `GPropExist` in Haskell, and its rules become the constructors, as given below:

```
1  data GPropExist =
2      GMkPosExist1 GQuant GProperties1 GType GExps GEquations
3    | GMkNegExist1 GQuant GProperties1 GType GExps GEquations
4    | GMkPosExist2 GProperties1 GExps GEquations
5    | GMkNegExist2 GProperties1 GExps GEquations
```

We now define the function `mkExistsProp`. As usual, it allows us to build context, apply linguistic features and translate the Haskell data objects for CLM to something which can easily be converted into MathAbs on the macro level. The first rule of existential statements is covered by lines 2–3 above, and the second, rule is covered by lines 4–5 above.

We now define the skeleton function function `mkExistsProp`:

```
1  Function name: mkExistsProp
2  Input:  GPropExist as exist, Context as cntxt
3  Output: Tuple (Context, [Assignment], [Assignment])
4  Procedure:
5   pattern matching on exist
6   IF exist is (GMkPosExist1 quant props1 tp exps equations)
7    THEN (mkExistsProp1 quant props1 tp exps equations PPosit cntxt)
8
9   ELSE IF exist is (GMkNegExist1 quant props1 tp exps equations)
10   THEN (mkExistsProp1 quant props1 tp exps equations PNeg cntxt)
11
12  ELSE IF exist is (GMkPosExist2 props1 exps equations)
13   THEN (mkExistsProp2 quant props1 exps equations PPosit cntxt)
14
15  ELSE IF exist is (GMkNegExist2 props1 exps equations)
16   THEN (mkExistsProp2 quant props1 exps equations PNeg cntxt)
```

Regarding the translation of existential statements into MathAbs, it is already discussed in §4.6. We will use those rules in the pseudo-code. It can also be observed from §4.6 that we sometimes produce similar output for different statements (based on polarity and statement type). Therefore, let us recap and see what parts of code could be shared:

1. $\exists x : \mathbb{T}\, P(x)$ as *assumption*: let $x : \mathbb{T}$ assume $P(x)$ (quantification of x be `QLet`).

2. $\neg(\exists x : \mathbb{T}\, P(x))$ as *assumption*: assume $\forall x : \mathbb{T}\, \neg P(x)$ (quantification of x be `QUniv`).

3. $\exists x : \mathbb{T}\, P(x)$ as *goal*: show $\exists x : \mathbb{T}\, P(x)$ (quantification of x be `QExists`).

4. $\neg(\exists x : \mathbb{T}\, P(x))$ as *goal*: We have two possibilities:

   (a) let $x : \mathbb{T}$ assume $\neg P(x)$ (**Not possible** because of the principle that a negative statement cannot introduce a variable).

   (b) assume $\forall x : \mathbb{T}\, \neg P(x)$ (quantification of x be `QUniv`).

5. $\exists\, x : \mathbb{T}\, P(x)$ as *deduction*: deduce $\exists\, x : \mathbb{T}\, P(x)$ (quantification of x be `QLet`).

6. $\neg(\exists\, x : \mathbb{T}\, P(x))$ as *deduction*: deduce $\forall\, x : \mathbb{T}\, \neg P(x)$ (quantification of x be `QUniv`).

So we would need three function: one for bullets 2, 4 and 6 (let us name it `mkForallP`), second function for bullets 3 and 5 (let us name it `mkExistsP`), and third for bullet 1 (let us name it `mkLetAssumeP`). They will be define and used in the subsequent pages.

The first rule: It is covered on lines 6–7 and 9–10 above. Here is an example:

> There exist three even integers x, y and z such that $x * a > b$, $y * a > b$ and $z * a > b$.

We now define the skeleton function `mkExistsProp1` for it. In it, we combine the procedures defined for the first rule of simple propositions (cf. function `makekExpsTypeProps` of §8.3.1.1 on page 194) and the equations (cf. §8.3.2). In the following function, we first define existential variables with type and then apply it to a list of equations.

On line 8 below, this semantic check is necessary because in existential formula only variables should be allowed. For instance, it is fine to say $\exists x : \mathbb{T}(...)$; but it is illegal to say: "$\exists\, x + y : \mathbb{T}\,(...)$", "$\exists\, 3 : \mathbb{T}\,(...)$", etc.

In contrast to the category 'equation with reference' (§8.3.2 on page 200), we have to deal with a list of equations here. It is covered on lines 10–15 below. We now give the example given on page 203 as an assumption:

> Assume that there exist three even integers x, y and z such that $x * a > b$, $y * a > b$ and $z * a > b$.
> (Consider it to be the third sentence, and a being already defined as integer with "let".)

We get $[x, y, z]$ in the list forms (on line 6 below). we get the list `ps` containing one property in it: `[even]` (on line 7). Then on line 8, we check that all elements of list $[x, y, z]$ are variables, which is true.

On line 10, we get the `list_eqs` containing equations: $[x*a > b,\, y*a > b,\, z*a > b]$. On line 11, we fold it with conjunctions, making it:

$$(x * a > b)\ \wedge\ (y * a > b)\ \wedge\ (z * a > b).$$

On line 12, we get all the unique variables from `equation_f`, i.e. the above equation that we have just created. These variables are: $[x, a, b, y, z]$ (we have named this list `all_vars` in the code below). On line 13, we subtract those variables from the list `all_vars` which we are going to be declared as existentially quantified. We name this subtracted list `vars` ($[a, b]$), and we will declare those variables automatically which are not in the context (i.e. b) with "let b : `NoType`". Also we add it to the context (line 14):

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
3	b	1	`QLet`	`Implicit`	`NoType`

CV

After that, as mentioned on line 15, from the above mentioned list `equations` (i.e. $[x * a > b,\ y * a > b,\ z * a > b]$), we extract their left sides and add them (i.e. $[x * a,\ y * a,\ z * a]$) to `left_eqs`. The remaining explanation is given after the functions:

```
1  Function name: mkExistsProp1
2  Input: GQuant as quant, GProperties1 as props, GType as tp, GExps as exps,
3   GEquations as eqs, Context as cntxt(cv,st,ce)
4  Output: Tuple (Context, [Assignment], [Assignment])
5  Procedure:
6   forms = mkExps2Forms exps cn
7   ps = translate props to normal list property //i.e [Property]
8   check that all elements of forms are variables (otherwise report error)
9
10  list_eqs = parse the lists eqs with BNFC parser
11  equation_f= fold list_eqs with conjunction or disjunction (determined by parse tree)
12  all_vars = get_all_variables_uniquely equation_f //e.g. 2*x+y=2*x returns [x,y]
13  vars = subtract elements of forms from all_vars
14  (cv1, autovars) = declSubVars vars cv
15  left_eqs = left sides of list_eqs
16
17  t = translate tp to MathAbs type
18
19  IF (chkCollectiveProps forms ps)==False
20   THEN return error that collective property must apply on at least two expressions
21  ELSE
22   do following:
23   IF stype==Asm AND pol==PPosit THEN
24      mkLetAssumeP equation_f forms t cntxt
25
26   ELSE IF stype==Asm AND pol==PNeg THEN
27      mkForallP equation_f forms t cntxt
28
29   ELSE IF stype==(Prv or Justif) AND pol==PPosit THEN
30      mkExistsP equation_f forms t cntxt
31
32   ELSE IF stype==(Prv or Justif) AND pol==PNeg THEN
33      mkForallP equation_f forms t cntxt
34
35   ELSE IF stype==Ddc AND pol==PPosit THEN
36      mkExistsP equation_f forms t cntxt
37
38   ELSE IF stype==Ddc AND pol==PNeg THEN
39      mkForallP equation_f forms t cntxt
```

We code the most important part of the above function in on lines 21–39 above. It forms quantification and context. For our running example, we fall in the case described on line 23, and therefore, handled by the function `mkLetAssumeP` given below:

```
1  Function name: mkLetAssumeP
2  Input: Formula as equation_f, [Formula] as forms, Type as t, Context as cntxt(cv,st,ce)
3  Output: Tuple (Context, [Assignment], [Assignment])
4  Procedure:
5   mathabs_forms = make MathAbs [let forms:t, assume equation_f]
6   logical_form = (forms:t => equation_f)
7   search forms in cv (if any one found then send warning of variable name reuse)
8   cv2 = add in cv1 (sn, forms, length forms, QLet, Explicit, t)
```

```
9   lookup each element of left_eqs in cv2 for quantification and type:
10     IF they all have same quantification THEN assign it to eqs_q
11     ELSE assign QUndecided to eqs_q
12     IF they all have same type THEN assign it to eqs_t
13     ELSE assign NoType to eqs_t
14  cv3 = add in cv2 (sn, left_eqs, length left_eqs, eqs_q, Explicit, eqs_t)
15  st1 = add in st (sn, logical_form, stype)
16  FOR EACH eq : list_eqs
17    ce_n = add in ce one by one (sn, eq, no reference)
18  cntxt1 = (cv1, st1, ce_n)
19  return (cntxt1, autovars, mathabs_forms)
```

On line 5, we make output for our example:

$$[\text{let } x, y, z : \mathbb{Z}, \text{ assume } (x * a > b) \wedge (y * a > b) \wedge (z * a > b)].$$

On line 6, the logical formula is:

$$\text{“}(b : \texttt{NoType} \wedge x, y, z : \mathbb{Z}) \Rightarrow ((x * a > b) \wedge (y * a > b) \wedge (z * a > b))\text{”}.$$

On line 7, we try to find x, y, z in the context `cv`, to send a warning of variable reuse just in case they are already declared before. After that on line 8, we add then in the context `cv`, as shown below:

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
3	b	1	QLet	Implicit	NoType
3	x, y, z	3	QExists	Explicit	Integer

CV

On lines 9–13 above, we lookup each of $[x*a, y*a, z*a]$ in the context `cv`, and assign `QUndecided` to `eqs_q` (because these elements do not have the same quantification); and `Integer` to `eqs_t` (because they have the same type “integer”). Then of course, on line 14–15, we add them in the context (i.e. in `CV`, `ST` and `CE`):

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
3	b	1	QLet	Implicit	NoType
3	x, y, z	3	QExists	Explicit	Integer
3	$x * a, y * a, z * a$	3	QUndecided	Explicit	Integer

CV

$\mathbb{S}_n$	Logical Formula	Statement Type
...	...	...
3	$(b : \texttt{NoType} \wedge x, y, z : \mathbb{Z}) \Rightarrow ((x * a > b) \wedge (y * a > b) \wedge (z * a > b))$	Hypothesis

ST

$\mathbb{S}_n$	Equation	Reference
...	...	...
3	$x * a > b$	no
3	$y * a > b$	no
3	$z * a > b$	no

CE

We give below the pseudo-code of the function `mkForallP` (without explanation and examples), which is used three times in our main function `mkExistsProp1` which we define for the first rule.

```
1  Function name: mkForallP
2  Input: Formula as equation_f, [Formula] as forms, Type as t, StType as stype,
3        Position as sn, Context as cntxt(cv,st,ce)
4  Output: Tuple (Context, [Assignment], [Assignment])
5  Procedure:
6      mathabs_forms = make MathAbs [assume ∀forms:t (equation_f)]
7      logical_form = ∀forms:t (equation_f)
8      search forms in cv (if any one found then send warning of variable name reuse)
9      cv2 = add in cv1 (sn, forms, length forms, QUniv, Explicit, t)
10
11      lookup each element of left_eqs in cv2 for quantification and type:
12        IF they all have same quantification THEN assign it to eqs_q
13        ELSE assign QUndecided to eqs_q
14        IF they all have same type THEN assign it to eqs_t
15        ELSE assign NoType to eqs_t
16     cv3 = add in cv2 (sn, left_eqs, length left_eqs, eqs_q, Explicit, eqs_t)
17     st1 = add in st (sn, logical_form, stype)
18     FOR EACH eq : list_eqs
19        ce_n = add in ce one by one (sn, eq, no reference)
20     cntxt1 = (cv1, st1, ce_n)
21     return (cntxt1, autovars, mathabs_forms)
```

We omit the implementation of function `mkExistsP` and other rules due to space limitations.

Take Statement as Proof Statement The proof statements containing the category `TakeStmnt` in given is bullet 1 of §7.2.1.6 on page 166. Instead of translating it to MathAbs, our aim is to show the use of constructor `QExists` of `Quantification`. We start by an example:

1. We can choose $u := 1$ and $v := 0$.

This statement suggests that there must be an existential statement appeared in the theorem as a goal (cf. bullet 3 of §4.6 on page 72). For instance:

2. Prove that there exist two integers u and v such that $u * n + v * m = \gcd(n, m)$.

(Such theorem and its proof is already given in Chapters 4 (figure 4.4 on page 77) and 7 (§7.2.1.7 on page 167). See more in Appendix A on page 248 and page 251).

When we encounter the second statement, the variables u, v are recorded with the construct `QExists` in the `Quantification` field. Now, when we encounter the first statement later in the text, we lookup these variables in the context `CV`, and impose the condition that these variables in the first statement also be quantified with the constructor `QExists`. Otherwise we treat them as usual let variables.

8.3.4 Statement and its lists

We combine propositions, equations (with references), relational and existential statements into one category called `Statement` as shown in §6.3.6 on page 143. In Haskell `Statement` becomes:

```
1 data GStatement =
2     GMkPropStmnt  GProposition //covered in §8.3.1 on page 191
3   | GMkExistStmnt GPropExist   //covered in §8.3.3 on page 202
4   | GMkRelStmnt   GPropRel     //omitted
5   | GMkEqRefStmnt GEqWithRef   //covered in §8.3.2 on page 200
```

We simply define a function with combines all of them together:

```
1 Function name: mkStmnt
2 Input:  GStatement as stmnt, Context as cntxt
3 Output: Tuple (Context, [Assignment], [Assignment])
4 Procedure:
5  pattern matching on stmnt
6  IF stmnt is (GMkPropStmnt  prop)        //GProposition as prop
7    THEN mkProp prop cntxt
8  ELSE IF stmnt is (GMkExistStmnt  eprop)  //GPropExist as eprop
9    THEN mkExistsProp eprop cntxt
10 ELSE IF stmnt is (GMkEqRefStmnt  eq)     //GEqWithRef as eq
11   THEN mkEqWithRef eq contxt
12 ELSE IF ....
13   THEN ....
```

In a similar way, we define function for `LetStatement` (but reusing as much code as we can).

We now define a function `mkStmnts` for list of statements (`Statements`). It will be extensively used in the macro level grammar (§8.3.7) for the list of statements. Recall that in the GF grammar, we make such lists in two parts: the part containing $n-1$ elements separated by comma (partial list, in case of Statement it is `PStatements`); and the part containing nth element. Both are joined by conjunction or disjunction (using constructs). However, we do not go into the technicalities of partial statements (`PStatements`) and how we discover if it is a list separated by conjunction or disjunction (or exclusive disjunction in case of `EStatements`, §6.3.6.2 on page 145). We simply consider it to be done, as shown on lines 5–6 below.

On lines 7–10, we apply `mkStmnt` to each element of `Statements`, and combine the results in a way that:

1. The MathAbs formulas for these elements are nicely folded on conjunction or disjunction (or exclusive disjunction in case of `EStatements`, See §6.3.6.2 on page 145).

2. Context is properly updated from first statement to the final statement. Note that the `ST` in context should not be updated for each statement in the list `Statements`. Instead, as shown on line 13, we add formula produced on line 11.

```
1 Function name: mkStmnts
2 Input:  GStatements as stmnts, Context as cntxt
```

```
3  Output: Tuple (Context, [Assignment], [Assignment])
4  Procedure:
5   list_stmnts = convert stmnts to normal Haskell list
6   logical_op  = extract the logical operator from stmnts
7   FOR EACH stmnt_i:list_stmnts
8     ((cv_i+1,st_i+1,ce_i+1), autoAsign_i, asign_i) = mkStmnt stmnt_i cntxt_i
9     let_i = extract all Let rules from asign_i
10    formula_i = extract formulas attached to rule Assume from asign_i
11  formula = fold formula_i with logical_op
12
13  final_st = add in st (sn, formula, stype)
14  final_context = (final cv_i+1, final_st, final ce_i+1)
15
16  return (final_context, autoAsign_i++let_i, [mkAssume formula])
```

We now give an example and demonstrate this procedure on it:

$$\text{We conclude that } \underbrace{m \text{ and } r \text{ are coprime}}_{\text{Statement 1}}, \text{ and } \underbrace{r < m}_{\text{Statement 2}}.$$

Suppose that m, r are already declared in the context as integers. Of course the key phrase "we conclude that" belongs to the proof block and it is not the part of `Statements`. The following context and MathAbs will be formed for it:

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
n	m, r	2	`QLet`	`Explicit`	Integer
n	r	1	`QLet`	`Explicit`	Integer

`CV`

$\mathbb{S}_n$	Logical Formula	Statement Type
...	...	...
n	coprime$(m, r) \wedge (r < m)$	Deduction

`ST`

$\mathbb{S}_n$	Equation	Reference
...	...	...
n	$r < m$	no

`CE`

MathAbs: assume coprime$(m, r) \wedge (r < m)$

8.3.5 Conditional Statement

As mentioned in §6.3.7 on page 146, conditional statements takes two list of statements, as show below in Haskell data type:

```
data GPropIfthen = GMkPropIfthen GStatements GStatements
```

The function we define, simply go through both `GStatements` and record the occurrence of each object(s) for `CV`. For the equations occurring in these lists of statements are stored in CE as usual. It is `ST`, which will different result. If we have the logical formula A for the first list of statement (i.e. Statements$_i$ folded by conjunction, disjunction, or

exclusive disjunction) and the logical formula B for the second list (i.e. Statements_j folded by conjunction, disjunction, or exclusive disjunction), then logical formula for conditional statement is $A \Rightarrow B$. But MathAbs would be "assume A deduce B". (or assume A deduce B based on the block):

```
1  Function name: mkPropIfthen
2  Input:  GStatements as cnds, GStatements as rsults, Context as cntxt(cv,st,ce)
3  Output: Tuple (Context, [Assignment], [Assignment])
4  Procedure:
5   (cntxt1,        auto1, asign1) = mkStmnts cnds   cntxt
6   ((cv2,st2,ce2), auto2, asign2) = mkStmnts rsults cntxt1
7
8   formula_cnds   = extract formula from asign1
9   formula_rsults = extract formula from asign2
10  logical_formula = formula_cnds => formula_rsults
11
12  final_st = add in st (sn, logical_formula, stype)
13  final_context = (cv2, final_st, ce2)
14
15  return (final_context, auto1++auto2, asign1++[mkDeduce formula_rsults])
```

Consider the following example:

> If x and y are two even integers then $x + y$ is even.
> Assume that x, y are not declared by let before. Otherwise we must return a warning as well.

The context will be following:

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
n	x, y	2	QLet	Explicit	Integer
n	$x + y$	1	QLet	Explicit	NoType

CV

$\mathbb{S}_n$	Logical Formula	Statement Type
...	...	...
n	$(x, y : \mathbb{Z} \wedge \text{even}(x) \wedge \text{even}(y)) \Rightarrow \text{even}(x + y)$	Deduction

ST

Nothing to add in CE.

> **MathAbs:** let $x, y : \mathbb{Z}$ assume $\text{even}(x) \wedge \text{even}(y)$ deduce $\text{even}(x + y)$

Consider another example:

> If $a = 2 * c$, and $4 * c^2 = 2 * b^2$, and $2 * c^2 = b^2$ then b is even.
> Consider a, b, c to be declared as integers by let.

The context will be following:

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
n	a	1	QLet	Explicit	Integer
n	$4 * c^2$	1	QLet	Explicit	NoType
n	$2 * c^2$	1	QLet	Explicit	NoType
n	b	1	QLet	Explicit	Integer

CV

$\mathbb{S}_n$	Logical Formula	Statement Type
...	...	...
n	$((a = 2 * c) \wedge (4 * c^2 = 2 * b^2) \wedge (2 * c^2 = b^2)) \Rightarrow \text{even}(b)$	Deduction

ST

$\mathbb{S}_n$	Equation	Reference
...	...	...
n	$a = 2 * c$	no
n	$4 * c^2 = 2 * b^2$	no
n	$2 * c^2 = b^2$	no

CE

MathAbs: assume $(a = 2*c) \wedge (4*c^2 = 2*b^2) \wedge (2*c^2 = b^2)$ deduce $\text{even}(b)$

8.3.6 Justifications

Justifications are defined in §6.3.9 on page 147. In Haskell it becomes the following data type:

```
1  data GJustification =
2         GMkStmntJust  GStatement
3       | GMkOperJust   GOperation
4       | GMkDefJust    GDefinition
5       | GMkAnphrJust  GAnaphor
6       | GMkDefRefJust GDefReference
```

Similar to `Statement` and its list `Statements`, we have the list `Justifications` (defined on page 152). Furthermore, we define function `mkJustif` and `mkJustifs` respectively for these data objects. We omit the implementation of function `mkJustifs`, but `mkJustif` is defined as shown below. But in its output the last element of tuple is `Hint` instead of `[Assignment]`. `Hint` is first defined on page 64, as part of MathAbs proof language. Like `[Assignment]`, `Hint` is also a list, but it terminates with construct `NoHint`, as shown below in the LBNF format grammar:

```
FormHt.  Hint ::= "by" "form" Formula Hint ;
OperHt.  Hint ::= "by" "oper" Ident Hint  ;
....
NoHint.  Hint ::= ; //empty
```

The first constructor of justification is formed by statements (`Statement`). Therefore, we reuse the function `mkStmnt` as shown on line 8 below. Consider the following example, in which justification is formed by a statement:

"We conclude that a is even $\underbrace{\text{because } a^2 \text{ is even}}_{\text{Justification}}$."

Suppose that a is already declared as integer. Context and automatically defined variables (which is none for our example) for justification remain the same (except that the statement type stype is `Justif` now). The only difference is the last element of the output tuple. It is a `Hint` now. So on line 9, we get the formulas from `[Assignment]`, and on lines 10–11, we make `Hint` from it. For this example, it is "`FormHt` even(a^2) `NoHint`". Consider another example translation: On line 9, in case we input "[let $v : \mathbb{T}$, assume A]", the function will return $[v : \mathbb{T}, A]$, and the hint would be "`FormHt` $v : \mathbb{T}$ (`FormHt` A `NoHint`)".

```
1  Function name: mkJustif
2  Input:  GJustification as justif, Context as cntxt
3  Output: Tuple (Context, [Assignment], Hint)
4  Procedure:
5    pattern match on justif
6    IF justif is (GMkStmntJust stmnt)       //GStatement as stmnt
7     THEN
8      (cntxt1, auto1, asgn1) = mkStmnt stmnt cntxt (with stype=Justif)
9      hintfs = get_formulas_from_Asgns asgn1
10     FOR hintf:hintfs (1 to n)
11       hint = FormHt hintf_1(FormHt hintf_2 ... (FormHt hintf_n NoHint)))
12     return (cntxt1, auto1, hint)
13
14   ELSE IF justif is (GMkOperJust oper)    //GOperation as oper
15    THEN transOperation oper cntxt
16
17   ELSE IF justif is (GMkDefJust df)       //GDefinition as df
18    THEN return (cntxt, [], DefHt (transDef df) NoHint)
19
20   ELSE IF justif is (....)
21    THEN ....
22
23   ...
```

Coming back to the example sentence given above, we produce following context (nothing to add in `CE`, therefore, it is omitted) and MathAbs for it:

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
...	...	...	...	...	...
n	a	1	`QLet`	`Explicit`	Integer
n	a^2	1	`QLet`	`Explicit`	`NoType`

`CV`

$\mathbb{S}_n$	Logical Formula	Statement Type
...	...	...
n	even(a)	Deduction
n	even(a^2)	Justification

`ST`

MathAbs: deduce even(a) by formula even(a^2)
(Of course, "by formula even(a^2)" is the linearization of `FormHt` even(a^2) `NoHint`)

This brings us to the explanation of other constructors starting from line 14 in the function above. These constructors are briefly explained in the following subsections:

8.3.6.1 Operations as justifications:

Operations as justifications (i.e. `Operation`) is in given §6.3.9.1 on page 148.

The first rule for operational justifications is formed by `Rel1`. For instance,

$$\underbrace{\text{Squaring at both sides}}_{\text{Operation as justification}}\text{, we get } (a+b)^2 = 4 * c^2$$

Clearly, there is nothing to be saved in the context for this justification. First, we apply a semantic check that there is an equation somewhere behind this sentence. Then finally we produce the MathAbs:

MathAbs: `deduce` $(a + b)^2 = 4 * c^2$ by oper `squaring_both_sides`(latest equation before this sentence)

The second rule for operational justifications is formed by `Rel2` and `Exp`. For instance,

$$\underbrace{\text{Multiplying both sides by 2}}_{\text{Operation as justification}} \text{ yields the result that } b^2 = 2 * c^2.$$

The literal 2 is saved in `CV` for the justification. Again, first, we apply a semantic check that there is an equation somewhere behind this sentence. Then finally we produce the MathAbs:

MathAbs: `deduce` $b^2 = 2 * c^2$ by oper `multiplying_by_2`(latest equation before this sentence)

We omit the rest of rules. See Appendix A for their examples.

8.3.6.2 Anaphoric Reference

It described in §2.5.2.2 on page 22. Later, as mentioned in §6.3.9.2, typical patterns for anaphoric references supported by CLM are:

1. (the | our) (first | last) (statement | hypothesis | deduction | equation | justification).

2. theorem 24, theorem, axiom, definition, etc.

References given in bullet 1 are solved; but reference to theorem, definition, axiom, etc, mentioned in bullet 2 are left as it is.

We use `CV` and `CE` tables, to know the (first | last) hypothesis, deduction, conclusion, equation, justification, etc. For instance, consider the following example:

> Since a^2 and b^2 are non zero integers, we conclude that a^2 is even. By the last deduction, a is even.

Here "last deduction" refers to "even(a^2)". See appendix A on page 239 for more examples.

The rest of categories

To avoid being too long, we omit the implementation of the rest of categories defined in Chapter 6. However, with examples given in appendix A on page 239, it seems to be easy to imagine this implementation. Of course, the MathNat source code is also made available for interested readers.

8.3.7 The Macro Level Grammar

The Math Document

The macro level grammar is defined in Chapter 7 on page 155. As stated before, the MathNat document in CLM is a collection of axioms, definitions, theorems and proofs, structured by specific keywords (i.e. Axiom, Definition, Theorem and Proof).

With a simple LBNF grammar, we describe a list of theorems and their proofs, list of axioms and a list of definitions. These lists are given to the host system MathNat, where (as already stated and defined for micro level grammar) we apply specialized functions for context building, block structure building, linguistic support and the translation to MathAbs. Because these procedures are somewhat interleaved. Therefore, we have already explained them all in one to save time, space and effort (because going through the abstract syntax several time is expensive).

Let us concentrate on the function for a theorem and its proof for the moment, which is of course applied on a list of theorems and their proofs with higher order map function. For each block structure, let us say, theorem and its proof, we start the analysis with an empty `Context` which evolves while examining the AST of each sentence. This procedure is repeated until we reach the end of the text in that block.

8.3.8 Theorem

Basically, we have already build most of the machinery for linguistic features and denotational semantics. In the blocks such theorem and proof we just have to put things together.

We start here the context building procedure with theorem block. Similar to the data type `GPrfStmnts` given in §8.1.2 on page 181, the theorem block is represented by the data type `GThmStmnts`. Analogous to its GF code (where it is simply a list of theorem statements in §7.2.2 on page 170), which is defined as:

```
data GThmStmnts =
    GBaseThmStmnts GThmStmnt
  | GConsThmStmnts GThmStmnt GThmStmnts
```

Similarly, `GThmStmnt` is formed from the corresponding GF code given in §7.2.2 on page 170, as shown below. Note that each constructor above correspond to a GF function defined between §7.2.2.1 to §7.2.2.3 (pp. 170–173).

```
data GThmStmnt =
    GMkThmSymb GSymbFormula
  | GThmProves  GStatements  GSubordinate
  | GThmEProves GEStatements GSubordinate
  ...
```

We now describe the function `buildThmStmnts`. Basically its a map function which applies function `buildThmStmnt` on the the list of theorem statements. Therefore, it takes the list `GThmStmnts` and `Context`, and returns a 2–tuple `(Context, Theorem)`, as shown below in the pseudo-code.

```
Function name: buildThmStmnts
Input:  GThmStmnts as thmstmnts, Context as cntxt
Output: Tuple (Context, Theorem)
Procedure:
 pattern matching on thmstmnts
 IF thmstmnts is (GBaseThmStmnts thmstmnt)
  THEN
   (cntxt1, thm) = buildThmStmnt thmstmnt cntxt
   return (cntxt1, thm)

 ELSE IF thmstmnts is (GConsThmStmnts thmstmnt thmstmnts)
  THEN
   (cntxt1, thm)  = buildThmStmnt  thmstmnt  cntxt
   (cntxt2, thms) = buildThmStmnts thmstmnts cntxt1
   updated_thm    = update thm with thms
   return (cntxt2, updated_thm)
```

As we can see, its main purpose is to update theorem (`Theorem`), which is finally returned in the output above. It is the MathAbs theorem whose definition is given in §4.3 on page 61. We can define it as LBNF grammar as shown below:

```
TAssgn.    Theorem ::= Assignment Theorem ;
TShow.     Theorem ::= "show" Formula ;
TEnd.      Theorem ::= "---" ;
TFullStop. Theorem ::= "." Theorem ;
```

As we can see, `Theorem` above is also a list, which terminates on two constructs: `TShow` and `TEnd`. `TShow` corresponds to the theorem statements which we ought to prove (given in §7.2.2.1). Whereas, `TEnd` is not a part of the definition of `Theorem` given in §4.3. Instead, it is just a placeholder in this tree. It is removed from MathAbs at the final step of translation. The construct `TAssgn` corresponds to the assumptions in the theorem (given in §7.2.2.2). The construct `Assignment` used in it is already defined on page 192. It makes "let" and "assume" in MathAbs theorem and Proof. Finally, the construct `Formula` used in these other constructs is defined in figure 6.2 on page 109.

The above LBNF code for `Theorem` corresponds to the following abstract syntax, which is of course accessible in the host system MathNat.

```
data Theorem = TAssgn Assignment Theorem
     | TShow Formula | TEnd | TFullStop Theorem
```

As evident from the pseudo-code of function `buildThmStmnts`, the main function for building context for theorem statements is `buildThmStmnt`, which we define below. It is a long function, covering all theorem statements defined in §7.2.2 by pattern matching.

```
1 Function name: buildThmStmnt
2 Input:  GThmStmnt as thm, Context as cntxt
```

```
3  Output: Tuple (Context, Theorem)
4  Procedure:
5   pattern matching on thm
6   IF thm is (GMkThmSymb symbForm) //first rule for goal in theorem
7    THEN (procedure_first_rule_goal_thm symbForm sn cntxt)
8
9   ELSE IF thm is (GThmProvesR2 stmnts subord) //second rule for goal
10   THEN (procedure_second_rule_goal_thm stmnts subord cntxt)
11
12  ELSE IF thm is (GThmProvesR3 eitherStmnts subord) //third rule for goal
13   THEN (procedure_third_rule_goal_thm eitherStmnts subord cntxt)
14  ....
15  ELSE IF thm is (GMkThmLet letStmnts subord) //first rule for assumption
16   THEN (procedure_first_rule_assume_thm stmnts subord cntxt)
17
18  ELSE IF thm is (GThereforeThmSt thm1) //rule for miscellaneous key phrases
19   THEN (procedure_rule_misc_keyphrases_thm thm1 cntxt)
20  ....
```

For the following sections and subsections, assume that each of them are accessed by pattern matching in the skeleton function `buildThmStmnts` defined on page 215. Therefore, these forthcoming sections and subsections should be treated as procedures, and each returns a tuple `(Context, Theorem)`.

8.3.8.1 Statement to Prove

The first rule for goal in theorem As we have seen in §7.2.2.1 on page 170, this rule is directly formed by symbolic formulas. We omit its implementation and directly jump to the following example to explain it:

> The formula: $\forall(x, y : \mathbb{Z})\,(\text{even}(x) \wedge \text{even}(y) \Rightarrow \text{even}(x+y))$, for statement:
> "if x and y are two even integers then $x + y$ is even"
> In theorem, it should be the only statement:
>
> Theorem. show $\forall(x, y : \mathbb{Z})\,(\text{even}(x) \wedge \text{even}(y) \Rightarrow \text{even}(x+y))$

This formula in MathAbs remains unchanged:

> Theorem. show $\forall(x, y : \mathbb{Z})\,(\text{even}(x) \wedge \text{even}(y) \Rightarrow \text{even}(x+y))$

And its context would be following. Note that in `CV`, only quantified variables are added. Even nested quantifiers including those which are deep inside the formula.

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
1	x, y	2	`QUniv`	`Explicit`	Integer

`CV`

$\mathbb{S}_n$	Logical Formula	Statement Type
1	$\forall(x, y : \mathbb{Z})\,(\text{even}(x) \wedge \text{even}(y) \Rightarrow \text{even}(x+y))$	Goal

`ST`

Nothing to add in `CE`.

So the output of this procedure is : (`Context`, show A), where A is an arbitrary logical formula. Consider another example below, in which we have both universal and existential variables:

Theorem. $\forall x : \mathbb{Z}\ \exists y : \mathbb{Z}(x + 1 = y)$

As usual this formula in MathAbs remains unchanged. However, we give `QUndecided` quantification to them in `CV`. Note that like other statements, x, y is added here together in one line, and therefore, they will be available as referents together.

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
1	x, y	2	`QUndecided`	`Explicit`	Integer

CV

$\mathbb{S}_n$	Logical Formula	Statement Type
1	$\forall x : \mathbb{Z}\ \exists y : \mathbb{Z}(x + 1 = y)$	Goal

ST

$\mathbb{S}_n$	Equation	Reference
1	$x + 1 = y$	no

CE

The second rule for goal in theorem As we have seen in §7.2.2.1 on page 171, it is formed by some key phrases, list of statements (i.e. `Statements`) and subordinate (i.e. `Subordinate`, defined on page 158). Some example statements are:

"$\underbrace{\text{prove that}}_{\text{key phrase}}$ $\underbrace{\text{there exists two integers } u \text{ and } v \text{ such that } u * n + v * m = 1}_{\texttt{Statements}}$ $\underbrace{\quad}_{\texttt{Subordinate}\text{(empty)}}$",

"$\underbrace{\text{show that}}_{\text{key phrase}}$ $\underbrace{x + y \text{ is even,}}_{\texttt{Statements}}$ $\underbrace{\text{where } x \text{ and } y \text{ are positive integers}}_{\texttt{Subordinate}}$", etc.

We define below a function `procedure_second_rule_goal_thm` for this rule as mentioned on page 215. Recall that the stype (i.e. statement type) from this rule is `Prv` (i.e. Goal).

```
1  Function name: procedure_second_rule_goal_thm
2  Input: GStatements as stmnts, GSubordinate as subord, Context as cntxt
3  Output: Tuple (Context, Theorem)
4  Procedure:
5   pattern matching on subord
6   IF subord is GEmptySubord //i.e. Empty subordinate
7    THEN
8     (cntxt1, auto, asgns) = mkStmnts stmnts cntxt
9     formula               = get formula from the list asgns
10    thm                   = mkAsgnPrvThm (auto,form)
11    return (cntxt1, thm)
12
13  ELSE IF subord is (GMkSubord sbrd) //GStatements as sbrd
14   THEN
15    (cntxt1, auto1, asgns1) = mkStmnts stmnts cntxt (with stype Prv)
16    (cntxt2, auto2, asgns2) = mktmnts sbrd cntxt1 (with stype Asm)
17    form                  = get formula from the list asgns1
18    thm                   = mkAsgnPrvThm (auto1++auto2++asgns2,form)
19    return (cntxt1, thm)
```

This code is quite easy to follow and it mostly uses the already defined functions. On line 10, we use a function `mkAsgnPrvThm`. It takes a list of assignment (i.e. rules

let and assume) and a MathAbs formula and simply forms rules: let assume show. For instance, in case of input: ([let $v_1 : \mathbb{T}_1$, let $v_2 : \mathbb{T}_2$, assume A, assume B], formula), the output be: let $v_1 : \mathbb{T}_1$ let $v_2 : \mathbb{T}_2$ assume A assume B show formula.

When subordinate is not empty, as shown in the second example above and on line 13 of the above function, it acts as an assumption in the discourse. Therefore, on line 16, we call `mkStmnts` function with `stype` equals to `Asm`. As described on page 158, Subordinate is formed from `Statements`. Therefore, we can use function `mkStmnts` for it.

Assignments such as `auto1`, `auto2` and `asgns2` forms the rules let, assume. Whereas, `asgns1` contains our goal. For instance, consider the second example again:

Theorem. $\underbrace{\text{Show that}}_{\text{key phrase}}$ $\underbrace{x + y \text{ is even,}}_{\texttt{Statements}}$ $\underbrace{\text{where } x \text{ and } y \text{ are positive integers}}_{\texttt{Subordinate}}$.

Theorem. let $x, y : \mathbb{Z}$ assume positive(x) $\wedge$ positive(y) show even($x + y$) •

And the context be:

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
1	$x + y$	1	`QLet`	`Explicit`	NoType
1	x, y	2	`QLet`	`Explicit`	Integer

`CV`

$\mathbb{S}_n$	Logical Formula	Statement Type
1	$(x, y : \mathbb{Z}) \wedge \text{positive}(x) \wedge \text{positive}(y)$	Hypothesis
1	$\text{even}(x + y)$	Goal

`ST`

Nothing yet in `CE`.

8.3.8.2 Assumptions in Theorem

As mentioned in §7.2.2.2 on page 173, there are many ways in which we can describe assumptions in theorem block. In fact we have already seen few examples in the description of micro level grammar (between pages 191 to 213). So we omit them also.

8.3.9 Proof

The proof block in the GF grammar is a list of proof statements `GPrfStmnts` given in §8.1.2 on page 181. Whereas, in MathAbs proof is a tree as described in §4.3 on page 61. We give below the LBNF grammar for it:

```
1 Trivial.      Proof   ::= "trivial" Hint ;
2 Show.         Proof   ::= "show" Formula Hint Proof ;
3 Deduce.       Proof   ::= "deduce" Formula Hint Proof ;
4 Assgn.        Proof   ::= Assignment Hint Proof ;
5 Split.        Proof   ::= "{" [Proof] "}" Hint;
6 Unfinished.   Proof   ::= "unfinished" ;
7 FullStop.     Proof   ::= "." Proof ;
8 End.          Proof   ::= "---" ;
```

As we see, `Proof` is a tree, which terminates on three constructs: `Trivial`, `Unfinished` and `End`. Similar to constructor `TEnd` of theorem, `End` is also a placeholder for empty nodes. Both are removed from the MathAbs at the final step of translation.

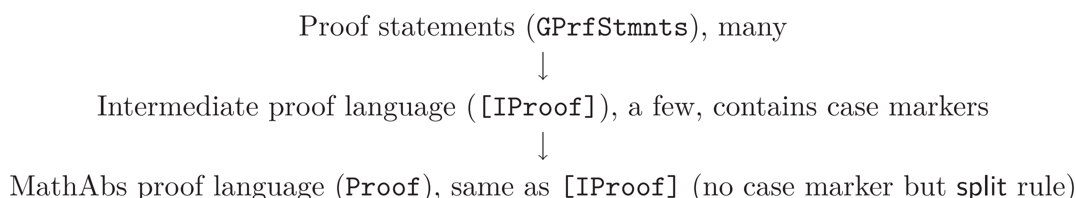


Figure 8.1: CLM grammar to MathAbs

However, we first translate the list of proof statements `GPrfStmnts` to an intermediate proof language, let us say `[IProof]`. Similar to the list `GPrfStmnts`, the list `[IProof]` is a linear list of proof statements. Instead of a tree, it is a list. So we do not need the constructor `End` as the proof language does. We also do not have split rule (cf. bullet 6 on page 63). Instead, we have many case markers which allows us to build the split rule during case analysis (again in §8.3.11 on page 221). We add extra 'I' to every construct, as shown below:

```
ITrivial.     IProof   ::= "trivial" Hint ;
IShow.        IProof   ::= "show" Formula Hint ;
IDeduce.      IProof   ::= "deduce" Formula Hint ;
IAssgn.       IProof   ::= Assignment Hint ;
...
```

The difference between the list of proof statements `GPrfStmnts` and the list `[IProof]` (and of course with `Proof`) can be described as a "many to one" relation. For instance: we may have n ways to describe a statement in CLM; most of them are translated to different abstract syntax trees. Whereas, all of them are translated to a few rules in `[IProof]` (i.e. `ILet`, `IAssume`, `IDeduce`, etc, which later correspond to the rules of MathAbs proof such as let, assume, deduce, etc, respectively). In figure 8.1, we summarize their relation. We summarize their relation in figure 8.1.

Analogous to the functions `buildThmStmnts` and `buildThmStmnt`, we first define skeleton functions `transPrfStmnts` and `transPrfStmnt`. The implementation of `transPrfStmnts` is omitted because it is simply made by applying the function `transPrfStmnt` on `GPrfStmnts` using a higher order map function. Whereas, the function skeleton `transPrfStmnt` is described below:

```
1 Function name: transPrfStmnt
2 Input:  GPrfStmnt as prf, Context as cntxt
3 Output: Tuple (Context, [IProof])
4 Procedure:
5  pattern matching on prf
6  IF prf is (GMkAssume stmnt sbord)
7   THEN (procedure_for_this_rule stmnt sbord cntxt)
8  ....
9  ....
```

Again, for the following subsections, we assume that each of them are accessed by pattern matching. Therefore, these forthcoming subsections should be treated as procedures, and each returns a tuple `(Context, [IProof])`. However, we only discuss assumptions and deductions with justifications.

8.3.9.1 Deductions with Justifications

These are mentioned in §7.2.1.5 on page 162.

The first rule for deduction with justifications in proof statement is formed by some key phrases, `Statements`, `justifications` and `Subordinate`. Here are few example:

"$\underbrace{(empty)}_{\text{key phrase}}$ $\underbrace{a^2 \text{ is even}}_{\texttt{Statements}}$ $\underbrace{\text{because it is a multiple of 2}}_{\texttt{Justifications}}$ $\underbrace{(empty)}_{\texttt{Subordinate}}$ ",

"$\underbrace{\text{we get that}}_{\text{key phrase}}$ $\underbrace{2 * b^2 = 4 * c^2}_{\texttt{Statements}}$ $\underbrace{\text{by substituting } a \text{ in equation 1}}_{\texttt{Justifications}}$ $\underbrace{\text{where } c \text{ is an integer}}_{\texttt{Subordinate}}$",

"there exist q and r such that $n = m * q + r$ by euclidean division",

"q divides r because $r = n - m * q$", etc.

We define a function for this rule below. Note the order in which we call other functions in this function. To ensure the correct context building, this order should be in accordance with the order in which these categories appear in the text. Also note that it is very important because the algorithm for anaphoric resolution always uses text order.

```
1  Function name: rule1_deduction_with_justifications
2  Input: GStatements as stmnts, GJustifications as justifs, GSubordinate as subord,
3      Context as cntxt@(cv,ce,st)
4  Output: Tuple (Context, [IProof])
5  Procedure:
6   pattern matching on subord
7   IF subord is GEmptySubord //i.e. Empty subordinate
8    THEN
9     (cntxt1, auto1, asgns) = mkStmnts stmnts cntxt (with stype Ddc)
10    (cntxt2, auto2, hints) = mkJustifs justifs cntxt1
11
12    formula = get formula from the list asgns
13    deduce_rule = [IDeduce formula hints]
14    iprfs = make [IProof] with (auto1++auto2) deduce_rule
15    return (cntxt2, iprfs)
16
17  ELSE IF subord is (GMkSubord sbrd) //GStatements as sbrd
18   THEN
19    (cntxt1, auto1, asgns1) = mkStmnts stmnts cntxt (with stype Ddc)
20    (cntxt2, auto2, hints)  = mkJustifs justifs cntxt1
21    (cntxt3, auto3, asgns2) = mktmnts sbrd cntxt2 (with stype Asm)
22
23    formula = get formula from the list asgns1
24    deduce_rule = [IDeduce formula hints]
25    iprfs = make [IProof] with (auto1++auto2++auto3++asgns2) deduce_rule
26    return (cntxt2, iprfs)
```

In case of example (assume that equation 1 is "$2 * b^2 = a^2$" and a is equal to $\sqrt{2} * b$. Also assume that that a, b are already defined as integers):

$$\text{"}\underbrace{\text{we get that}}_{\text{key phrase}}\ \underbrace{2 * b^2 = 4 * c^2}_{\texttt{Statements}}\ \underbrace{\text{by substituting } a \text{ in equation 1}}_{\texttt{Justifications}},\ \underbrace{\text{where } c \text{ is a positive integer"}}_{\texttt{Subordinate}}$$

Its MathAbs is:

let $c : \mathbb{Z}$ assume positive(c)
deduce $2 * b^2 = 4 * c^2$ by oper substitution($a, 2 * b^2 = a^2$)

The context will be:

$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
n	c	1	`QLet`	`Implicit`	NoType
n	$2 * b^2$	1	`QLet`	`Explicit`	NoType
n	a	1	`QLet`	`Explicit`	Integer
n	c	1	`QLet`	`Explicit`	Integer

`CV`

Note that when we encounter the equation "$2 * b^2 = 4 * c^2$", the variable c is not yet defined and therefore, it is automatically define in the first row. Later when we discover its declaration in subordinate, the fourth row suppresses its old declaration.

$\mathbb{S}_n$	Logical Formula	Statement Type
n	$2 * b^2 = 4 * c^2$	Deduction
n	$c : \mathbb{Z} \wedge \text{positive}(c)$	Hypothesis
n	substitution$(a, 2 * b^2 = a^2)$	Justification

`ST`

Note that in MathAbs, the hypothesis in second row appears first. But in the table below, this order does not matter. So we add them as they appear in the text.

$\mathbb{S}_n$	Equation	Reference
n	$2 * b^2 = 4 * c^2$	no

`CE`

8.3.10 Theorem and its Proof

As already said in §8.1.2 on page 181, we have a category `ThmPrf` for theorem and its proof in our CLM implementation in the GF grammar. Of course it becomes following in Haskell:

```
data GThmPrf = GMkThmPrf GThmStmnts GPrfStmnts
```

The CLM data object `GThmPrf` should be translated to the MathAbs' data object for theorem and its proof, given below:

```
ThmPrf. Math ::= ThmLabel Theorem ";" "Proof." Proof ";" ;
```

Where `ThmLabel` is simply a string label, which allows us to write labels such as: “Theorem 1.3.”, “Theorem.”, etc.

We define the following function which works on `GThmPrf` and adds a MathAbs theorem and proof in a MathAbs document (which is formed by `[Math]`).

```
1 Function name: transThmPrf
2 Input: GThmStmnts as gthms, GPrfStmnts as gprfs
3 Output: [Math]
4 Procedure:
5   (cntxt_thm, thm)   = buildThmStmnts gthms emptyCntxt
6   (cntxt_prf, iprfs) = transPrfStmnts gprfs cntxt_thm
7   prfs    = buildProof iprfs emptyTree emptyEnv
8   thmprfs = [mkThmPrf thm prf thmLabel prfLabel | prf <- prfs]
9   return thmprfs
```

On lines 5–6, we use already defined functions for theorem and proof blocks (i.e. `buildThmStmnt` and `transPrfStmnts`). The function `transPrfStmnts` returns the list of intermediate proof constructs (`[IProof]`) with context `cntxt_prf`.

As we have seen that this context has helped us to support various linguistic features including anaphoric resolution. When the function `transPrfStmnts` ends, proof statements are already translated to the intermediate proof language `IProof` (which is very similar to the MathAbs proof language) and all of the linguistic features are applied. Therefore, we no longer need this context for the final procedure `buildProof` on line 7.

The procedure `buildProof` is for the case analysis for ‘proof by case’ explained in the next section. On line 7, the list of intermediate proof statements `iprfs` is given to this function along with some other empty environments to produce the final proof tree (in both case: when the proof in consideration has a case structure, and when it has not).

The output of this function could be more than one proof for those proofs which requires case handling. Because proof with cases can be ambiguous having more than one interpretation. So on line 8, we select each proof from this list of proofs (i.e. `prf <- prfs`) and simply put them together with MathAbs theorem one by one, and return again a list: `[Math]`. We explain it further with the following pattern which has at least two interpretations:

> **Theorem.** *Theorem Statement*
> **Proof.** [*Proof statements*]
> If condition then ... . If condition then ... . Otherwise ... .
> [*Proof statements*]

In the first interpretation, we attach the ‘otherwise’ case to the first conditional, and in the second interpretation, we attach it to the second conditional. In both interpretations of this proof we simply attach the same theorem block.

8.3.11 Case Analysis for Proof by Cases

The grammar we support for proofs having cases is given in §7.2.1.7 on page 167. As mentioned there, in the current algorithm, we only try to disambiguate cases with case markers. We recognize case markers with various sentences, treated as proof statements in CLM (see page 168). Consequently, this translation is called from the function

`transPrfStmnt` given in §8.3.9 on page 219. The GF implementation of these specific rules is also given on page 168.

Also Recall the definition of `[IProof]` from §8.3.9 on page 217. It is an intermediate proof language which contains case markers as well. We now describe the proof statements for cases and their equivalent constructs in `[IProof]` side by side between brackets **(...)**, as shown below:

Pattern 1:

We proceed by case analysis.
Case: **(IStartCaseAnls)** *Condition* **(ICondMarker)** `PrfStmnts` [This ends the case. **(IEndCase)**]
Case: **(IStartCase)** *Condition* **(ICondMarker)** `PrfStmnts` [This ends the case. **(IEndCase)**]
. . . . . .
Case: **(IStartCase)** *Condition* **(ICondMarker)** `PrfStmnts` [This ends the case. **(IEndCase)**]
[This was the last case. **(IEndLastCase)**]

`IStartCaseAnls` above shows that it is the first case.

Pattern 2:

If **(IIfCaseAnlys)** *condition* **(ICondMarker)** then `PrfStmnts`
Otherwise if **(IOtherwiseIfC)** *condition* **(ICondMarker)** then `PrfStmnts`
Otherwise if **(IOtherwiseIfC)** *condition* **(ICondMarker)** then `PrfStmnts`
. . . . . .
Otherwise **(IOtherwiseC)** `PrfStmnts`

Other than these case markers, the following rules may also trigger an end of case:

"This ends the proof", "It is trivial", etc, —–> `ITrivial`
"we will prove it later", etc, —–> `IUnfinished`

To build these case structures, we use a data structure Zipper[1] [Huet 1997] and modify it to our needs. A Zipper is a forest of trees with path. It provides an easy way to move between the sibling, child and parent nodes of the structure. Here is the description of some functions used in pseudo code:

- `addinTree`: adds a tree at the current focus of zipper.
- `insertDown`: inserts a child node in the zipper data structure. This new child becomes the current tree in focus. e.g. `(insertDown End prfZipr)` adds an empty proof tree (`End`) as the first child in `prfZipr` zipper.
- `scope`: returns the tree in focus.
- `insertRight`: inserts a tree as sibling to the right of the current position. This new tree becomes the current tree. Note that we do not have the function `insertLeft`. Consequently, we impose the restriction that the current case is always on the right most branch.
- `root`: returns the top-most parent of the given location.
- `depth`: if it is greater than zero then we have children (i.e. there are case structures in our environment. How many? It could be determined by the number depth.)

[1] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/rosezipper`

- fstBranch returns the first branch of our case analysis.

The algorithm to build case structure is a bit long and tedious. We present here a simplified implementation of a few cases below, in the form of pseudo code. For that the function buildProof takes the list [IProof], a zipper data structure for MathAbs Proof (ProofLoc), a zipper data structure for the environment in which only above mentioned case markers are stored as a list (EnvLoc) and returns a list of MathAbs Proofs ([Proof]). It returns a list because the case structure may also result into more than one proof if the input is ambiguous. However, in §8.3.11.2 on page 225, we'll come to know that this ambiguity is often temporary.

```
1 Function name: buildProof
2 Input: [IProof] as iprfs, ProofLoc as prfLoc, EnvLoc env //(two zippers)
3 Output: [Proof]
4 Procedure: ...
```

We now describe both zippers one by one followed by an example. The zipper EnvLoc is forest of lists, defined as shown below:

```
type Env = [IProof]
data EnvLoc = ELoc {
      list   ::  Env        // The currently selected list
    , lefts  :: [Env]       // Siblings on the left, closest first
    , rights :: [Env]       // Siblings on the right, closest first
     // The contexts of the parents for this location
    , parents :: [([Env], Env, [Env])] //[(lefts, list, rights)]
      }
```

At any point in the procedure buildProof, EnvLoc gives access to previously stored case markers. That is why we call it environment. For instance, the labels list, lefts and rights give access to the current case analysis in consideration. Whereas, the label parents gives access to all the parent levels of case analysis.

For building a proof, we mostly consider the current proof statement (i.e. IProof) and 1 to n most recent case markers in our environment to decide about where we shall place the coming proof statements in our zipper for proof (i.e. ProofLoc).

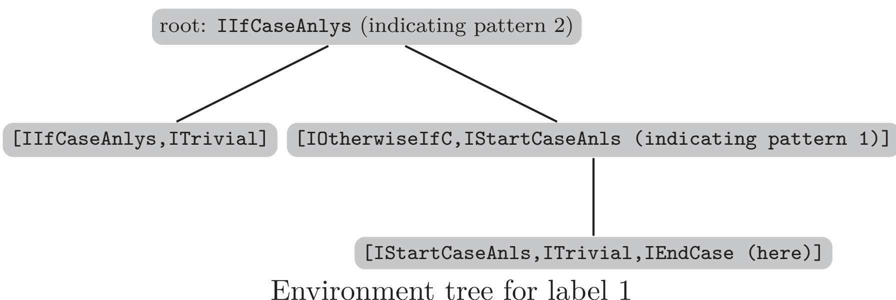
The zipper ProofLoc is a forest of proof trees. It remembers the paths to build the list of proofs later. The ProofLoc also gives access to the proof trees that we have build so far. Note that it is made by Proof, and therefore, it does not contain case markers but split rule(s) (see page 63 to recall its definition). The zipper ProofLoc is defined as shown below:

```
data PrfLoc = PLoc {
        ptree   ::  Proof         // The currently selected proof tree
      , plefts  :: [Proof]        // Siblings on the left, closest first
      , prights :: [Proof]        // Siblings on the right, closest first
        // The contexts of the parents for this location
      , pparents :: [([Proof], Proof, [Proof])] //(plefts, ptree, prights)
  }
```

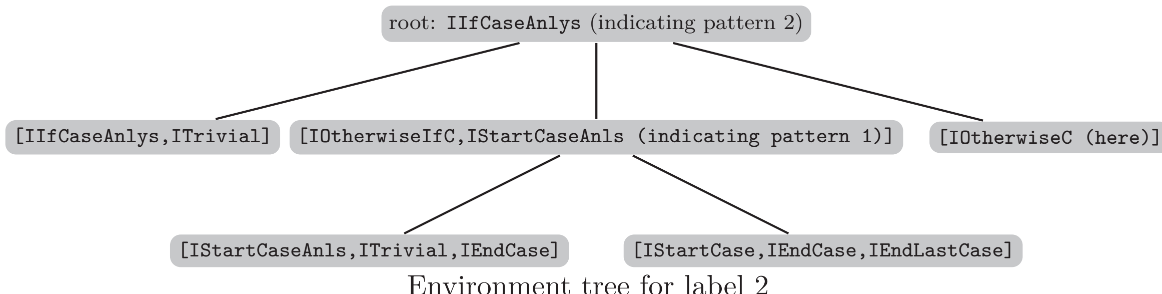
We now give an example, followed by its EnvLoc and PrfLoc.

Proof. *Proof Statements*
If $condition_{1.1}$ then ... . It is trivial.
Otherwise if $condition_{1.2}$ then ... . We proceed by case Analysis.
Case: $condition_{2.1}$ ... . It is trivial. This ends the case.
(*we are here: label 1*)
Case: $condition_{2.2}$ ... . This ends the case. This was the last case.
Otherwise ... . (*we are here: label 2*)

First, the environment tree formed by `EnvLoc` zipper will contain case markers in the following fashion for the *label 1* and *label 2* respectively. Note that nothing will be saved in it except the case markers:



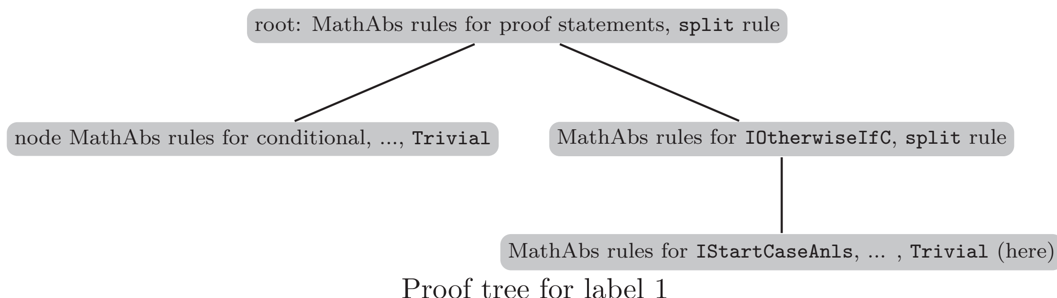
Environment tree for label 1



Environment tree for label 2

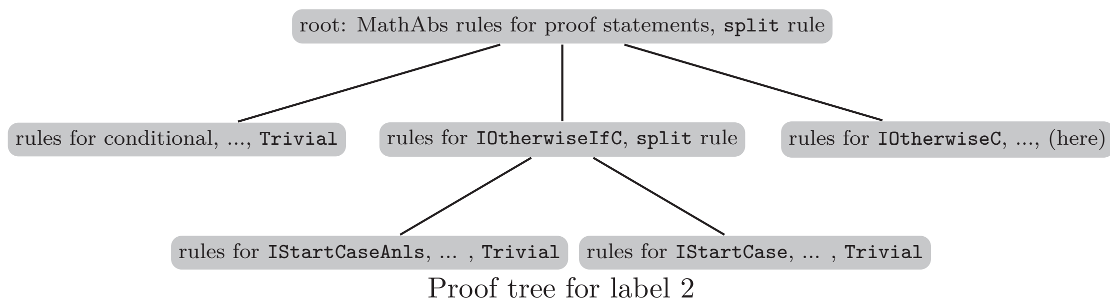
As we saw, `EnvLoc` is a tree of lists. So when we fetch the "current tree in focus", we get a list. Also it is evident from the example that case markers are added in the tree in such a way that the latest added case marker is always the last element. We access it with the function `last`[2].

We now show the proof tree built by the proof zipper `PrfLoc` for this example. It contains proof statements in the following fashion. Note that only case markers are stored in it:



Proof tree for label 1

[2]In actual code, it is the opposite: the latest added case marker is always the last element for the purpose of efficiency, and hence accessed with the function `first`.

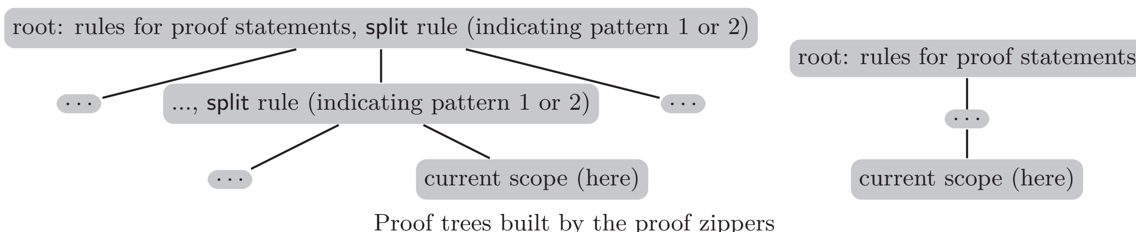


Proof tree for label 2

Note for subsequent pages: To avoid the confusion between the case of the 'proof by case' method and the case in phrases such as "in this case, we do ...", we rephrase the later as "in this possibility, we do ...".

8.3.11.1 Consider the possibility, when the current element in `[Iproof]` is not a case marker:

When the current element of the list `[IProof]` is not a case marker, it is a normal proof statement. Therefore, we add it in the `ProofLoc`, in our tree in focus (the current scope), as shown in the proof trees below.



Proof trees built by the proof zippers

```
5  iprf = take first element of iprfs
6  IF iprf is one of the following rules:
7     let, assume, deduce, show, take, fullstop
8   THEN do following:
9     prf = translate iprf to the rule of MathAbs proof
10    prfLoc1 = addinTree prf in prfLoc
11    // recursive call for the remaining iproof statements (i.e. iprfs - iprf)
12    buildProof (iprfs - iprf) prfLoc1 env
```

8.3.11.2 Consider the possibility, when the current element of `[IProof]` is a conditional statement:

Here, we check the environment `env` which contains all the case markers that have already occurred in the textual proof. We have five possibilities to consider as shown below.

Consider the first possibility, when no case marker is found in the environment `env` before this conditional (i.e. `env` is empty), as shown by the following pattern:

> [Usual proof statements such as assumptions, deductions, ...] **If** *condition* **then** ... .

Note that conditional is shown as a last statement. It is because this conditional is in focus, and we always lookup in backward direction in the list `[IProof]` (using environment), but never in forward direction.

We always add a sub proof as shown in the example and on line 16 in the pseudo code below. Because if the remaining proof statements in the list `[IProof]` contain case markers such as "otherwise if", "otherwise", etc, then the new branch(es) to the proof will be added later.

However, if this conditional turns out to be a normal statement (i.e. it is not followed by statements containing the case markers "otherwise if" or "otherwise") then, it will be a proof which have only one branch. In other words, it will be a proof without case analysis, and therefore, on this proof we will apply flattening. To explain it further, we use the following example proof:

> Suppose that p is prime and p divides $a * b$. **If** $p \nmid a$ **then gcd**$(a, p) = 1$.
> [(proof statements)]
> **MathAbs**:
> let $p \in$ Prime assume $p \mid a * b$ $\bullet$
> { assume $p \nmid a$ deduce $\gcd(a, p) = 1$ $\bullet$ [(proof statements)] }

Recall that in MathAbs, we represent branches by split rule (represented by curly brackets {...}; see page 63). For instance, if we have two branches then the MathAbs will be:

> ... {*Sub-proof 1* ; *Sub-proof 2*}

If the proof turns out to have only one branch in split rule(s) as we have discussed above, then we apply flattening to such a tree by removing split rule(s), as following:

> *Proof* {*Sub-proof 1*} $\equiv$ *Proof Sub-proof 1*
> *Proof* {*Sub-proof 1* { *Sub-proof 2* {...} }} $\equiv$ *Proof Sub-proof 1 Sub-proof 2 ...*

For the example given above, the MathAbs would be:

> let $p \in$ Prime assume $p \mid a * b$ $\bullet$
> assume $p \nmid a$ deduce $\gcd(a, p) = 1$ $\bullet$ ....(usual steps)

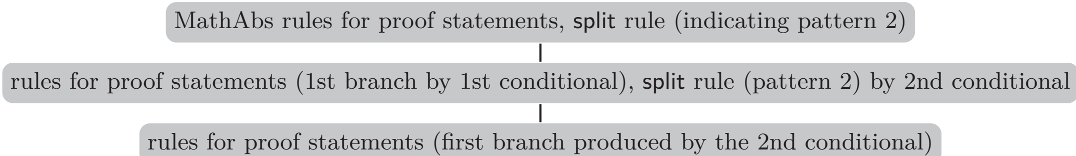
```
13 ELSE IF iprf is IIfCaseAnlys THEN
14
15   IF (scope env) is empty THEN
16     insertDown End prfLoc //i.e. an empty sub proof
17     insertDown IIfCaseAnlys env
```

Note that other than adding an empty sub proof on line 16, we did nothing for the conditional. It is because the function `transPrfStmnts` on page 221, has already translated the conditional to the combination of `IProof` rules: (`ILet`, `IAssume`, (`IShow` or `IDeduce`). Of course, they will be translated to the combination of MathAbs rules: let, assume, (show or deduce) respectively by the procedure on lines 5–12 above.

Consider the second possibility, when there is a conditional statement in the environment `env`, as shown by the following pattern (this conditional is encapsulated in brackets):

> [proof statements] (**If** *condition* **then** ...) [proof statements] **If** *condition* **then** ... .

Similar to the first possibility, we add an empty sub proof for this conditional. Because it may be a case inside case. Of course, we cannot be sure with the information in hand and wait for the remaining proof statements to occur. Then we'll be able to decide if we should apply flattening to this proof of not. The proof tree built by the `ProfLoc` is:



Recall that for environment, we access the list in focus with function `scope` and latest added case marker with function `last` (as shown on line 19 below). We use 'caseMarker' as a placeholder on line 18 below. We define a placeholder because we will refer to the same procedure again, but for different 'caseMarker':

```
18  caseMarker = IIfCaseAnlys
19  ELSE IF (last (scope env)) is caseMarker THEN
20    newprfLoc = (insertDown End prfLoc)
21    newenv    = (insertDown caseMarker env)
22    buildProof (iprfs except iprf) newprfLoc newenv
```

Consider the third possibility, when there is an "otherwise if" case marker in the environment `env` (i.e. there is an "otherwise if" statement before the current conditional). It means we already have encountered at least one case analysis. So we check the depth of the zipper, which must be greater than 0. Also we must have the first branch produced by "if" case marker. We check both of these conditions in pseudo code below on line 25. Note that in the code: "`(last (scope (fstBranch env)))`", the function `fstBranch` returns the zipper with first sibling in focus. Then `scope` gets the list, and `first` returns its first element (i.e. the earliest case marker).

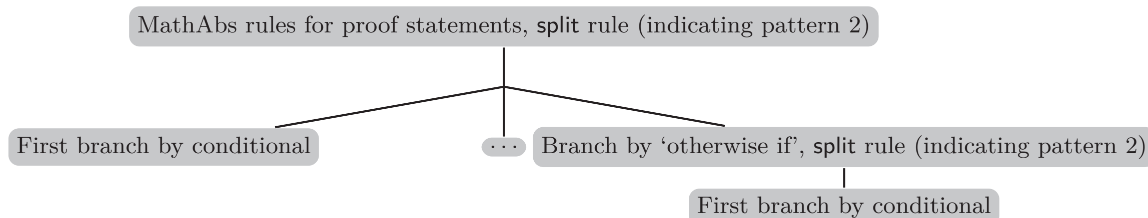
```
23  ListCaseMarker = [IOtherwiseIfC, IOtherwiseC]
24  ELSE IF (last (scope env)) is (caseMarker in ListCaseMarker) THEN
25    IF (depth of prfLoc>0) AND (first (scope (fstBranch env))) is IIfCaseAnlys THEN
26       same procedure given on lines 20-22
27    ELSE return a warning that 'caseMarker before "if"; nested case analysis expected.'
```

Similarly, consider the possibility of an "otherwise" case marker in `env`. These two possibilities could be demonstrated by the following pattern:

1. [proof statements] (Otherwise if *condition* then ... .) [proof statements] **If** *condition* **then** ... .

2. [proof statements] (Otherwise ... .) [proof statements] **If** *condition* **then** ... .

Again, we trigger a new case level with this conditional. So its implementation is similar to the second possibility. We show proof tree for the first bullet above:



Consider another similar possibility, when case markers: `IStartCaseAnalys`, `IStartCase` are in the environment `env`, forming the following patterns:

1. [proof statements] (we proceed by case analysis. Case: *condition* ... .) [proof statements] **If** *condition* **then** ... .

2. [proof statements] (Case: *condition* ... .) [proof statements] **If** *condition* **then** ... .

Note that the code remains the same as given on lines 23–27 above; except two changes:

1. The list `ListCaseMarker` becomes equal to `[IStartCaseAnls, IStartCase]`

2. On line 25, for the second check, we confirms that the first element of the first sibling (i.e. case) is `IStartCaseAnls`.

It is not necessary to perform the check in the second bullet if the `caseMarker` in consideration from the list `ListCaseMarker` is `IStartCaseAnls`. It is because `IStartCaseAnls` is already the first case. However, checking this condition does not harm in any way.

Finally, the rest of possibilities, (i.e. `IEndCase`, `IEndLastCase`, `ITrivial`, `IUnfinished`), they should not be here. Because they show that the current branch is already finished. Consequently, no statement (i.e. `iprf`) including conditional cannot be here. Therefore, if the above mentioned case markers appear, we return an error:

```
28   ELSE return error that 'a pattern is given which is not supported.'
```

8.3.11.3 Consider the possibility, when the current element of `[IProof]` is an "otherwise if" statement:

Consider the first possibility, when no case marker is found in the environment `env` before this case marker. In this case we return an error:

```
29 ELSE IF iprf is IOtherwiseIfC THEN
30   IF (last (scope env)) is empty THEN
31     return error that '"if" case marker is missing'
```

Consider the second possibility, when the case marker `IIfCaseAnlys` (i.e. a conditional) is found the environment. There could be following possibilities:

(If *condition* ...) **Otherwise if** *condition* ...
If *condition* ... (If *condition* ...) **Otherwise if** *condition* ...
If *condition* ... If *condition* ... (If *condition* ...) **Otherwise if** *condition* ...,
and so on.

To know if there are any other conditionals before case marker `IOtherwiseIfC`, we keep looking the parent nodes as on line 40 in the pseudo code below, and check it further recursively on line 42.

If there are conditionals before, then it is ambiguous to decide where the branch produced by the current case marker "otherwise if" should attach. Therefore, for n conditionals behind this case marker, we produce n different interpretations.

For instance, in case of the third pattern above, we produce following interpretations:

1. If *condition* ... If *condition* ...
   $\underbrace{\textbf{If}\ \textit{condition}\ ...\ \textbf{Otherwise if}\ \textit{condition}\ ...}$.

2. If *condition* ...
   $\underbrace{\textbf{If}\ \textit{condition}\ ...\ \text{If}\ \textit{condition}\ ...\ \textbf{Otherwise if}\ \textit{condition}\ ...}$.

3. $\underbrace{\textbf{If}\ \textit{condition}\ ...\ \text{If}\ \textit{condition}\ ...\ \text{If}\ \textit{condition}\ ...\ \textbf{Otherwise if}\ \textit{condition}\ ...}$
   .

```
32  caseMarker = IIfCaseAnlys
33  ELSE IF (last (scope env)) is caseMarker THEN
34
35     prfLoc_a = (insertRight End prfLoc) //an empty sub proof
36     env_a    = (insertRight caseMarker env)
37     prf_a    = buildProof (iprfs except iprf) prfLoc_a env_a
38
39     IF prfLoc has a parent prfLoc1 AND env has a parent env1 AND
40        (last scope env1) is caseMarker THEN
41
42       prf_n = buildProof iprfs prfLoc1 env1  //recursive call
43       return (prf_a++prf_n)
44
45     ELSE return prf_a
```

Consider the third possibility, when case marker `IOtherwiseIfC` is found before "otherwise if", as shown by the following pattern:

If *condition* then ... . [Otherwise if *condition* ... .] (Otherwise if *condition* ... .) **Otherwise if** *condition* ... .

Unlike the above possibility it is not ambiguous. Therefore, we add a branch to the right of the current tree in focus (lines 49–50 below). Also, as we have done in other possibilities, we lookup the first branch and confirms that it is formed by "if" case marker (line 47):

```
46   caseMarker = IOtherwiseIfC
47   ELSE IF (last (scope env)) is caseMarker AND
48          (first (scope (fstBranch env)) is IIfCaseAnlys) THEN
49
50     prfLoc_a = (insertRight End prfLoc) //an empty sub proof
51     env_a    = (insertRight caseMarker env)
52     buildProof (iprfs except iprf) prfLoc_a env_a
```

Consider the fourth possibility, when case marker `IOtherwiseC` is found before "otherwise if", as shown by the following pattern:

> If *condition* then ... . [Otherwise if *condition* ... .] (Otherwise ... .)
> **Otherwise if** *condition* ... .

The case marker `IOtherwiseC` implies that we are in the last case of *pattern 2*. Therefore, there must be "if" or "otherwise if" statement before in the environment at the same case level (i.e. "otherwise" must have sibling branches containing case(s) "if" or "otherwise if"). However, it should have already been checked when we have processed this statement (as we do on line 48 above). Therefore, for the purpose of clarity and because the case marker `IOtherwiseC` cannot be attached here as branch (sibling), we will not talk about this case level in subsequent paragraphs anymore.

We now extend the above mentioned pattern a bit more:

> . . .
> 1. If condition ... . [Otherwise if condition ... .](*could be more above*)
> 2. $\underbrace{\text{If condition ... . [Otherwise if condition ... .] \textbf{(Otherwise ... .)}}}$
> (*most recent case marker in scope* ↖)
> 3. **Otherwise if** *condition* ... .(*we are on this case marker*)

For our possibility in hand (i.e. "otherwise if" case marker on line 3 above), there must be at least one "if" or "otherwise if" statement as parent in the environment (line 1).

As we have seen in second and third possibilities on previous pages, when the case markers "if" and "otherwise if" are the most recent in the environment, we follow two different strategies. Therefore, we have to take this into account here, and have two possibilities as well:

1. When there is an "if" as the most recent case marker in the parent of the environment. Then follow the procedure in the second possibility on page 229.

2. When there is an "otherwise if" as the most recent case marker in the parent of the environment. Then follow the procedure in the third possibility on page 229.

We omit its implementation.

Consider the fifth possibility, when one of the case markers (`IStartCase` | `IEndCase` | `IEndLastCase`) is the most recent. For instance, in case of `IStartCase` case marker, we could have following pattern:

If *condition* ... . [Otherwise if *condition* ... .]
We proceed by case analysis. Case: *condition* ... .
[Case: *condition* ... .]
(Case: *condition* ... .) (*most recent case marker in scope*)
Otherwise if *condition* then ... . (*we are on this case marker*)

Fundamentally, it is similar to the fourth possibility and it is treated in a similar way. Therefore, we omit the implementation.

Consider the sixth possibility, when the case markers behind "otherwise if" is `IStartCaseAnls`, as shown in the following pattern:

If *condition* ... . [Otherwise if *condition* ... .]
(We proceed by case analysis. Case: *condition* ... .)
(*most recent case marker in scope*)
Otherwise if *condition* then ... . (*we are on this case marker*)

It means that *pattern 1* here has only one case. We do not allow it and return an error.

Recall that proof statements: `ITrivial`, and `IUnfinished` are also recorded in the environment. Therefore, **consider the seventh possibility** when the case marker behind "otherwise if" is one of above proof statements. In this case we simply look behind these case markers in the environment and recursively call the same function again.

Note that it is important to record `ITrivial` and `IUnfinished` as case markers. It is because, they can indicate an 'end of case'. Also, if we encounter all 'end case' markers for a case in *pattern 1* such as: `ITrivial`, `IEndCase`, `IEndLastCase`, then all of them must be translated to only one rule (i.e. trivial) in a proof. It becomes easier to do it by recording `ITrivial` and `IUnfinished` in the environment.

Other than the above mentioned possibilities, if the most recent case marker is other than above case markers, we return an error that ' "if" case marker is missing before "otherwise if" case marker.'

8.3.11.4 The case when the list [IProof] is empty

It means we have reached the end of proof. We lookup our environment and consider the following possibilities:

If the environment is empty, then this proof is a linear proof. Consequently, our proof tree does not contain the split rule (i.e. no 'proof by case'). We simply collect this proof from the `ProfLoc` zipper.

If the environment is not empty, then we simply collect all branches on each level from the zipper `ProfLoc`, and build proof tree. If this proof is incomplete, we find out in the process of building the proof tree, and report an appropriate reason.

8.3.11.5 The rest of cases

The rest of cases are built in a similar way. That is at each step, we look backward and build all the possible proofs.

pattern 1 causes more ambiguities than *pattern 2*. It is because, we always know its last case with the 'otherwise' case marker. But in *pattern 1*, '`IEndLastCase`' case marker is optional. When it is not given we have more ambiguity.

8.3.12 The rest of blocks, categories, statements and rules

We omit the implementation of the rest of blocks, categories, statements and rules defined in Chapter 7 due to space limitations.

CHAPTER 9

Concluding Remarks

Contents

9.1 Results **233**

9.2 Limitations and Future Directions **234**

9.2.1 Ambiguity 234

9.2.2 Proof Checking 237

9.2.3 Multilingual CLM 237

9.2.4 Generation from MathAbs 238

9.1 Results

We repeat the two very hard problems raised in the introduction chapter:

1. **Automatic Formalization:** Parsing mathematical texts (mainly proofs) and translating their parse trees to a formal language after resolving linguistic issues. This formal language should be able to represent mathematics as close as possible to the intentions expressed by the author and must be independent of any logic and prover.

2. **Validation** of this formal version of mathematics.

The project MathNat (**Math**ematics in controlled **Nat**ural language) aims at being a first step towards addressing these problems. However, this thesis attempted to answer the first question: the formalization of the mathematical texts. Whereas, the second problem is answered partially in a very limited way.

The partial results we achieved so far are encouraging. It seems possible to develop a system which scales up to the mathematical texts for elementary mathematics. However, it would clearly require a lot of time, effort and man power (i.e. linear efforts).

As a first step, we analyzed the language of mathematics in Chapter 3 and gave a comprehensive survey with a focus on elementary mathematics. After this survey, we developed a controlled language CLM. It has the look and feel of textbook mathematics and it supports some rich linguistic features (Chapters 5, 6 and 7).

We have tried to develop this grammar in such a way that it is modular and easily scalable. We also support the method 'proof by case' with two patterns.

This supported linguistic features turns out to be very useful (as demonstrated by the examples). They add flexibility in the language and increases the expressive power. Anaphoric resolution of pronouns and references is naive yet very useful.

The MathNat system is still a prototype and it can formalize only a handful of examples mainly from elementary number theory and analysis. However, with the best of our knowledge, this number is equally comparable (if not better) than what other related systems can parse (for evidence, see Appendix A).

However, as soon as we add examples from other branches of mathematics, the MathNat may face the danger of becoming insufficient to cope with. The only way to tackle this problem would be to add examples from various branches of mathematics and then adapt the system MathNat to the new challenges when they occur.

Notwithstanding this, a careful reading of any math book is sufficient to realize that developing a system which even allows to write math texts from a single branch is itself a challenging task. Therefore, in our opinion, if a system succeed to provide a grammar which allows to write, let us say, around thirty different enough mainstream textual proofs[1] is a huge success.

9.2 Limitations and Future Directions

9.2.1 Ambiguity

As we have seen in §2.5.1 and in Chapters 5, 6 and 7, type plays no part at syntax level (i.e. in the GF implementation of CLM). Therefore, we expect typing information to be provided by the semantics module. This approach is called "semantic type approach". This way the grammar in GF remains cleaner, easier to extend and scale up easily in future.

However, this approach clearly over-generates. For instance, statement such as "Suppose that the set A is positive." is successfully parsed by our GF grammar for CLM. Similarly, we have the problem of notational collisions in symbolic mathematics. For instance, we cannot differentiate operator "+" in "$x + y$ for numbers" and "$A + B$ for sets" as we do not know the type of these variables[2]. Therefore, we translate them as "plus(x, y)" and "plus(A, B)" respectively, which is incorrect (except if the proof system allows overriding). Furthermore, we cannot correctly define precedence for more than one domain as we do not know the types yet.

In contrast, as noted by [Ganesalingam 2009], "typed parsing" of artificial languages in the current state of art is not adequate for the language of mathematics. He also demonstrates that "type casting" and "type coercion" in its usual manner are two grave problems for mathematical language. There is a trade-off between type-safety and expressibility. For instance, if the grammar supports typed parsing, a lot of extra rules are required to cover those categories which cannot be combined by type coercion. Of course the situation becomes worse for a bigger grammar in comparison to the smaller grammar. It is because, it would require more efforts to manage it and scale it up.

Three Possible Solutions:

First: Ganesalingam proposes a convincing account for "mathematical types" (cf. chapter 5 of [Ganesalingam 2009]) that can sufficiently express the mathematical lan-

[1] As regards the content of these proofs, they should be the taken from mainstream text which mathematicians usually write; not the text which is too simple and somewhat rigid.

[2] x, y being of type "number" and A, B being of type "set". However, we only get one parse tree for them. It is because we do not define "+" for numbers and sets separately.

guage. He also propose a novel algorithm for typed parsing (cf. chapter 6 of [Ganesalingam 2009]). However, it is a theoretical work and it seems too early to see if his theoretical model captures well what we find in practice in the language of mathematics.

In the context of MathNat, if we consider implementing this algorithm, it would be in the host system MathNat rather than in the CLM grammar, as we do for various semantic constraints in Chapter 8. If we consider each stage (or module) as a function, then this approach would be four functions:

$$\underbrace{\text{Untyped Syntax} \rightarrow \text{Type checking}}_{\text{Syntax}} \rightarrow \underbrace{\text{Semantic Constraints} \rightarrow \text{MathAbs}}_{\text{Semantics}}$$

In contrast to the MathAbs of our examples which is not type checked, MathAbs in the above diagram is type checked.

Typed parsing algorithm also demonstrate that type checking of mathematical language is a lot of work. It seems that implementing it is perhaps equivalent to designing a simple proof assistant.

Second: An alternate to Ganesalingam's "typed parsing", one can use "dependent types", which GF supports. These dependent types may allow to do typed incremental parsing (on going work). It requires further investigation but the dependent types may suffice in theory. An overall view of the system then would be:

$$\underbrace{\text{Typed Syntax (with dependent types)}}_{\text{Syntax}} \rightarrow \underbrace{\text{Semantic Constraints (if any)} \rightarrow \text{MathAbs}}_{\text{Semantics}}$$

Due to the typed syntax (with dependent types), we do not need to type check the MathAbs in the above diagram. Because typed ill-formed sentences should have already rejected by the typed syntax. However, implementing dependent types for the language of mathematics is not trivial. According to Ranta:

> "[...] parsing intertwined with type checking can be theoretically understood via the use of dependent types [...] But it is not yet a piece of technology that can be called *easy*; building a natural language interface that uses GF's dependent types is still a research project."
>
> Ranta defines *easy* as:
> "*Easy* problems are ones that can be solved by well-known techniques".
>
> [Ranta 2011b]

It may also be a lot of work perhaps equivalent to designing a simple proof assistant.

Third: As we have seen, both solutions above are research problems and may require a lot of work. Therefore, instead of doing it at grammar level (with dependent types) or as a semantic check (with Ganesalingam's typed parsing), it could be postponed to the type checking of MathAbs. When MathAbs is plugged into a proof assistant (abbreviated as PA in diagrams below), we can simply ask for types rather than doing it ourselves (the use of bidirectional arrow is explained near the end of this section):

$$\underbrace{\text{Untyped Syntax}}_{\text{Syntax}} \rightleftarrows \underbrace{\text{Semantic Constraints} \rightleftarrows \text{MathAbs} \rightleftarrows \text{Type checking by PA}}_{\text{Semantics}}$$

In case of an untyped theory such as ZF set theory and proof assistants which are untyped (such as Mizar [Trybulec *et al.* 1973]), we can send types as predicates:

$$\underbrace{\text{Untyped Syntax}}_{\text{Syntax}} \rightleftarrows \underbrace{\text{Semantic Constraints} \rightleftarrows \text{MathAbs} \rightleftarrows \text{Boolean output by PA}}_{\text{Semantics}}$$

However, if we do not want to plug-in a proof assistant and prefer doing it ourselves, we can add another program doing type checking of MathAbs and reject such type-incorrect statements in semantics phase.

$$\underbrace{\text{Untyped Syntax}}_{\text{Syntax}} \rightleftarrows \underbrace{\text{Semantic Constraints} \rightleftarrows \text{MathAbs} \rightleftarrows \text{Type checker of MathAbs}}_{\text{Semantics}}$$

However, in a worst case scenario, we may receive too many parse trees which are translated in too many versions of MathAbs. In this case the program may take too long to process them all. Currently, GF does not allow to reject trees (if we want) on the fly according to our semantics.

As noted by Ranta [Ranta 2011b]: "Because of intertwining parsing and type checking, the usual pipe-lined techniques are insufficient for automatically formalizing arbitrary mathematical text". Therefore, instead of unidirectional arrow: $\rightarrow$, we use bidirectional arrow ($\rightleftarrows$). The first step for an intertwined pipe-line would be to develop an interactive interpretor instead of a compiler.

Finally, whatever method we choose, it mostly remains future work. We currently like the third approach and would like to plug MathAbs to major proof assistants. Until then, we minimize the over-generation with the following measures:

9.2.1.1 Notational Collisions:

1. To differentiate operator "+" in "$x + y$ for numbers" and "$A + B$ for sets", until we have a fully effective type system for MathAbs, we can employ a naïve type system to disambiguate. By naïve, we mean that it would not be complete or perfect (theoretically or practically). It is enough if it works for most common cases. (also a future work).

2. We use different symbol for similar notations to disambiguate (whenever possible), as we already do for multiplication (i.e. $x * y$ instead of xy).

9.2.1.2 Precedence:

It is difficult to get precedence right for a large grammar. CLM supports basic precedence for elementary number theory and the rest is enforced by brackets mainly.

9.2.1.3 New symbols and notations:

New symbols and notations which are not part of formal grammar should be represented by functions. E.g. absolute $\mid a \mid$ could be represented as a function "abs(a)". However, note the limitation of this approach. For instance, if we represent equality $(x = y)$ and addition $(x+y)$ as functions: 'equal(x, y)' and 'plus(x, y)', the system will consider both as expressions and the semantic checks and linguistic features will be applied accordingly.

9.2.2 Proof Checking

As a first prototype, we translated MathAbs to its equivalent first order formulas for the purpose of verification by the automated theorem provers (ATP). Unfortunately, these first-order formulas can hardly be validated because of the reasons mentioned in §4.7. In this sense, we have answered the problem of validation partially.

However, we sufficiently explained the various reasons which makes it a hard problem. We also set some preliminary directions for the future work. For instance, we argued that there is a need of such ATP which can use the *justifications* found in textual proofs. In our opinion, it will allow us to reduce the search space, which in return will help us to check textual proofs. We also would also like to translate MathAbs of some mathematical text in the language of famous theorem provers.

9.2.3 Multilingual CLM

An abstract syntax in GF defines an ontology[3] which is independent of language details, forming *abstract syntax tree*. As we have seen, it is the concrete syntax which contains linguistic information and maps abstract syntax to a language (and back by reversibility). We may have multiple concrete syntax for an abstract syntax, allowing multilingual translation. For instance consider the abstract syntax tree below which is first given in §2.5.1 on page 17:

```
MkProp (MkSymbSubj (MkSymb "sqrt(2)")) Irrational
```

As we have already seen, its concrete syntax in English is:

{s = " ‘`sqrt(2)`‘ is irrational"}.

Similarly, its concrete syntax in French would be:

{s=" ‘`sqrt(2)`‘ est irrationnel"}.

It's grammar for French could be easily defined as shown in figure 9.1. Similarly, its concrete syntax in German and Urdu would be following respectively:

{s=" ‘`sqrt(2)`‘ ist Irrationale"},

{s="ہے غیر نَاطِق ‘`sqrt(2)`‘ "}, *transliteration*: ‘`sqrt(2)`‘ ġaēr nāṫiq hae.

Writing such multilingual grammar from scratch could be expensive in terms of time, effort and expertise. Therefore, we can make this task easy with the use of GF resource library [Ranta 2009a]. GF resource library is linguistic oriented, general purpose grammar which tries to cover the general aspects of a language. It may allows us to develop multilingual CLM with a limited linguistic knowledge.

Finally, we can think of multilingual translation in MathNat with a limitation that the anaphoric resolution of pronouns will not work for all the languages. It is because the anaphoric resolution of pronouns may require some extra linguistic knowledge in some languages, which is not present in English (such as gender in French). Also, when translating from English to French, it will not be possible to correctly translate pronouns. However, when translating from French to English, we'll have no problem.

Also note that, this translation will be performed on GF level (i.e. the abstract syntax of CLM), and hence, MathAbs will not be involved.

[3]Anything that is definable in type theory.

```
1  lincat Proposition, Subject, Type, SymbMath, Pron = {s:Str};
2
3  lin MkProp subj type = {s: subj.s ++ "est" ++ type.s};
4
5  lin MkPronSubj pron = pron ;
6  lin MkSymbSubj symb = symb ;
7
8  lin It = {s="il"};
9  lin MkSymb symb = {s= "‘" ++ symb.s ++ "‘"};
10
11 lin Irrational = {s= "irrationnel"};
```

Figure 9.1: Concrete syntax for French for the abstract syntax given in figure 2.7

9.2.4 Generation from MathAbs

If we suppose that a translation link from MathAbs to various proof assistants is established, we can think of the possibility of trivial text generation for formalized proofs. We can think of a translation in which the rules are translated with some simple rules. For instance, the simplest rule would be to translate all the MathAbs rules in one statements which are separated by full-stops. For instance:

- “let $a, b \in \mathbb{Q}$ •”, can always be translated in a specific statement, such as “let a and b be rational.”.
- “assume A show B •”, can always be translated in “if A then B.”.
- ...

In our opinion, even a very simple and naive translation could be useful.

Appendix A

A Few Selected Examples

Contents

A.1 Irrationality of $\sqrt{2}$ **239**
- A.1.1 First Version: 239
- A.1.2 Second Version: 242
- A.1.3 Third Version: 244
- A.1.4 Fourth Version: 246

A.2 Second Example **247**

A.3 Third Example **248**

A.4 Fourth Example **251**
- A.4.1 First Version: 251
- A.4.2 Second Version: 251

A.5 Fifth Example **252**

A.6 Sixth Example **253**

A.7 Seventh Example **254**

A.8 Eighth Example **254**

Introduction

In this appendix we give a number of example theorem and proofs which are supported by CLM. These examples are also available in "mathnat/examples" directory[1] (if you download the MathNat software).

In each theorem and proof, we resolve the linguistic issues and add the anaphoric referents besides these anaphoric pronouns and references in brackets [. . .]. Also these anaphoric pronouns and references are boldfaced.

In §2.5.2, page 19 and §8.3 on page 189, we have fixed some notations for the *context*. Some of the notations used here could be a bit different. For instance, in quantification (`QNo` $\equiv$ None, QUndecided $\equiv$ –) and in Type (`NoType` $\equiv$ Unknown). We do not display the rule full-stop ($\bullet$) in the MathAbs because each sentence is already separated by new line.

A.1 Irrationality of $\sqrt{2}$

A.1.1 First Version:

1. **"Theorem"** Prove that $\sqrt{2}$ is irrational.

[1] In it, for each example file let us say `1.1.nat`, we have `1.1.output` file containing its output produced by MathNat.

2. **"Proof"** Suppose that $\sqrt{2}$ is a rational number.

3. By the definition of rational numbers, we can assume that $\sqrt{2} = a/b$ where a and b are non-zero integers with no common factor.

4. Thus, $b * \sqrt{2} = a$.

5. **Squaring both sides** [of $b * \sqrt{2} = a$] yields that $2 * b^2 = a^2$ (1).

6. a^2 is even because **it** [a^2] is a multiple of 2.

7. So we can write that $a = 2 * c$, where c is an integer.

8. We get that $2 * b^2 = (2 * c)^2 = 4 * c^2$ by substituting the value of a into equation (**1**) [$2 * b^2 = a^2$].

9. **Dividing both sides** [of $2 * b^2 = a^2$] by 2, yields the result that $b^2 = 2 * c^2$.

10. Thus b is even because 2 is a factor of b^2.

11. If a and b are even then **they** [a and b] have a common factor.

12. It is a contradiction.

13. Therefore, we conclude that $\sqrt{2}$ is an irrational number.

14. This concludes the proof.

Context:

CV: Expression context					
$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
1	$\sqrt{2}$	1	QNo	Explicit	Irrational
2	$\sqrt{2}$	1	QNo	Explicit	Rational
3	a	1	QLet	Implicit	NoType
3	b	1	QLet	Implicit	NoType
3	$\sqrt{2}$	1	QNo	Explicit	Rational
3	a, b	2	QLet	Explicit	Integer
4	$b * \sqrt{2}$	1	QLet	Explicit	NoType
5	$2 * b^2$	1	QLet	Explicit	NoType
6	a^2	1	QLet	Explicit	NoType
6	It [a^2]	1	QLet	Explicit	NoType
6	2	1	QNo	Explicit	NoType
7	c	1	QLet	Implicit	NoType
7	a	1	QLet	Explicit	Integer
7	c	1	QLet	Explicit	Integer
8	$2 * b^2$	1	QLet	Explicit	NoType
8	a	1	QLet	Explicit	Integer
9	2	1	QNo	Explicit	NoType
9	b^2	1	QLet	Explicit	NoType
10	b	1	QLet	Explicit	Integer
10	2	1	QNo	Explicit	NoType
10	b^2	1	QLet	Explicit	NoType
11	a, b	2	QLet	Explicit	Integer
11	they [a, b]	2	QLet	Explicit	Integer
13	$\sqrt{2}$	1	QNo	Explicit	Irrational

ST: Logical Formulas		
$\mathbb{S}_n$	Logical Formula	Statement Type
1	$\neg\sqrt{2} : \mathbb{Q}$	Goal
2	$\sqrt{2} : \mathbb{Q}$	Hypothesis
3	def_rational	Justification
3	$(a, b : \mathbb{Z}) \wedge$ positive$(a) \wedge$ positive$(b) \wedge$ no_cmn_factor$(a, b) \wedge (\sqrt{2} = a/b)$	Hypothesis
4	$b * \sqrt{2} = a$	Deduction
5	squaring_both_sides$(b * \sqrt{2} = a)$	Justification
5	$2 * b^2 = a^2$	Deduction
6	even(a^2)	Deduction
6	multiple_of$(a^2, 2)$	Justification
7	$c : \mathbb{Z} \wedge a = 2 * c$	Hypothesis
8	$2 * b^2 = (2 * c)^2 = 4 * c^2$	Deduction
8	substitution$(a, 2 * b^2 2 = a^2)$	Justification
9	division$(2, 2 * b^2 = a^2)$	Justification
9	$b^2 = 2 * c^2$	Deduction
9	$b^2 = 2 * c^2$	Deduction
10	even(b)	Deduction
10	factor_of$(2, b^2)$	Justification
11	(even$(a) \wedge$ even$(b)) \Rightarrow$ one_cmn_factor(a, b)	Deduction
12	False	Deduction
13	$\neg(\sqrt{2} : \mathbb{Q})$	Deduction

CE: Equations		
$\mathbb{S}_n$	Equation	Reference
3	$\sqrt{2} = a/b$	No
4	$b * \sqrt{2} = a$	No
5	$2 * b^2 = a^2$	1
7	$a = 2 * c$	No
8	$2 * b^2 = (2 * c)^2 = 4 * c^2$	No
9	$b^2 = 2 * c^2$	No

MathAbs:

1. Theorem. show $\neg\sqrt{2} : \mathbb{Q}$
2. Proof. assume $\sqrt{2} : \mathbb{Q}$
3. let $a, b : \mathbb{Z}$ assume $\sqrt{2} = a/b$ assume `positive`(a) $\wedge$ `positive`(b) $\wedge$ `no_cmn_factor`(a, b) `by def Rational_Number`
4. deduce $b * \sqrt{2} = a$
5. deduce $2 * b^2 = a^2$ `1 by oper squaring_both_sides`$(b * \sqrt{2} = a)$
6. deduce `multiple_of`$(a^2$`, 2)`
   deduce `even`(a^2) `by form multiple_of`$(a^2$`, 2)`
7. let $c \in$ `Z` assume $a = 2 * c$
8. deduce $2 * b^2 = (2 * c)^2 = 4 * c^2$ `by oper substitution`$(a, 2 * b^2 = a^2)$
9. deduce $b^2 = 2 * c^2$ `by oper division`$(2, 2 * b^2 = (2 * c)^2 = 4 * c^2)$
10. deduce `factor_of (2,` b^2`)`
    deduce `even`(b) `by form factor_of(2,`b^2`)`
11. deduce `(even`(a) $\wedge$ `even`(b)`)` $\Rightarrow$ `one_cmn_factor`$(a$`,` $b)$
12. show $\perp$ trivial

ASCII Version:

```
1.  "Theorem" Prove that ‘sqrt(2)‘ is irrational.
2.  "Proof" Suppose that ‘sqrt(2)‘ is  a rational number.
3.  By the definition of rational numbers,  we can assume that ‘sqrt(2) = a/b‘ where
   ‘a‘ and ‘b‘ are non-zero integers with no common factor.
4.  Thus, ‘b*sqrt(2) = a‘.
5.  Squaring both sides yields that ‘2*b^2 = a^2‘ (1).
6.  ‘a^2‘ is even because it is a multiple of ‘2‘.
7.  So we can write that ‘a = 2*c‘, where ‘c‘ is an integer.
8.  We get that ‘2*b^2 = (2*c)^2 = 4*c^2‘ by substituting the value of ‘a‘ into equation (1).
9.  Dividing both sides by ‘2‘, yields the result that ‘b^2 = 2*c^2‘.
10. Thus ‘b‘ is even because ‘2‘ is a factor of ‘b^2‘.
11. If ‘a‘ and ‘b‘ are even then they have a common factor.
12. It is a contradiction.
13. Therefore, we conclude that ‘sqrt(2)‘ is an irrational number.
14. This concludes the proof.
```

A.1.2 Second Version:

Its a variant of example 1.

1. **"Theorem."** Prove that $\sqrt{2}$ is an irrational number.

2. **"Proof"** Suppose that $\sqrt{2}$ is a rational number.

3. We suppose that $\sqrt{2} = a/b$ by the definition of rational number, where a and b are non zero integers with no common factor.

4. Thus, $\sqrt{2} * b = a$.

5. We get that $2 * b^2 = a^2 - (i)$ by **squaring both sides** [of $\sqrt{2} * b = a$].

6. Since a^2 and b^2 are non zero integers, we conclude that a^2 is even.

7. By the **last deduction** [of even(a^2)], a is even.

8. We can write that $a = 2*c$ by the definition of even numbers, where c is an integer.

9. We get that $2 * b^2 = (2 * c)^2 = 4 * c^2$ by substituting the value of a into equation *(i)* [of $2 * b^2 = a^2$].

10. **Dividing both sides** [of $2*b^2 = (2*c)^2 = 4*c^2$] by 2, yields the fact that $b^2 = 2*c^2$.

11. Because **it** [b^2] is positive and b is a multiple of 2, we conclude that b^2 is even.

12. If a and b are even, then **they** [a, b] have a common factor.

13. The fact that **they** [a, b] have a common factor implies a contradiction.

Context:

CV: Expression context					
$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
1	$\sqrt{2}$	1	QNo	Explicit	Irrational
2	$\sqrt{2}$	1	QNo	Explicit	Rational
3	$\sqrt{2}$	1	QNo	Explicit	Rational
3	a, b	2	QLet	Explicit	Integer
4	$\sqrt{2} * b$	1	QLet	Explicit	NoType
5	$2 * b^2$	1	QLet	Explicit	NoType
6	a^2, b^2	2	QLet	Explicit	Integer
6	a^2	1	QLet	Explicit	Integer
7	a	1	QLet	Explicit	Integer
6	a	1	QLet	Explicit	Integer
8	c	1	QLet	Explicit	Integer
9	$2 * b^2$	1	QLet	Explicit	NoType
9	a	1	QLet	Explicit	Integer
10	2	1	QNo	Explicit	NoType
10	b^2	1	QLet	Explicit	Integer
11	it $[b^2]$	1	QLet	Explicit	Integer
11	b	1	QLet	Explicit	Integer
11	2	1	QNo	Explicit	NoType
11	b^2	1	QLet	Explicit	Integer
12	a, b	2	QLet	Explicit	Integer
12	they $[a, b]$	2	QLet	Explicit	Integer
13	they $[a, b]$	2	QLet	Explicit	Integer

ST: Logical Formulas		
$\mathbb{S}_n$	Logical Formula	Statement Type
1	$\neg\sqrt{2} : \mathbb{Q}$	Goal
2	$\sqrt{2} : \mathbb{Q}$	Hypothesis
3	def_rational	Justification
3	$(a, b : \mathbb{Z}) \wedge$ positive$(a) \wedge$ positive$(b) \wedge$ no_cmn_factor$(a, b) \wedge (\sqrt{2} = a/b)$	Hypothesis
4	$\sqrt{2} * b = a$	Deduction
5	$2 * b^2 = a^2$	Deduction
5	squaring_both_sides$(b * \sqrt{2} = a)$	Justification
6	$(a^2, b^2 : \mathbb{Z}) \wedge$ positive$(a^2) \wedge$ positive(b^2)	Justification
6	even(a^2)	Deduction
7	last deduction [even(a^2)]	Justification
7	even(a)	Deduction
8	$c : \mathbb{Z} \wedge a = 2 * c$	Hypothesis
9	$2 * b^2 = (2 * c)^2 = 4 * c^2$	Deduction
8	substitution$(a, 2 * b^2 2 = a^2)$	Justification
10	division$(2, 2 * b^2 = (2 * c)^2 = 4 * c^2)$	Justification
10	$b^2 = 2 * c^2$	Deduction
11	positive$(b^2) \wedge$ multiple_of$(b, 2)$	Justification
11	(even(b^2)	Deduction
12	even$(a) \wedge$ even$(b) \Rightarrow$ one_cmn_factor(a, b)	Deduction
13	one_cmn_factor(a, b)	Justification
12	$\perp$	Deduction

CE: Equations		
$\mathbb{S}_n$	Equation	Reference
3	$\sqrt{2} = a/b$	No
4	$\sqrt{2} * b = a$	No
5	$2 * b^2 = a^2$	i
8	$a = 2 * c$	No
9	$2 * b^2 = (2 * c)^2 = 4 * c^2$	No
10	$b^2 = 2 * c^2$	No

MathAbs:

1. Theorem. show $\neg\sqrt{2} : \mathbb{Q}$
2. Proof. assume $\sqrt{2} : \mathbb{Q}$
3. let $a, b : \mathbb{Z}$ assume $\sqrt{2} = a/b$ assume `positive(`a`)` $\wedge$ `positive(`b`)`
   $\wedge$ `no_cmn_factor(`a, b`) by def Rational_Number`
4. deduce $\sqrt{2} * b = a$
5. deduce $2 * b^2 = a^2$ `i by oper squaring_both_sides(`$\sqrt{2} * b = a$`)`
6. deduce $(a^2, b^2 : \mathbb{Z}) \wedge \text{positive}(a^2) \wedge \text{positive}(b^2)$
   deduce $\text{even}(a^2)$
7. deduce `even(`a`) by form even(`a^2`)`
8. let $c : \mathbb{Z}$ assume $a = 2 * c$
9. deduce $2 * b^2 = (2 * c)^2 = 4 * c^2$ `by oper substitution(`$a, 2 * b^2 = a^2$`)`
10. deduce $b^2 = 2 * c^2$ `by oper division(`$2, 2 * b^2 = (2 * c)^2 = 4 * c^2$`)`
11. deduce $\text{positive}(b^2) \wedge \text{multiple_of}(2, b)$
    deduce $\text{even}(b^2)$
12. deduce `(even(`a`)` $\wedge$ `even(`b`))` $\Rightarrow$ `one_cmn_factor(`a`, `b`)`
13. show $\perp$ by oper one_cmn_factor(a, b) trivial

ASCII Version:

```
1  "Theorem" Prove that ‘sqrt(2)‘ is an irrational number.
2  "Proof" Suppose that ‘sqrt(2)‘ is a rational number.
3  We suppose that ‘sqrt(2) = a/b‘ by the definition of rational number, where
4  ‘a‘ and ‘b‘ are non zero integers with no common factor.
5  Thus, ‘sqrt(2)*b = a‘.
6  We get that ‘2*b^2 = a^2‘ - (i) by squaring both sides.
7  Since ‘b^2‘ and ‘a^2‘ are non zero integers, we conclude that ‘a^2‘ is even.
8  By the last deduction, ‘a‘ is even.
9  We can write that ‘a = 2*c‘ by the definition of even numbers, where ‘c‘ is an integer.
10 We get that ‘2*b^2 = (2*c)^2 = 4*c^2‘ by substituting the value of ‘a‘ into equation (i).
11 Dividing both sides by ‘2‘, yields the fact that ‘b^2 = 2*c^2‘.
12 Because it is positive and ‘b‘ is a multiple of ‘2‘, we conclude that ‘b^2‘ is even.
13 If ‘a‘ and ‘b‘ are even, then they have a common factor.
14 The fact that they  have a common factor implies a contradiction.
```

A.1.3 Third Version:

1. **Theorem 43 (PYTHAGORAS' THEOREM)** $\sqrt{2}$ is irrational.
2. **Proof.** If $\sqrt{2}$ is rational, then $a^2 = 2 * b^2 \;-\; (4.3.1)$, where a and b are positive integers with $\gcd(a, b) = 1$.
3. Hence a^2 is even; and therefore a is even.
4. If $a = 2 * c$, then $4 * c^2 = 2 * b^2$.

5. If $4 * c^2 = 2 * b^2$ then $2 * c^2 = b^2$.

6. Consequently, b is even.

7. It is contrary to our hypothesis.

Context:

CV: Expression context					
$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
1	$\sqrt{2}$	1	QNo	Explicit	Irrational
2	$\sqrt{2}$	1	QNo	Explicit	Rational
2	a	1	QLet	Implicit	NoType
2	a^2	1	QLet	Explicit	NoType
2	a, b	2	QLet	Explicit	Integer
2	$\gcd(a, b)$	1	QLet	Explicit	NoType
3	a^2	1	QLet	Explicit	NoType
3	a	1	QLet	Explicit	Integer
4	c	1	QLet	Implicit	NoType
4	a	1	QLet	Explicit	Integer
4	$4 * c^2$	1	QLet	Explicit	NoType
5	$4 * c^2$	1	QLet	Explicit	NoType
5	$2 * c^2$	1	QLet	Explicit	NoType
6	b	1	QLet	Explicit	Integer

ST: Logical Formulas		
$\mathbb{S}_n$	Logical Formula	Statement Type
1	$\neg\sqrt{2} : \mathbb{Q}$	Goal
2	$\sqrt{2} : \mathbb{Q} \Rightarrow ((a, b : \mathbb{Z}) \wedge \text{positive}(a) \wedge \text{positive}(b) \wedge (\gcd(a, b) = 1) \wedge (a^2 = 2 * b^2))$	Hypothesis
3	$\text{even}(a^2)$	Deduction
3	$\text{even}(a)$	Deduction
4	$a = 2 * c \Rightarrow 4 * c^2 = 2 * b^2$	Deduction
5	$4 * c^2 = 2 * b^2 \Rightarrow 2 * c^2 = b^2$	Deduction
6	$\text{even}(b)$	Deduction
7	$\perp$	Deduction

CE: Equations		
$\mathbb{S}_n$	Equation	Reference
2	$a^2 = 2 * b^2$	4.3.1
4	$a = 2 * c$	No
4	$4 * c^2 = 2 * b^2$	No
5	$4 * c^2 = 2 * b^2$	No
5	$2 * c^2 = b^2$	No

MathAbs:

1. Theorem. show $\neg\sqrt{2} : \mathbb{Q}$

2. Proof. let $a, b : \mathbb{Z}$ assume $\text{positive}(a) \wedge \text{positive}(b) \wedge (\gcd(a, b) = 1) \wedge (\sqrt{2} : \mathbb{Q})$ deduce $a^2 = 2 * b^2$

3. deduce $\text{even}(a^2)$ deduce $\text{even}(a)$

4. let c : `NoType` assume $a = 2 * c$ deduce $4 * c^2 = 2 * b^2$

5. assume $4 * c^2 = 2 * b^2$ deduce $2 * c^2 = b^2$

6. deduce $\text{even}(b)$

7. show $\perp$ trivial

ASCII Version:

```
1 "Theorem 43" ‘sqrt(2)‘ is irrational.
2 "Proof" If ‘sqrt(2)‘ is rational, then ‘a^2 = 2*b^2‘ - (4.3.1), where $a$
3 and $b$ are positive integers with ‘gcd(a,b)=1‘.
4 Hence ‘a^2‘ is even; and therefore ‘a‘ is even.
5 If ‘a=2*c‘, then ‘4*c^2 = 2*b^2‘.
6 If ‘4*c^2 = 2*b^2‘ then ‘2*c^2 = b^2‘.
7 Consequently, ‘b‘ is even.
8 It is contrary to our hypothesis.
```

A.1.4 Fourth Version:

1. **"Theorem 43"** $\sqrt{2}$ is irrational.
2. **"Proof"** Suppose that $\sqrt{2}$ is rational.
3. Let a and b be positive integers with $\gcd(a, b) = 1$.
4. Then $a^2 = 2 * b^2 \qquad (4.3.1)$.
5. **By the last equation** [of $a^2 = 2 * b^2$] a^2 is even; and therefore, a is even.
6. So, we can assume that $a = 2 * c$.
7. Substituting the value of a in equation **(4.3.1)** [of $a^2 = 2*b^2$] returns $4*c^2 = 2*b^2$.
8. **Dividing both sides** [of $4 * c^2 = 2 * b^2$] by 2 yields $2 * c^2 = b^2$.
9. **The last equation** [of $2 * c^2 = b^2$] implies that b^2 is even; and therefore, b is even.
10. It is contrary to the hypothesis that $\gcd(a, b) = 1$.

Context:

CV: Expression context					
$\mathbb{S}n$	Object	Number	Quantification	How Declared	Type
1	$\sqrt{2}$	1	QNo	Explicit	Irrational
2	$\sqrt{2}$	1	QNo	Explicit	Rational
3	a, b	2	QLet	Explicit	Integer
3	$\gcd(a, b)$	1	QLet	Explicit	NoType
4	a^2	1	QLet	Explicit	NoType
5	a^2	1	QLet	Explicit	NoType
5	a	1	QLet	Explicit	Integer
6	c	1	QLet	Implicit	NoType
6	a	1	QLet	Explicit	Integer
7	a	1	QLet	Explicit	Integer
7	$4 * c^2$	1	QLet	Explicit	NoType
8	2	1	QNo	Explicit	NoType
8	$2 * c^2$	1	QLet	Explicit	NoType
9	b^2	1	QLet	Explicit	NoType
9	b	1	QLet	Explicit	Integer
10	$\gcd(a, b)$	1	QLet	Explicit	NoType

ST: Logical Formulas		
$\mathbb{S}_n$	Logical Formula	Statement Type
1	$\neg\sqrt{2} : \mathbb{Q}$	Goal
2	$\sqrt{2} : \mathbb{Q} \Rightarrow ((a, b : \mathbb{Z}) \wedge \text{positive}(a) \wedge \text{positive}(b) \wedge (\text{gcd}(a, b) = 1) \wedge (a^2 = 2 * b^2))$	Hypothesis
3	even(a^2)	Deduction
3	even(a)	Deduction
4	$a = 2 * c \Rightarrow 4 * c^2 = 2 * b^2$	Deduction
5	$4 * c^2 = 2 * b^2 \Rightarrow 2 * c^2 = b^2$	Deduction
6	even(b)	Deduction
7	$\perp$	Deduction

CE: Equations		
$\mathbb{S}_n$	Equation	Reference
2	$a^2 = 2 * b^2$	4.3.1
4	$a = 2 * c$	No
4	$4 * c^2 = 2 * b^2$	No
5	$4 * c^2 = 2 * b^2$	No
5	$2 * c^2 = b^2$	No

Its MathAbs and ASCII Version is omitted.

A.2 Second Example

1. **"Theorem"** If p is prime and $p \mid a * b$, then $p \mid a$ or $p \mid b$.

2. **"Proof"** Suppose that p is prime and p divides $a * b$.

3. If $\neg (p \mid a)$ then $\text{gcd}(a, p) = 1$; and therefore, by theorem 24, there are x and y such that $x * a + y * p = 1$ or $x * a * b + y * p * b = b$.

4. We conclude that $p \mid b$ because $p \mid a * b$ and $p \mid p * b$.

5. (an alternate ending: Because $p \mid a * b$ and $p \mid p * b$ we conclude that $p \mid b$).

Context:

CV: Expression context					
$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
1	p	1	QLet	Explicit	Prime
1	a	1	QLet	Implicit	NoType
1	b	1	QLet	Implicit	NoType
1	p	1	QLet	Explicit	Prime
1	p	1	QLet	Explicit	Prime
1	p	1	QLet	Explicit	Prime
2	p	1	QLet	Explicit	Prime
2	p	1	QLet	Explicit	Prime
2	a, b	2	QLet	Explicit	NoType
3	p	1	QLet	Explicit	Prime
3	$\text{gcd}(a, b)$	1	QLet	Explicit	NoType
3	x, y	2	QExist	Explicit	NoType
3	$x * a + y * p$	1	–	Explicit	NoType
3	$x * a * b + y * p * b$	1	–	Explicit	NoType
4	p	1	QLet	Explicit	Prime
4	p	1	QLet	Explicit	Prime
4	p	1	QLet	Explicit	Prime

ST: Logical Formulas		
$\mathbb{S}_n$	Logical Formula	Statement Type
1	p : Prime $\wedge$ divide$(p, a*b)$	Hypothesis
1	divide$(p, a) \vee$ divide(p, b)	Goal
2	p : Prime $\wedge$ divide$(p, a*b)$	Hypothesis
3	$\neg$divide(p, a)	Hypothesis
3	$\gcd(a, p) = 1$	Deduction
3	$\exists(x, y$: NoType$)$ $(x*a + y*p = 1 \vee x*a*b + y*p*b = b)$	Deduction
4	divide(p, b)	Deduction
4	divide$(p, a*b) \wedge$ divide$(p, p*b)$	Deduction

CE: Equations		
$\mathbb{S}_n$	Equation	Reference
3	$\gcd(a, p) = 1$	No
3	$x*a + y*p = 1$	No
3	$x*a*b + y*p*b = b$	No

MathAbs:

1. Theorem. let a, b : NoType let p : Prime assume divide$(p, a*b)$ show divide$(p, a) \vee$ divide(p, b) ;

2. Proof. let p : Prime assume divide$(p, a*b)$

3. assume $\neg$divide(p, a) deduce $\gcd(a, p) = 1$ deduce $\exists(x, y$: NoType$)$ $(x*a + y*p = 1 \vee x*a*b + y*p*b = b)$ by thm theorem_24

4. deduce divide$(p, a*b)$ deduce divide$(p, p*b)$
   deduce divide(p, b) by form divide$(p, a*b)$ by form divide$(p, p*b)$ trivial ;

Note: in line 4, we first deduce the justifications and then use them in the main proof statement.

ASCII Version:

```
"Theorem" If ‘p‘ is prime and ‘p | a*b‘, then ‘p|a‘ or ‘p|b‘.
"Proof" Suppose that ‘p‘ is prime and ‘p‘ divides ‘a*b‘.
If ‘not p | a ‘ then ‘gcd(a,p)=1‘ ; and therefore, by theorem 24, there are ‘x‘ and ‘y‘
such that ‘x*a + y*p = 1‘ or ‘x*a*b + y*p*b = b‘.
We conclude that ‘p|b‘ because ‘p|a*b‘ and ‘p|p*b‘.
```

A.3 Third Example

1. **"Theorem"** Assume that n and m are integers.

2. Suppose that either $n! = 0$ or $m! = 0$.

3. Prove that there exist two integers u and v such that $u*n + v*m = \gcd(n, m)$.

4. **"Proof"** If $n = 0$ then $\gcd(n, m) = m$.

5. In this case we can choose $u := 0$ and $v := 1$.

6. Otherwise if $m = 0$ then we can choose $u := 1$ and $v := 0$.

7. Otherwise there are r and q such that $n = m * q + r$ holds by euclidean division.

8. Let us prove that $\gcd(n, m) = \gcd(m, r)$.

9. We assume that n and m have a common divisor d.

10. d divides r because $r = n - m * q$.

11. If m and r have a common divisor d then **it** [d] divides n.

12. It is trivial that m and r are coprime and $r < m$.

13. So by induction hypothesis there are u' and v' such that $u' * m + v' * r = 1$.

14. It implies that $u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$.

15. Therefore we can choose $u := v'$ and $v := u' - v' * q$.

Context:

CV: Expression context					
$\mathbb{S}_n$	Object	Number	Quantification	How Declared	Type
1	m, n	2	QLet	Explicit	Integer
2	m, n	2	QLet	Explicit	Integer
3	u, v	2	QExistential	Explicit	Integer
3	$u * n + v * m$	1	–	Explicit	Integer
4	n	1	QLet	Explicit	Integer
4	m, n	2	QLet	Explicit	Integer
5	u, v	2	QExistential	Explicit	Integer
6	m	1	QLet	Explicit	Integer
6	u, v	2	QExistential	Explicit	Integer
7	r, q	2	QExistential	Explicit	NoType
7	n	1	QLet	Explicit	Integer
8	m, n	2	QLet	Explicit	Integer
9	m, n	2	QLet	Explicit	Integer
9	d	1	QLet	Explicit	NoType
10	d	1	QLet	Explicit	NoType
10	r	1	QExistential	Explicit	NoType
10	r	1	QExistential	Explicit	NoType
11	m, r	2	–	Explicit	NoType
11	d	2	QLet	Explicit	NoType
11	It(d)	1	QLet	Explicit	NoType
11	n	1	QLet	Explicit	Integer
12	m, r	2	–	Explicit	NoType
12	r	1	QExistential	Explicit	NoType
13	u', v'	2	QExistential	Explicit	NoType
13	$u' * m + v' * r$	1	–	Explicit	NoType
14	$u' * m + v' * (n - m * q)$	1	–	Explicit	NoType
15	u, v	2	QExistential	Explicit	Integer

ST: Logical Formulas		
$\mathbb{S}_n$	Logical Formula	Statement Type
1	$(m, n : \mathbb{Z})$	Hypothesis
2	$(m! = 0 \wedge \neg(n! = 0)) \vee (\neg(m! = 0) \wedge n! = 0)$	Hypothesis
3	$\exists\, u, v : \mathbb{Z}\,(u * n + v * m = \gcd(m, n))$	Goal
4	$(n = 0) \Rightarrow (\gcd(m, n) = m)$	Deduction
5	$u := 0 \wedge v := 1$	Deduction
6	$(\neg(n = 0) \wedge (m = 0)) \Rightarrow (u := 1 \wedge v := 0)$	Deduction
7	$\neg(n = 0) \wedge \neg(m = 0) \wedge \exists\, r, q : \texttt{NoType}\,(n = m * q + r)$	Deduction
8	$\gcd(m, n) = \gcd(m, r)$	Goal
9	$\text{one_cmn_divisor}([m, n], d)$	Hypothesis
10	$(r = n - m * q) \Rightarrow \text{divides}(d, r)$	Deduction
11	$\text{one_cmn_divisor}([m, n], d) \Rightarrow \text{divides}(d, n)$	Deduction
12	$\text{coprime}(m, n) \wedge (r < m)$	Deduction
13	$\exists\, u', v' : \texttt{NoType}\,(u' * m + v' * r = 1)$	Deduction
14	$u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$	Deduction
15	$u := v' \wedge v := u' - v' * q$	Deduction

CE: Equations		
$\mathbb{S}_n$	Equation	Reference
2	$m! = 0$	No
2	$n! = 0$	No
3	$u * n + v * m = \gcd(m, n)$	No
4	$n = 0$	No
4	$\gcd(m, n) = m$	No
6	$m = 0$	No
7	$n = m * q + r$	No
8	$\gcd(m, n) = \gcd(m, r)$	No
10	$r = n - m * q$	No
12	$r < m$	No
13	$u' * m + v' * r = 1$	No
14	$u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$	No

MathAbs:

1. Theorem. let $n, m : \mathbb{Z}$
2. assume $(n! = 0 \wedge \neg m! = 0) \wedge (\neg n! = 0 \wedge m! = 0)$
3. show $\exists(u, v : \mathbb{Z})\ (u * n + v * m = \text{gcd}(n, m))$
4. Proof. $\{$ assume $n = 0$ deduce $\gcd(n, m) = m$
5. trivial by form $u := 0$ by form v := 1 ;
6. assume $\neg(n = 0)$ assume $m = 0$ trivial by form $u := 1$ by form $v := 0$;
7. assume $\neg(n = 0)$ assume $\neg m = 0$ deduce $\exists(r, q : \texttt{NoType})\ (n = m * q + r)$ by def Euclidean_Division
8. show $\gcd(n, m) = \gcd(m, r)$
9. let d : NoType assume $\text{cmn_divisor_of}([d], n, m)$
10. deduce $r = n - m * q$
    deduce $\text{divide}(d, r)$ by form $r = n - m * q$
11. assume $\text{cmn_divisor_of}([d], m, r)$ deduce $\text{divide}(d, n)$

12. deduce $\text{coprime}(m, r) \wedge r < m$
13. deduce $\exists(u', v' : \texttt{NoType})\ (u' * m + v' * r = 1)$ by def Induction_Hypothesis
14. deduce $u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$
15. trivial by form $u := v'$ by form $v := u' - v' * q$ };

Everything else for this example is omitted.

A.4 Fourth Example

A.4.1 First Version:

1. **Theorem.** Assume that m and n are relatively prime integers.
2. Suppose that either $m \neq 0$ or $n \neq 0$.
3. Then prove that there exist two integers u and v such that $u * n + v * m = 1$ holds.
4. **Proof. If** $n = 0$ then $m = 1$ because m and n are coprime.
5. We can choose $u := 0$ and $v := 1$.
6. **Otherwise if** $m = 0$ and $n = 1$ then we can choose $u := 1$ and $v := 0$.
7. **Otherwise** there exist r and q such that $n = m * q + r$ holds by euclidean division.
8. It is obvious that m and r are coprime and $r < m$.
9. So by induction hypothesis there are u' and v' such that $u' * m + v' * r = 1$ holds.
10. It implies that $u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$.
11. So we can choose $u := v'$ and $v := u' - v' * q$.

See figure 4.4 on page 77 for MathAbs. Everything else for this example is omitted.

A.4.2 Second Version:

1. **"Theorem"** Assume that m and n are relatively prime integers.
2. Suppose that either $m! = 0$ or $n! = 0$.
3. Show that there are two integers u and v such that $u * n + v * m = 1$.
4. "Proof" We proceed by case analysis.
5. Case: $n = 0$.
6. In this case $m = 1$ because m and n are coprime.
7. Therefore we can select $u := 0$ and $v := 1$.
8. Case: $m = 0$ and $n = 1$.

9. In this case we can select $u := 1$ and $v := 0$.

10. Case: $n! = 0$ and $m! = 0$.
11. By euclidean division there are r and q such that $n = m * q + r$.
12. It is trivial that m and r are coprime and $r < m$.
13. So there are u' and v' such that $u' * m + v' * r = 1$.
14. It implies that $u' * m + v' * (n - m * q) = v' * n + (u' - v' * q) * m = 1$.
15. So we can select $u := v'$ and $v := u' - v' * q$.

Everything else for this example is omitted.

A.5 Fifth Example

1. **"Theorem"** Let x be a rational number.
2. Prove that $-5 <= |x + 2| - |x - 3| <= 5$.
3. "Proof" We proceed by the case analysis.
4. Case: $x <= -2$.
5. In this case, $|x + 2| - |x - 3| = -(x + 2) - (-(x - 3)) = -5$.
6. Therefore, $-5 <= |x + 2| - |x - 3| <= 5$.

7. Case: $-2 < x <= 3$.
8. Therefore, $|x + 2| - |x - 3| = |x + 2| - (-(x - 3)) = 2 * x - 1$.
9. Because, $-2 < x <= 3$, we get that $-4 < 2 * x <= 6$ and $-5 < 2 * x - 1 <= 5$.
10. So, we conclude that $-5 <= |x + 2| - |x - 3| <= 5$.

11. Case: $x > 3$.
12. In this case, $|x + 2| - |x - 3| = (x + 2) - (x - 3) = 5$.
13. Therefore, $-5 <= |x + 2| - |x - 3| <= 5$.

MathAbs:

1. Theorem. let $x : \mathbb{Q}$
2. show $-5 <= abs(x + 2) - abs(x - 3) <= 5$
3. Proof. $\{$
4. assume $x <= -2$

5. deduce $\text{abs}(x+2) - \text{abs}(x-3) = -(x+2) - (-(x-3)) = -5$
6. deduce $-5 <= \text{abs}(x+2) - \text{abs}(x-3) <= 5$ trivial ;

7. assume $\neg(x <= -2)$ assume $-2 < x <= 3$
8. deduce $\text{abs}(x+2) - \text{abs}(x-3) = x+2-(-(x-3)) = 2*x-1$
9. deduce $-4 < 2*x <= 6 \wedge -5 < 2*x-1 <= 5$ by form $-2 < x <= 3$
10. deduce $-5 <= \text{abs}(x+2) - \text{abs}(x-3) <= 5$ trivial ;

11. assume $\neg(-2 < x <= 3)$ assume $\neg(x <= -2)$ assume $x > 3$
12. deduce $\text{abs}(x+2) - \text{abs}(x-3) = x+2-(x-3) = 5$
13. deduce $-5 <= \text{abs}(x+2) - \text{abs}(x-3) <= 5$ trivial
    } ;

ASCII Version:

```
"Theorem" Let ‘x‘ be a rational number. Prove that ‘-5 <= |x+2| - |x-3| <=5‘.
"Proof" We proceed by the case analysis.
Case: ‘x <= -2‘.
In this case, ‘|x+2| - |x-3| = -(x+2) - (-(x-3)) = -5‘.
Therefore, ‘-5 <= |x+2| - |x-3| <= 5‘.

Case: ‘-2 < x <= 3‘.
Therefore, ‘|x+2| - |x-3| = |x+2| - (-(x-3)) = 2*x - 1‘.
Because, ‘-2 < x <= 3‘, we get that ‘-4 < 2*x <= 6 ‘ and ‘-5 < 2*x -1 <= 5‘.
So, we conclude that ‘-5 <= |x+2| - |x-3| <=5‘.

Case: ‘x>3‘.
In this case, ‘|x+2| - |x-3| = (x+2) - (x-3) = 5‘.
Therefore, ‘-5 <= |x+2| - |x-3| <= 5‘.
```

Everything else for this example is omitted.

A.6 Sixth Example

1. **"Theorem"** Let A and B be two sets. If $A \cup B = A \cap B$ then $A \subseteq B$.
2. **"Proof"** We assume that $A \cup B = A \cap B$.
3. It is sufficient to prove that if $x \in A$ then $x \in B$.
4. We suppose that $x \in A$.
5. Since $A \subseteq A \cup B$, then $x \in A \cup B$.
6. Assume that $x \in A \cap B$ because $A \cup B = A \cap B$.
7. By the fact that $A \cap B \subseteq B$, we conclude that $x \in B$.
8. This concludes the proof.

MathAbs

1. Theorem. let $A, B : Set$ assume $A \cup B = A \cap B$ show $A \subseteq B$
2. Proof. assume $A \cup B = A \cap B$
3. let x : NoType show $x \in A \Rightarrow x \in B$
4. assume $x \in A$
5. deduce $A \subseteq A \cup B$
   deduce $x \in A \cup B$ by form $A \subseteq A \cup B$
6. assume $x \in A \cap B$ by form $A \cup B = A \cap B$
7. deduce $A \cap B \subseteq B$
   deduce $x \in B$ by form $A \cap B \subseteq B$
8. trivial

Everything else for this example is omitted.

A.7 Seventh Example

1. **"Theorem"** If x and y are two positive odd integers then $x^2 + y^2$ is even.
2. **"Proof"** Let x and y be two positive odd integers.
3. We suppose that $x = 2 * a + 1$ and $y = 2 * b + 1$ by the definition of odd numbers.
4. Then $x^2 + y^2 = (2 * a + 1)^2 + (2 * b + 1)^2$.
5. By the last equation we have $x^2 + y^2 = 2 * (2 * (a^2 + b^2 + a + b) + 1)$.
6. Therefore, we deduce that $x^2 + y^2$ is even because it is a multiple of 2.

MathAbs: We give MathAbs in verbatim to show exactly how the MathAbs' output look like:

```
1. "Theorem" let x:Integer let y:Integer
             assume positive(x) land positive(y) land (odd(x) land odd(y))
             show even(x^2 + y^2) ;
2. "Proof" let x:Integer let y:Integer
           assume positive(x) land positive(y) land (odd(x) land odd(y))
3.  let a:NoType let b:NoType assume x = 2*a + 1 land y = 2*b + 1 by def Odd_Numbers
4.  deduce x^2 + y^2 = (2*a + 1)^2 + (2*b + 1)^2
5.  deduce x^2 + y^2 = 2*(2*(a^2 + b^2 + a + b)+ 1)
          by form x^2 + y^2 = (2*a + 1)^2 + (2*b + 1)^2
6.  deduce multiple_of([x^2 + y^2], 2)
    deduce even(x^2 + y^2) by form multiple_of([x^2 + y^2], 2)
    trivial ;
```

Everything else for this example is omitted.

A.8 Eighth Example

See figure 4.1 on page 61.

Bibliography

[Angelov & Ranta 2010] Krasimir Angelov and Aarne Ranta. *Implementing Controlled Languages in GF.* In Norbert E. Fuchs, editor, Controlled Natural Language, volume 5972 of *Lecture Notes in Computer Science*, pages 82–101. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-14418-9_6. 17, 93

[Bagchi & Wells 1998] Atish Bagchi and Charles Wells. *Varieties of Mathematical Prose.* PRIMUS - Problems, Resources and Issues in Mathematics Undergraduate Studies, vol. 8, pages 116–136, 1998. http://www.dean.usma.edu/math/pubs/primus/. 31, 34, 39

[Baker 2009] Andrew Baker. Algebra and Number Theory. Published on website: `http://www.maths.gla.ac.uk/~ajb`. Department of Mathematics, University of Glasgow, January 2009. 31, 34, 35, 36, 38, 39, 40, 41, 47, 48, 50, 51, 52, 53, 56

[Barendregt 2003] Henk Barendregt. *Towards an Interactive Mathematical Proof Language.* In Fairouz D. Kamareddine, editor, Thirty-five Years of Automating Mathematics, volume 28 of *Applied Logic series*, pages 25–36. Kluwer Academic Publishers, 2003. 23

[Bobrow 1964] D. G. Bobrow. *Natural Language Input for a Computer Problem Solving System.* PhD thesis, Massachusetts Institute of Technology (MIT), 1964. 24

[Bundy 1996] Alan Bundy. *Proof Planning.* In B. Drabble, editor, Proceedings of the 3rd International Conference on AI Planning Systems, (AIPS), pages 261–267, 1996. 25, 84

[Burton 2007] David M. Burton. Elementary number theory, 6th edition. McGraw-Hill, 2007. 31, 33, 37, 39, 53, 54, 56, 74

[Chan 2010] Joel Chan. *An Introduction to Proofs.* `www.math.toronto.edu/joel/137/handouts/numbers.pdf`, last accessed: June 2010. MAT 137Y Course, Department of Mathematics, University of Toronto. 54

[Clark 2001] W. Edwin Clark. Elementary abstract algebra. Department of Mathematics, University of South Florida. `http://www.math.usf.edu/~eclark/Elem_abs_alg.pdf`, 2001. 34, 37, 50, 53, 56

[Clark 2002] W. Edwin Clark. Elementary number theory. Department of Mathematics, University of South Florida. `http://www.math.usf.edu/~eclark/elem_num_th_book.pdf`, 2002. 49, 50, 56

[Cramer *et al.* 2010a] M. Cramer, P. Koepke, D. Kühlwein and B. Schröder. *Premise Selection in the Naproche System.* International Joint Conference on Automated Reasoning (IJCAR), 2010. Available at:`http://naproche.net/downloads/2010/IJCAR.pdf`. 26, 84

[Cramer *et al.* 2010b] Marcos Cramer, Bernhard Fisseni, Peter Koepke, Daniel Kühlwein, Bernhard Schröder and Jip Veldman. *The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts*. In Norbert Fuchs, editor, Controlled Natural Language, volume 5972 of *Lecture Notes in Computer Science*, pages 170–186. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-14418-9_11. 26

[Cupillari 2001] Antonella Cupillari. The nuts and bolts of proofs. Academic Press, 2001. 37

[Curry 1961] H. B. Curry. *Some logical aspects of grammatical structure*. In Roman Jakobson, editor, Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics, pages 56–68. American Mathematical Society, 1961. 17, 92

[David *et al.* 2001] René David, Karim Nour and Christophe Raffalli. Introduction à la logique, théorie de la démonstration, cours et exercices corrigés. Dunod, Paris, second edition, 2001. 69

[de Bruijn 1994] N. G. de Bruijn. *The Mathematical Vernacular, a Language for Mathematics with Typed Sets*. In J. H. Geuvers Nederpelt R. P. and R. C. de Vrijer, editors, Selected Papers on Automath, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 865–935, 1994. 23, 58

[Deransart *et al.* 1988] Pierre Deransart, Martin Jourdan and Bernard Lorho. Attribute grammars: Definitions, systems, and bibliography, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988. 4, 17, 92

[Devlin 1993] Keith Devlin. The joy of sets, fundamentals of contemporary set theory, 2nd edition. Springer, 2 edition, 1993. 32

[Erickson & Heeren 2007] Jeff Erickson and Cinda Heeren. *Jeff's class notes on proofs, CS173 - Discrete Mathematical Structures - Fall 2007*. Course Website, 2007. `http://www.cs.uiuc.edu/class/fa07/cs173/notes/proofs.pdf`. 38, 47, 76

[Forsberg & Ranta 2004] Markus Forsberg and Aarne Ranta. *Tool Demonstration: BNF Converter*. In Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, pages 94 – 95, Snowbird, Utah, USA., 2004. Association for Computing Machinery (ACM) and ACM Special Interest Group on Programming Languages (SIGPLAN), Homepage: `http://www.cse.chalmers.se/research/group/Language-technology/BNFC/`. 18, 106

[Forsberg & Ranta 2005] Markus Forsberg and Aarne Ranta. *The Labelled BNF Grammar Formalism*. Technical report, Department of Computing Science, Chalmers University of Technology and the University of Gothenburg, SE-412 96 Gothenburg, Sweden, February 11 2005. Homepage: `http://www.cse.chalmers.se/research/group/Language-technology/BNFC/`. 18, 107

[Forsberg 2007] Markus Forsberg. *Three Tools for Language Processing: BNF Converter, Functional Morphology, and Extract*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, SE–412 96 Göteborg, Sweden, 2007. 180

[Frank & Kamp 1997] A. Frank and H. Kamp. *On context-dependence in modal constructions*. In SALT 7, 1997. 36

[Frege 1879] Gottlob Frege. Bergriffsschrift: eine der arithmetischen nachgebildete formelsprache des reinen denkens. Halle, 1879. 2

[Ganesalingam 2009] Mohan Ganesalingam. *The Language of Mathematics*. PhD thesis, Cambridge University, 2009. 6, 25, 27, 40, 43, 44, 92, 234, 235

[Goldberger 2002] Assaf Goldberger. What are mathematical proofs and why they are important? Math 216 class, University of Connecticut. http://www.math.uconn.edu/~hurley/math315/proofgoldberger.pdf, Spring 2002. 35, 36

[Gordon & Melham 1993] M. J. C. Gordon and T. F. Melham, editors. Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, 1993. 24, 84

[Hackman 2007] Peter Hackman. Elementary number theory. Published on website: http://www.mai.liu.se/~pehac/kurser/TATM54/booktot.pdf , Linköping University, Sweden, November 2007. 31, 33, 35, 39, 47, 51, 52, 54, 55, 56

[Hallgren & Ranta 2000] Thomas Hallgren and Aarne Ranta. *An Extensible Proof Text Editor*. In M. Parigot and A. Voronkov, editors, Logic for Programming and Automated Reasoning (LPAR'2000), LNCS/LNAI, pages 70–84. Springer Verlag, Heidelberg, 2000. 24

[Hardy & Wright 1975] G. H. Hardy and E. M. Wright. An introduction to the theory of numbers. Oxford University Press, Ely House, London, fourth edition, 1975. 6, 31, 34, 35, 36, 38, 46, 47, 50, 52, 55, 56

[Higham 1998] Nicholas J. Higham. Handbook of writing for the mathematical sciences, 2nd edition. Society for Industrial and Applied Mathematics, 1998. ISBN 0898714206. 34

[Huet 1997] Gérard Huet. *Functional Pearl: The Zipper*. Journal of Functional Programming, vol. 7 (5), pages 549–554, 1997. 222

[Humayoun & Raffalli 2010a] Muhammad Humayoun and Christophe Raffalli. *MathAbs: A Representational Language for Mathematics*. In Proceedings of the 8th International Conference on Frontiers of Information Technology, FIT '10, pages 37:1–37:7, New York, NY, USA, 2010. ISBN: 978-1-4503-0342-2, ACM. 59

[Humayoun & Raffalli 2010b] Muhammad Humayoun and Christophe Raffalli. *MathNat - Mathematical Text in a Controlled Natural Language*. In Alexander Gelbukh, editor, Special issue: Natural Language Processing and its Applications, Journal on Research in Computing Science, volume 46, pages 293–310. CICLing-2010, 2010. ISSN 1870-4069. 5

[Jech 2000] Thomas Jech. Set theory, 3rd edition. Springer, 2000. 31, 32, 35, 37, 47, 49, 50, 51, 52, 54, 56

[Jones & Jones 2007] Gareth A. Jones and J. Mary Jones. Elementary number theory. Springer, 2007. 31, 56

[Kamareddine & Wells 2008] Fairouz Kamareddine and J. B. Wells. *Computerizing Mathematical Text with MathLang.* Electronic Notes in Theoretical Computer Science, vol. 205, pages 5–30, 2008. 23, 58, 84

[Kamareddine *et al.* 2007] Fairouz Kamareddine, Manuel Maarek, Krzysztof Retel and J. B. Wells. *Narrative structure of mathematical texts.* In Towards Mechanized Mathematical Assistants (Calculemus 2007 and MKM 2007 Joint Proceedings), volume 4573 of *Lecture Notes in Artifitial Intelligence*, pages 296–311. Springer, 2007. 23, 85

[Kamareddine *et al.* 2008] Fairouz Kamareddine, J.B. Wells and Christoph Zengler. Computerising mathematical text with mathlang. Unpublished but available at: `http://www.cedar-forest.org/forest/papers/drafts/mathlang-coq-short.pdf`, 2008. 23, 85

[Kamp & Reyle 1993] H. Kamp and U. Reyle. From discourse to logic. Kluwer, Dordrecht, 1993. 25

[Knuth 1968] Donald E. Knuth. *Semantics of context-free languages.* Theory of Computing Systems, vol. 2, no. 2, pages 127–145, June 1968. 4, 17, 92

[Lamar *et al.* 2009] Robert Lamar, Fairouz Kamareddine and Joe Wells. *MathLang Translation to Isabelle Syntax.* In 16th Symposium, Calculemus 2009, 8th International Conference, MKM 2009, volume 5625 of *Lecture Notes in Computer Science*, pages 373–388, ISBN: 978–3–642–02613–3, 2009. Springer. 23, 85

[Lamport 1986] Leslie Lamport. Latex: A document preparation system. Addison-Wesley, Reading, Mass., 1986. 6, 41

[Landau 1958] Edmund Landau. Elementary number theory. Translated by Jacob E. GoodMan, Chelsea publishing company, New York, 1958. Translated by Jacob E. GoodMan. 31, 33, 38, 39, 53, 54, 56

[Landau 1966] Edmund Landau. Foundations of analysis, 3rd edition. Translated by F. Steinhardt, Chelsea Publishing Company, New York, 1966. 23, 26, 31, 40, 49, 51, 56, 84

[Lang 1997] Serge Lang. Undergraduate analysis, 2nd edition. Springer, 1997. 31, 56

[Lebl 2010] Jiri Lebl. Basic analysis - introduction to real analysis. `http://www.jirka.org/ra/`. University of Illinois, Urbana-Champaign. Also published on `http://www.lulu.com`, 2010. 33, 35, 56

[LeVeque 1965] W. J. LeVeque. Elementary theory of numbers. Series in introductory mathematics. Addison-Wesley, 1965. 24, 47

[Maarek 2007] Manuel Maarek. *Mathematical documents faithfully computerised: the grammatical and text and symbol aspects of the MathLang framework.* PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, June 2007. 58

[Marlow 2010] Simon Marlow. *Haskell 2010: language and libraries.* Technical report, `http://www.haskell.org/haskellwiki/Language_and_library_specification`, 2010. 17, 179, 180

[Martin-Löf 1984] Per Martin-Löf. Intuitionistic type theory. Bibliopolis, Napoli, 1984. 26, 93

[Montague 1974] Richard Montague. Formal philosophy. Yale University Press, New Haven, 1974. 27

[Mukherjee & Garain 2008] Anirban Mukherjee and Utpal Garain. *A review of methods for automatic understanding of natural language mathematical problems*. Artificial Intelligence Review, vol. 29, pages 93–122, 2008. 10.1007/s10462-009-9110-0. 26

[Neumaier & Schodl 2010] Arnold Neumaier and Peter Schodl. *A Framework for Representing and Processing Abitrary Mathematics*. In Proceedings of the International Conference on Knowledge Engineering and Ontology Development, pages 476–479, 2010. 26

[Norell 2007a] Ulf Norell. *Agda 2*. Web, 2007. 60, 82

[Norell 2007b] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007. 60, 82

[Nour & Raffalli 2003] Karim Nour and Christophe Raffalli. *Simple proof of the completeness theorem for second-order classical and intuitionistic logic by reduction to first-order mono-sorted logic*. Theor. Comput. Sci., vol. 308, no. 1-3, pages 227–237, 2003. 68

[Paulson 1994] Larry Paulson. *Isabelle: A Generic Theorem Prover. (Isabelle Homepage: `http://isabelle.in.tum.de/`)*. Lecture Notes in Computer Science, Springer, vol. 828, 1994. 24, 84

[Peters & Westerstahl 2006] Stanley Peters and Dag Westerstahl. Quantifiers in language and logic. Clarendon Press. Oxford, 2006. ISBN: 0–19–929125–X. 47, 73, 127

[Prawitz 1965] Dag Prawitz. Natural deduction, a proof-theoretical study. Dover Publications, Inc., 1965. 68

[Raffalli 2005] Christophe Raffalli. *The PhoX Proof Assistant*. `http://www.lama.univ-savoie.fr/~RAFFALLI/af2.html`, 2005. 3, 59

[Ranta *et al.* 2010] A. Ranta, K. Angelov, B. Bringert, H. Burden, H. J. Daniels, M. Forsberg, T. Hallgren, H. Hammarström, K. Johannisson, J. Khegai, P. Ljunglöf and P. Mäenpää. *Grammatical Framework, Version 2.7*. `http://www.grammaticalframework.org/`, 1998-2010. 4, 17, 92

[Ranta 1994] Aarne Ranta. *Type Theory and the Informal Language of Mathematics*. In H. Barendregt and T. Nipkow, editors, Types for Proofs and Programs, Lecture Notes in Computer Science, volume 806, pages 352–365. Springer-Verlag, Heidelberg, 1994. 26, 29, 43, 44

[Ranta 1995] Aarne Ranta. *Syntactic categories in the language of mathematics.* In B. Nordström P. Dybjer and J. Smith, editors, Types for Proofs and Programs, Lecture Notes in Computer Science, volume 996, pages 162–182, Heidelberg, 1995. Springer-Verlag. 26

[Ranta 1996] Aarne Ranta. *Context-relative syntactic categories and the formalization of mathematical text.* In S. Berardi and M. Coppo, editors, Types for Proofs and Programs, Lecture Notes in Computer Science, volume 1158, pages 231–248. Springer-Verlag, Heidelberg, 1996. 26, 41

[Ranta 1997] Aarne Ranta. *Structures grammaticales dans le francais mathématique.* Mathématiques, informatique et Sciences Humaines, vol. 138 and 139, pages 5–56 and 5–36 respectively, 1997. 26

[Ranta 2004] Aarne Ranta. *Grammatical Framework: A Type-Theoretical Grammar Formalism.* The Journal of Functional Programming, vol. 14(2), pages 145–189, 2004. 4, 17, 92

[Ranta 2009a] Aarne Ranta. *The GF Resource Grammar Library: A systematic presentation of the library from the linguistic point of view.* Linguistics in Language Technology, 2(2), 2009. 93, 111, 116, 237

[Ranta 2009b] Aarne Ranta. *Grammars as Software Libraries.* In Y. Bertot, G. Huet, J-J. Lévy and G. Plotkin, editors, From Semantics to Computer Science, pages 281–308. Cambridge, Cambridge University Press, 2009. 40

[Ranta 2011a] Aarne Ranta. Grammatical Framework: Programming with Multilingual Grammars. CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth). 4, 17, 92

[Ranta 2011b] Aarne Ranta. *Translating between Language and Logic: What Is Easy and What Is Difficult.* In Nikolaj Bjorner and Viorica Sofronie-Stokkermans, editors, Automated Deduction – CADE-23, volume 6803 of *Lecture Notes in Computer Science*, pages 5–25. Springer Berlin / Heidelberg, 2011. 24, 26, 235, 236

[Roberts 1989] C. Roberts. *Modal subordination and pronominal anaphora in discourse.* Linguistics and Philosophy, vol. 12, pages 683–721, 1989. 36

[Russell 1998] Bertrand Russell. Autobiography. Routledge, 1998. 2

[Sacerdoti Coen 2009] Claudio Sacerdoti Coen. *A User Interface for a Mathematical System that Allows Ambiguous Formulae.* Electron. Notes Theor. Comput. Sci., vol. 226, pages 67–87, January 2009. 7, 42

[Seki *et al.* 1991] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii and Tadao Kasami. *On multiple context-free grammars.* Theoretical Computer Science, vol. 88, no. 2, pages 191 – 229, 1991. 17

[Simon 1988] D. L. Simon. *Checking natural language proofs.* In 9th International Conference on Automated Deduction (CADE-9), volume 310 of *Lecture Notes in Computer Science*. Springer, 1988. 24

[Simon 1990] D. L. Simon. *Checking Number Theory Proofs in Natural Language*. PhD thesis, University of Texas at Austin, 1990. 24, 25

[Taylor 2010] Joseph L. Taylor. Foundations of analysis. Self published, Department of Mathematics, University of Utah, `http://www.math.utah.edu/~taylor/foundations.html`, 2010. 40

[Team 2010] The Coq Development Team. *Coq Reference Manual*. INRIA-Rocquencourt et CNRS-ENS, Lyon, France. Coq Homepage: `http://coq.inria.fr`, 8.3 edition, 2010. 24, 60, 82, 84

[Thévenon 2005] Patrick Thévenon. *Validation of Proofs Using PhoX*. Electronic Notes in Theoretical Computer Science, vol. 140, pages 55 – 66, 2005. Proceedings of the Second Workshop on Computational Logic and Applications (CLA 2004). 3, 59

[Thévenon 2006] Patrick Thévenon. *Vers un assistant à la preuve en langue naturelle*. PhD thesis, Université de Savoie, France, 2006. 3, 59

[Trybulec *et al.* 1973] Andrzej Trybulec, Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Robert Milewski, Adam Naumowicz, Josef Urban and Others. *Mizar Home Page. www. mizar. org*. 1973. 23, 84, 176, 236

[van Dalen 1980] Dirk van Dalen. Logic and structure. Springer Verlag, Berlin, 1980. 69

[Wenzel 1999] Markus Wenzel. *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents*. In Y. Bertot G. Dowek A. Hirschowitz C. Paulin and L. Thery, editors, 12th International Conference on Theorem Proving in Higher Order Logics, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999. 23

[Whitehead & Russell 1962] Alfred North Whitehead and Bertrand Russell. Principia mathematica. Cambridge University Press, 1962. 2

[Wiedijk 2003] Freek Wiedijk. *Formal Proof Sketches*. In Types for Proofs and Programs: TYPES 2003, volume 3085 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2003. 23, 85

[Wikipedia 2010] Wikipedia. *Example: $\sqrt{2}$ is irrational*. `http://en.wikipedia.org/wiki/Mathematical_proof#Proof_by_contradiction`, June 2010. last accessed. 54

[Zinn 2004] Claus Zinn. *Understanding Informal Mathematical Discourse*. PhD thesis, Institut für Informatik, Universität Erlangen-Nürnberg, 2004. Arbeitsberichte des Instituts für Informatik, Band 37, Nr.4, ISSN 0344-3515. 6, 25, 37, 84

[Zinn 2006] Claus Zinn. *Supporting the formal verification of mathematical texts*. Journal of Applied Logic, vol. 4, no. 4, pages 592 – 621, 2006. Towards Computer Aided Mathematics. 25, 36