COMPUTERISING MATHEMATICAL TEXT

Fairouz Kamareddine, Joe Wells, Christoph Zengler and
Henk Barendregt

Reader: Serge Autexier

1 BACKGROUND AND MOTIVATION

Mathematical texts can be computerised in many ways that capture differing amounts of the mathematical meaning. At one end, there is document imaging, which captures the arrangement of black marks on paper, while at the other end there are proof assistants (e.g., Mizar, Isabelle, Coq, etc.), which capture the full mathematical meaning and have proofs expressed in a formal foundation of mathematics. In between, there are computer typesetting systems (e.g., LaTeX and Presentation MathML) and semantically oriented systems (e.g., Content MathML, OpenMath, OMDoc, etc.). In this paper we advocate a style of computerisation of mathematical texts which is flexible enough to connect the different approaches to computerisation, which allows various degrees of formalisation, and which is compatible with different logical frameworks (e.g., set theory, category theory, type theory, etc.) and proof systems. The basic idea is to allow a man-machine collaboration which weaves human input with machine computation at every step in the way. We propose that the huge step from informal mathematics to fully formalised mathematics be divided into smaller steps, each of which is a fully developed method in which human input is minimal.

Let us consider the following two questions:

1. What is the relationship between the logical foundations of mathematical reasoning and the actual practice of mathematicians?

2. In what ways can computers support the development and communication of mathematical knowledge?

1a Logical Foundations

Our first question, of the relationship between the practice of mathematics and its logical foundations, has been an issue for at least two millennia. Logic was already influential in the study and development of mathematics since the time of the ancient Greeks. One of the main issues was already known by Aristotle, namely that for a logical/mathematical proposition Φ,

- given a purported proof of Φ, it is not hard to check whether the argument really proves Φ, but
- in contrast, if one is asked to find a proof of Φ, the search may take a very long time (or even go forever without success) even if Φ is true.

Aristotle used logic to reason about everything (mathematics, farming, medicine, law, etc.). A formal logical style of deductive reasoning about mathematics was introduced in Euclid's geometry [Heath, 1956].

The 1600s saw a increase in the importance of logic. Researchers like Leibniz wanted to use logic to address not just mathematical questions but also more esoteric questions like the existence of God. In the 1800s, the need for a more precise style in mathematics arose, because controversial results had appeared in analysis [Kamareddine *et al.*, 2004a]. Some controversies were solved by Cauchy's precise definition of convergence in his *Cours d'Analyse* [Cauchy, 1821], others benefited from the more exact definition of real numbers given by Dedekind [Dedekind, 1872], while at the same time Cantor was making a tremendous contribution to the formalisation of set theory and number theory [Cantor, 1895; Cantor, 1897] and Peano was making influential steps in formalised arithmetic [Peano, 1889] (albeit without an extensive treatment of logic or quantification).

In the last decades of the 1800s, the contributions of Frege made the move toward formalisation much more serious. Frege found

> "*. . . the inadequacy of language to be an obstacle; no matter how unwieldy the expressions I was ready to accept, I was less and less able, as the relations became more and more complex, to attain precision*"

Based on this understanding of a need for greater preciseness, Frege presented *Begriffsschrift* [Frege, 1879], the first formalisation of logic giving logical concepts via symbols rather than natural language. "Begriffsschrift" is the name both of the book and of the formal system the book presents. Frege wrote:

> "*[Begriffsschrift's] first purpose, therefore, is to provide us with the most reliable test of the validity of a chain of inferences and to point out every presupposition that tries to sneak in unnoticed, so that its origin can be investigated.*"

Later, Frege wrote the *Die Grundlagen der Arithmetik* and *Grundgesetze der Arithmetik* [Frege, 1893; Frege, 1903; van Heijenoort, 1967] where he argued that mathematics is a branch of logic and described arithmetic in the Begriffsschrift. Grundgesetze was the culmination of Frege's work on building a formal foundation for mathematics.

One of the major issues in the logical foundations of mathematics is that the naive approach of Frege's Grundgesetze (and Cantor's earlier set theory) is inconsistent. Russell discovered a paradox in Frege's system (and also in Russell's own system) that allows proving a contradiction, from which everything can be proven, including all the false statements [Kamareddine *et al.*, 2004a]. The need to build

logical foundations for mathematics that do not suffer from such paradoxes has led to many diverging approaches. Russell invented a form of *type theory* which he used in the famous *Principia Mathematica* [Whitehead and Russel, 1910–1913]. Others have subsequently introduced many kinds of type theories and modern type theories are quite different from Russell's [Barendregt *et al.*, 2013]. Brouwer introduced a different direction, that of *intuitionism*. Later, ideas from intuitionism and type theory were combined, and even extended to cover the power of classical logic (which Brouwer's intuitionism rejects). Zermelo followed a different direction in introducing an axiomatisation of set theory [Zermelo, 1908], later extended by Fraenkel and Skolem to form the well known Zermelo/Fraenkel (ZF) system. In yet another direction, it is possible to use category theory as a foundation. And there are other proposed foundations, too many to discuss here.

Despite the variety of possible foundations for mathematics, in practice real mathematicians do not express their work in terms of a foundation. It seems that most modern mathematicians tend to *think* in terms that are compatible with ZFC (which is ZF extended with the Axiom of Choice), but in practice they almost never write the full formal details. And it is quite rare for mathematicians to do their thinking while regarding a type theory as the foundation, even though type theories are among the most thoroughly developed logical foundations (in particular with well developed computer proof software systems). Instead, mathematicians write in a kind of *common mathematical language* (CML) (sometimes called a *mathematical vernacular*), for a number of reasons:

- Mathematicians have developed conventional ways of using nouns, adjectives, verbs, sentences, and larger chunks of text to express mathematical meaning. However, the existing logical foundations do not address the convenient use of natural language text to express mathematical meanings.

- Using a foundation requires picking one specific foundation, and any foundation commits to some number of fixed choices. Such choices include what kinds of mathematical objects to take as the primitives (e.g., sets, functions, types, categories, etc.), what kinds of logical rules to use (e.g., "natural deduction" vs. "logical deduction", whether to allow the full power of classical logic, etc.), what kinds of syntax and semantics to allow for logical propositions (first-order vs. higher-order), etc. Having made some initial choices, further choices follow, e.g., for a set theory one must then choose the axioms (Zermelo/Fraenkel, Tarski/Grothendieck, etc.), or for a type theory the kinds of types and the typing rules (Calculus of Constructions, Martin-Löf, etc.). Fixed choices make logical foundations undesirable to use for three reasons:

  - Much of mathematics can be built on top of all of the different foundations. Hence, committing to a particular foundation would seem to unnecessarily limit the applicability of mathematical results.

  - The details of how to build some mathematical concepts can vary quite a bit from foundation to foundation. Issues that cause difficulty include

how to handle "partial functions", induction, reasoning modulo equations, etc. Since these issues *can* be handled in all foundations, mathematicians tend to see the low-level details of these issues as inessential and uninteresting, and are not willing to write the low-level details.

– Some mathematics only works for some foundations. Hence, for a mathematician to develop the specialised expertise needed to express mathematics in terms of one particular foundation would seem to unnecessarily limit the scope of mathematics he/she could address. A mathematician is happy to be reassured by a mathematical logician that what they are doing *can* be expressed in some foundation, but the mathematician usually does not care to work out precisely how. Moreover there is no universal agreement as to which is the best logical foundation.

- In practice, formalising a mathematical text in any of the existing foundations is an extremely time-consuming, costly, and mentally painful activity. Formalisation also requires special expertise in the particular foundation used that goes far beyond the ordinary expertise of even extremely good mathematicians. Furthermore, mathematical texts formalised in any of the existing foundations are generally structured in a way which is radically different from what is optimal for the human reader's understanding, and which is difficult for ordinary mathematicians to use. (Some proof software systems like Mizar, which is based on Tarski/Grothendieck set theory, attempt to reduce this problem, and partially succeed.) What is a single step in a usual human-readable mathematical text may turn into a multitude of smaller steps in a formalised version. New details completely missing from the human-readable version may need to be woven throughout the entire text. The original text may need to be reorganised and reordered so radically that it seems like it is almost turned inside out in the formal version.

So, although mathematics was a driving force for the research in logic in the 19th or 20th century, mathematics and logic have kept a distance from each other. Practising mathematicians do not use mathematical logic and have for centuries done most mathematical work outside of the strict boundaries of formal logic.

1b Computerisation of Mathematical Knowledge

Our second question, of how to use mechanical computers to support mathematical knowledge, is more recent but is unavoidable since automation and computation can provide tremendous services to mathematics. There are also extensive opportunities for combining progress in logic and computerisation not only in mathematics but also in other areas: bio-informatics, chemistry, music, etc.

Mechanical computers have been used from their beginning for mathematical purposes. Starting in the 1960s, computers began to play a role in handling not just computations, but abstract mathematical knowledge. Nowadays, computers can represent mathematical knowledge in various ways:

- Pixel map images of pages of mathematical articles may be stored on the computer. While useful, it is very difficult for computer programs to access the semantics of mathematical knowledge presented this way [Autexier *et al.*, 2010]. Even keyword searching is hard, since OCR (Optical Character Recognition) must be performed and high quality OCR for mathematical texts is an area with significant research challenges rather than a proven technology (e.g., there is great difficulty with matrices [Kanahori *et al.*, 2006]).

- Typesetting systems like LaTeX or TeX_{MACS} [van der Hoeven, 2004], can be used with mathematical texts for editing them and formatting them for viewing or printing. The document formats of these systems can also be used for storage and archiving. Such systems provide good defaults for visual appearance and allow fine control when needed. They support commonly needed document structures and allow custom structures to be created, at least to the extent of being able to produce the correct visual appearance.

  Unfortunately, unless the mathematician is amazingly disciplined, the logical structure of symbolic formulas is not directly represented. Furthermore, the logical structure of mathematics as embedded in natural language text is not represented at all. This makes it difficult for computer programs to access document semantics because fully automated discovery of the semantics of natural language text still performs too poorly to use in practical systems. Even human-assisted semi-automated semantic analysis of natural language is primitive, and we are aware of no such systems with special support for mathematical text. As a consequence, there is generally no computer support for checking the correctness of mathematics represented this way or for doing searching based on semantics (as opposed to keywords).

- Mathematical texts can be written in more semantically oriented document representations like OpenMath [Abbott *et al.*, 1996] and OMDoc [Kohlhase, 2006], Content MathML [W3C, 2003], etc. There is generally support for converting from these representations to typesetting systems like LaTeX or Presentation MathML in order to produce readable/printable versions of the mathematical text. These systems are 1) better than the typesetting systems at representing the knowledge in a computer-accessible way, and 2) can represent Some aspects of the semantics of symbolic formulas.

- There are software systems like *proof assistants* (also called *proof checkers*, these include Coq [Team, 1999–2003], Isabelle [Nipkow *et al.*, 2002], NuPrL [Constable and others, 1986], Mizar [Rudnicki, 1992], HOL [Gordon and Melham, 1993], etc.) and automated *theorem provers* (Boyer-Moore, Otter, etc.), which we collectively call *proof systems*. Each proof system provides a formal language (based on some foundation of logic and mathematics) for writing/mechanically checking logic, mathematics, and computer software. Work on computer support for formal foundations began in the late 1960s with work by de Bruijn on Automath (AUTOmating MATHe-

matics) [Nederpelt *et al.*, 1994]. Automath supported automated checking of the full correctness of a mathematical text written in Automath's formal language. Generally, most proof systems support checking full correctness, and it is possible in theory (although not easy) for computer programs to access and manipulate the semantics of the mathematical statements.

Closely related to proof systems, we find proof development/planning systems (e.g., ΩMEGA [Siekmann *et al.*, 2002; Siekmann *et al.*, 2003] and λCLAM [Bundy *et al.*, 1990]) which are mathematical assistant tools that support proof development in mathematical domains at a user-friendly level of abstraction. An additional advantage of these systems is that they focus on *proof planning* and hence can provide different styles of proof development.

Unfortunately, there are great disadvantages in using proof systems. First, all of the problems mentioned for logical foundations in section 1a are incurred, e.g., the enormous expense of formalisation. Furthermore, one must choose a specific proof system (Isabelle, Coq, Mizar, PVS, etc.) and each software system has its own advantages and pitfalls and takes quite some time to learn. In practice, some of these systems are only ever learnt from a "master" in an "apprenticeship" setting. Most proof systems have no meaningful support for the mathematical use of natural language text. A notable exception is Mizar, which however requires the use of natural language in a rigid and somewhat inflexible way. Most proof systems suffer from the use of proof tactics, which make it easier to construct proofs and make proofs smaller, but obscure the reasoning for readers because the meaning of each tactic is often *ad hoc* and implementation-dependent. As a result of these and other disadvantages, ordinary mathematicians do not generally read mathematics written in the language of a proof system, and are usually not willing to spend the effort to formalise their own work in a proof system.

- Computer algebra systems (CAS: e.g., Maxima, Maple, Mathematica, etc.) are widely used software environments designed for carrying out computations, primarily symbolic but sometimes also numeric. Each CAS has a language for writing mathematical expressions and statements and for describing computations. The languages can also be used for representing mathematical knowledge. The main advantage for such a language is integration with a CAS. Typically, a CAS language is not tied to any specific foundation and has little or no support for guaranteeing correctness of mathematical statements. A CAS language also typically has little or no support for embedded natural language text, or for precise control over typesetting. So a CAS is often used for calculating results, but these results are usually converted into some other language or format for dissemination or verification. Nonetheless, there are useful possibilities for using a CAS for archiving and communicating mathematical knowledge.

It is important to build a bridge between more than one of the above categories of ways of representing mathematical knowledge, and to make easier (without re-

quiring) the partial or full formalisation of mathematical texts in some foundation. In this paper, we discuss two approaches aimed at achieving this: Barendregt's approach [Barendregt, 2003] towards an interactive mathematical proof mode and Kamareddine and Wells' MathLang approach [Kamareddine and Wells, 2008] towards the gradual computerisation of mathematics.

2 INTRODUCTION

Mathematical assistants are workstations running a program that verifies the correctness of mathematical theorems, when provided with enough evidence. Systems for automated deduction require less evidence or even none at all; proof-checkers on the other hand require a fully formalised proof. In the pioneering systems Automath[1] (of N.G. de Bruijn, based on dependent type theory), and Mizar (of Andrzej Trybulec based on set-theory), proofs had to be given ready and well. On the other hand for systems like NuPrl, Isabelle, and Coq, the proofs are obtained in an interactive fashion between the user and the proof-checker. Therefore one speaks about an interactive mathematical assistant. The list of statements that have to be given to such a checker, the *proof-script*, is usually not mathematical in nature, see e.g. table 8. The problem is that the script consists of fine-grained steps of what should be done, devoid of any mathematical meaning. Mizar is the only system having a substantial library of certified results in which the proof-script is mathematical in nature. Freek Wiedijk [Wiedijk, 2006] speaks of the declarative style of Mizar. In [de Bruijn, 1987] a plea was given to use a mathematical vernacular for formalising proofs.

This paper discusses two approaches influenced by de Bruijn's mathematical vernacular: MathLang [Kamareddine and Wells, 2008] and MPL (*Mathematical Proof Language*) [Barendregt, 2003]. These approaches aim to develop a framework for computerising mathematical texts which is flexible enough to connect the different approaches to computerisation, which allows various degrees of formalisation, and which is compatible with different logical frameworks (e.g., set theory, category theory, type theory, etc.) and proof systems. Both approaches aim to bridge informal mathematics and formalized mathematics via automatic translations into the formalised language of interactive proof-assistants. In particular:

- MPL aims to provide an interactive script language for an interactive proof assistant like Coq, that is declarative and hence mathematical in flavor.[2]
- MathLang is embodied in a computer representation and associated software tools, and its progress and design are driven by the need for computerising representative mathematical texts from various branches of mathematics.

At this stage, MPL remains a script language and has no associated software tools. MathLang on the other hand, supports entry of original mathematical texts either

[1] `<www.cs.kun.nl/~freek/aut>`

[2] A similar approach to MPL is found in Isar[3], with an implemented system.

in an XML format or using the TEX_{MACS} editor and these texts are manipulated by a number of MathLang software tools. These tools provide methods for adding, checking, and displaying various information *aspects*. One aspect is a kind of weak type system that assigns categories (term, statement, noun (class), adjective (class modifier), etc.) to parts of the text, deals with binding names to meanings, and checks that a kind of grammatical sense is maintained. Another aspect allows weaving together mathematical meaning and visual presentation and can associate natural language text with its mathematical meaning. Another aspect allows identifying chunks of text, marking their roles (theorem, definition, explanation, example, section, etc.), and indicating relationships between the chunks (A uses B, A contradicts B, A follows from B, etc.). Software tool support can use this aspect to check and explain the overall logical structure of a text.

Further aspects are being designed to allow adding additional formality to a text such as proof structure and details of how a human-readable proof is encoded into a fully formalised version (previously [Kamareddine *et al.*, 2007b; Lamar, 2011] we used Mizar and Isabelle but here, for the first time we develop the MathLang formalisation into Coq). [Kamareddine and Wells, 2008] surveyed the status of the MathLang project up to November 2007. This paper picks on from that survey, fills in a number of formalisation and implementation gaps and creates a formalisation path via MathLang into Coq. We show for the first time how the DRa information can be used to automatically generate proof skeletons for different theorem provers, we formalise and implement the textual order of a text and explain how it can be derived from the original text. Our proposed generic algorithm (for generating the proof skeleton which depends on the original mathematical text and the desired theorem prover), is highly configurable and caters for arbitrary theorem provers. This generic algorithm as well as all the new algorithms and concepts we present here, are implemented in our software tool. We give hints for the development of an algorithm which is able to convert parts of a CGa annotated text automatically into the syntax of a special theorem prover.

To test our approaches we specify using MPL a feasible interactive mathematical proof development for Newman's Lemma and we create the complete path of encoding in and formalising through MathLang, for the first chapter of Landau's book "Grundlagen der Analysis". For Newman's Lemma in MPL, we show that the *declarative* interactive mathematical mode is more pleasant than the *operational* mode of Coq. For Landau's chapter in MathLang, we show that the entire path from the informal text into the fully formalised Coq text is much easier to construct and comprehend in MathLang than in Coq. For this, we show how the plain text document of Landau's chapter, can be easily annotated with categories and mathematical roles and how a Coq and a Mizar proof skeletons can be automatically generated for the chapter. We then use hints to convert parts of the annotated text of Landau's first chapter into Coq. Both the Coq proof skeleton and the converted parts into Coq, simplified the process of the full formalisation of the first chapter of Landau's book in Coq.

Although in this paper we only illustrate MPL and MathLang for Coq, the proposed approaches should work equally well for other proof systems (indeed, we have previously illustrated MathLang for Mizar and Isabelle [Kamareddine *et al.*, 2007a; Kamareddine *et al.*, 2007b]).

3 THE GOALS OF MATHLANG AND MPL

Sections 1a and 1b described issues with the practice of mathematics: the difficulty for the normal mathematician in directly using a formal foundation, and the disadvantages of the various computer representations of mathematics. To address these issues, we set out to develop two new mathematical languages, so that texts written in CML (the common mathematical language, expressed either with pen and paper, or LaTeX) is written instead in a way that satisfies these goals:

1. A MathLang/MPL text should support the usual features of CML: natural language text, symbolic formulas, images, document structures, control over visual presentation, etc. And the usual computer support for editing such texts should be available.

2. It should be possible to write a MathLang/MPL text in a way that is significantly less ambiguous than the corresponding CML text. A MathLang/MPL text should *somehow* support representing the text's mathematical semantics and structure. The support for semantics should cover not just individual pieces of text and symbolic formulas but also the entire document and the document's relationship to other documents (to allow building connected libraries). The degree of formality in representing the mathematical semantics should be flexible, and at least one choice of degree of formality should be both inexpensive *and* useful. There should be some automated checking of the well-formedness of the mathematical semantics.

3. The structure of a MathLang/MPL text should follow the structure of the corresponding CML, so that the experience of reading and writing MathLang/MPL should be close to that of reading and writing CML. This should make it easier for an author to see and have confidence that a MathLang/MPL text correctly represents their intentions. Thus, if any foundational formal systems are used in MathLang/MPL, then the latter should somehow adapt the formal systems to the needs of the authors and readers, rather than requiring the authors and readers to adapt their thinking to fit the rigid confines of any existing foundations.

4. The structure of a MathLang/MPL text should make it easier to support further post-authorship computer manipulations that respect its mathematical structure and meaning. Examples include semantics-based searches, computations via computer algebra systems, extraction of proof sketches (to be completed into a full formalisation in a proof system), etc.

5. A particular important case of the previous point is that MathLang/MPL should support (but not require) interfacing with proof systems so that a MathLang/MPL text can contain full formal details in some foundation and the formalisation can be automatically verified.

6. Authoring of a MathLang/MPL text should not be significantly harder for the ordinary mathematician than authoring LaTeX. Features that the author does not want (such as formalisation in a proof system) should not require any extra effort from an author.

7. The design of MathLang/MPL should be compatible with (as yet undetermined) future extensions to support additional uses of mathematical knowledge. Also, the design of MathLang/MPL should make it easy to combine with existing languages (e.g., OMDoc, TEX_{MACS}). This way, MathLang/MPL might end up being a method for extending an existing language in addition to (or instead of) a language on its own.

None of the previously existing representations for mathematical texts satisfies our goals, so we have been developing new techniques. In this paper we discuss where we are with both MathLang and MPL.

MathLang/MPL are intended to support different degrees of formalisation. Furthermore, for those documents where full formalisation is a goal, we intend to allow this to be accomplished in gradual steps. Some of the motivations for varying degrees of formalisation have already been discussed in sections 1a and 1b. Full formalisation is sometimes desirable, but also is often undesirable due to its expense and the requirement to commit to many inessential foundational details. Partial formalisation can be desirable for various reasons; as examples, it has the potential to be helpful with automated checking, semantics-based searching and querying, and interfacing with computer algebra systems (and other mathematical computation environments). In both our languages, MathLang and MPL, partial formalisation can be carried out to different degrees. For example:

- The abstract syntax trees of symbolic formulas can be represented accurately. This is usually missing when using systems like LaTeX or Presentation MathML, while more semantically oriented systems provide this to some degree. This can provide editing support for algebraic rearrangements and simplifications, and can help interfacing with computer algebra systems.

- The mathematical structure of natural language text can be represented in a way similar to how symbolic formulas are handled. Furthermore, mixed text and symbols can be handled. This can help in the same way as capturing the structure of symbolic formulas can help. Nearly all previous systems do not support handling natural language text in this way.

- A weak type system can be used to check simple grammatical conditions without checking full semantic sensibility.

- Justifications (inside proofs and between formal statements) can be linked (without necessarily always indicating precisely how they are used). Some examples of potential uses of this feature include the following:
  - Extracting only those parts of a document that are relevant to specific results. (This could be useful in educational systems.)
  - Checking that each instance of apparently circular reasoning is actually handled via induction.
  - Calculating proof gaps as a first step toward fuller formalisation.
- If one commits to a foundation (or in some cases, to a family of foundations), one can start to use more sophisticated type systems in formulas and statements for checking more aspects of well-formedness.
- And there are further possibilities.

4 AN OVERVIEW OF MATHLANG

The design of MathLang is gradually being refined based on experience testing the use of MathLang for representative mathematical texts. Throughout the development, the design is tested by evaluating encodings of real mathematical texts, during which issues and difficulties are encountered, which lead to new needs being discovered and corresponding design adjustments. The design includes formal rules for the representation of mathematical texts, as well as patterns and methodology for entering texts in this representation, and supporting software.

The choice of mathematical texts for testing is primarily oriented toward texts that represent the variety of mathematical writing by ordinary mathematicians rather than texts that represent the interests of formalists and mathematical logicians. Much of the testing has been with pre-existing texts. In some cases, texts that have previously been formalised by others were chosen in order to compare representations, e.g., *A Compendium of Continuous Lattices* [Gierz *et al.*, 1980] of which at least 60% has been formalised in Mizar [Rudnicki, 1992], and Landau's *Foundations of Analysis* [Landau, 1951] which was fully formalised in Automath [van Benthem Jutting, 1977a]. In other cases, texts of historical value which are known to have errors were chosen to ensure that MathLang's design will not exclude them, e.g., Euclid's *Elements* [Heath, 1956]. Other texts were chosen to exercise other aspects of MathLang. Authoring new texts has also been tested.

In addition to the design of MathLang itself, there has been work on relating a MathLang text to a fully formalised version of the text. Using the information in the CGa and DRa aspects of a MathLang text, [Kamareddine *et al.*, 2007b; Retel, 2009] developed a procedure for producing a corresponding Mizar document, first as a proof sketch with holes and then as a fully completed proof. [Lamar, 2011] attempted to follow suit with Isabelle. In this paper, we make further progress in completing the path in MathLang in order to reach full formalisation and we introduce a third theorem prover (Coq) as a test bed for MathLang (in addition to Mizar and Isabelle). We develop the proof skeleton idea presented

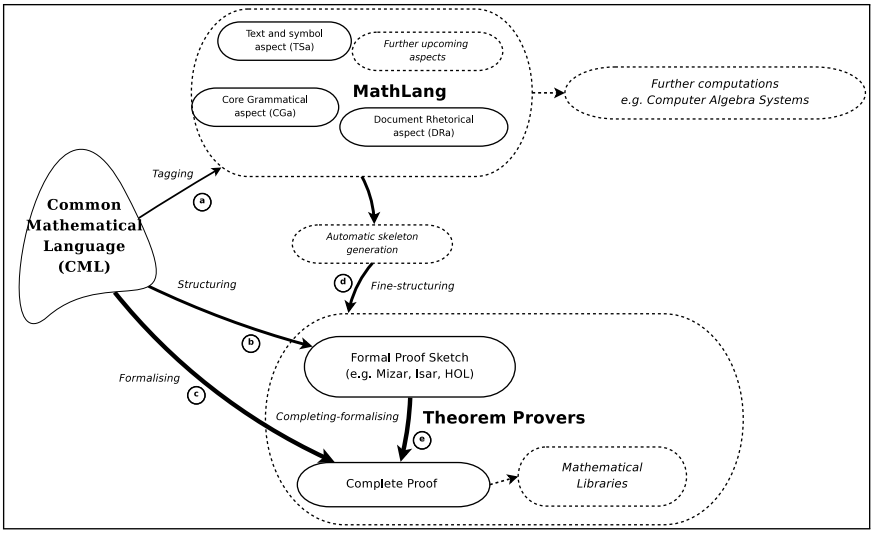


Figure 1. Overall situation of work in MathLang

earlier in [Kamareddine *et al.*, 2007b] specifically for Mizar, into an automatically generated proof skeleton in a choice of theorem provers (including Mizar, Isar and Coq). To achieve this, we give a generic algorithm for proof skeleton generation which takes the required prover as one of its arguments. We also give hints for the development of a generic algorithm which automatically converts parts of a CGa annotated text into the syntax of the theorem prover it is given as an argument.

Figure 1 (adapted from [Kamareddine *et al.*, 2007b]) diagrams the overall current situation of work on MathLang. In the rest of this paper, we discuss the aspects CGa, TSa, and DRa in more detail, we introduce the generic automatic proof skeleton generator and how parts of the CGa annotated text can be formalised into a theorem prover. We also discuss interfacing MathLang with Coq.

4a The Core Grammatical aspect (CGa)

The Core Grammatical aspect (CGa) [Kamareddine *et al.*, 2004c; Kamareddine *et al.*, 2006; Maarek, 2007] is based on the Weak Type Theory (WTT) of Nederpelt [Nederpelt, 2002] whose metatheory was established by Kamareddine [Kamareddine and Nederpelt, 2004]. WTT in turn was heavily inspired by the Mathematical Vernacular (MV) [de Bruijn, 1987].

In WTT, a document is a *book* which is a sequence of *lines*, each of which is a pair of a *sentence* (a *statement* or a *definition*) and a *context* of facts (*declarations* or *statements*) assumed in the sentence. WTT has four ways of introducing names. A *definition* introduces a name whose scope is the rest of the book and associates the name with its meaning. A name introduced by a definition can have *parameters* whose scope is the body of the definition. A *declaration* in a context introduces a name (with no parameters) whose scope is only the current line. Fi-

nally, a *preface* gives names whose scope is the document; names introduced by prefaces have parameters but unlike definitions their meanings are not provided (and thus presumed to be given externally to the document). Declarations, definitions, and statements can contain *phrases* which are built from *terms*, *sets*, *nouns*, and *adjectives*. Using the terminology of object-oriented programming languages, nouns act like *classes* and adjectives act like *mixins* (a special kind of function from classes to classes). WTT uses a weak type system with types like `noun`, `set`, `term`, `adjective`, `statement` `definition`, `context`, and `book` to check basic well-formedness. Sets are used when something is definitely known to be a set and the richer structure of a noun is not needed, and terms are used for things that are not sets (and sometimes for sets in cases where the type system is too weak).

Although WTT provides many useful ideas, the definition of WTT has many limitations. The many different ways of introducing names are too complicated and awkward. WTT provides no way to indicate which statements are used to justify other statements and in general does not deal with *proofs* and logical correctness. WTT provides no ways to present the structure of a text to human readers; there is no way of grouping statements and identifying their mathematical/discourse roles such as *theorem*, *lemma*, *conjecture*, *proof*, *section*, *chapter*. WTT provides no way to give human names to statements (e.g., "Newman's Lemma"). WTT provides no way to use in one document concepts defined in another document.

The Core Grammatical aspect (CGa) was shaped by repeated experiences of annotating mathematical texts. CGa simplifies difficult aspects of WTT, and enhances the nouns and adjectives of WTT with ideas from object-oriented programming so that nouns are more like classes and adjectives are more like mixins. In CGa, the different kinds of name-introducing forms of WTT are unified; all definitions by default have indefinite forward scope and a local scope operator allows local definitions. The basic constructs of CGa are the *step* and the *expression*. The tasks handled in WTT by books, prefaces, lines, declarations, definitions, and statements are all represented as steps in CGa. A step can be a *block* $\{s_1, \ldots, s_n\}$, which is merely a sequence of steps. A step can be a *local scoping* $s_1 \triangleright s_2$, which is a pair of steps s_1 and s_2 where the definitions and declarations of s_1 are restricted in scope to s_2 and the assertions of s_1 are assumptions of s_2. A step can also be a *definition*, a *declaration*, or an expression (which asserts a truth). Expressions are also used for the bodies of definitions and inside the types in declarations. The possibilities for expressions include uses of defined identifiers, identifier declarations, and *noun descriptions*. A noun description allows specifying *characteristics* of a class of entities. For example,
$\{M : set;\ y : natural\_number;\ x : natural\_number;\ \in(x, M)\} \triangleright = (+(x, y), +(y, x))$
is an encoding of this (silly) CML text:

> "*Given that* $\mathfrak{M}$ *is a set,* y *and* x *are natural numbers, and* x *belongs to* $\mathfrak{M}$*, it holds that* $x + y = y + x$*.*"

This example assumes that earlier in the document there are declarations like:

$\ldots;\ \in(\texttt{term}, \texttt{set}) : \texttt{stat};\ =(\texttt{term}, \texttt{term}) : \texttt{stat};\ \texttt{natural\_number} : \texttt{noun};$
$+\,(\texttt{natural\_number}, \texttt{natural\_number}) : \texttt{natural\_number};\ \ldots$

Here, `M`, `y`, `x`, $\in$, $=$, and $+$ are *identifiers*[4] while `term`, `set`, `stat`, and `noun` are *keywords* of CGa. The semicolon, colon, comma, parentheses, braces, and right triangle ($\triangleright$) symbols are part of the syntax of CGa. The statements like $\in$(`term`, `set`) : `stat` are declarations; this example declares $\in$ to be an operator that takes two arguments, one of type `term` and one of type `set`, and yields a result of type `stat` (statement). The statement `M` : `set` is an abbreviation for `M`() : `set` which declares the identifier `M` to have zero parameters.

CGa uses grammatical/linguistic/syntactic *categories* (also called *types*) to make explicit the grammatical role played by the elements of a mathematical text. In the above example, we see the category expressions `term`, `set`, `stat`, `noun`, and `natural_number`. In fact, the category expression `natural_number` acts as an abbreviation for `term`(`natural_number`), and `term`, `set`, and `noun` are abbreviations for `term`(`Noun` {}), `set`(`Noun` {}), and `noun`(`Noun` {}), which all use the *uncharacterised* noun description `Noun` {}. A noun description is of the form `Noun` s and describes a class of entities with *characteristics* (declared operations and true facts) defined by the step s. The arguments of the category constructors `term`, `set`, and `noun` are expressions which evaluate to noun descriptions. The category `term`(e) describes individual entities belonging to the class described by the noun expression e, and the category `set`(e) describes any set of such entities. The category `noun`(e) describes any noun which defines all the operations described by e with the same types. So in the above example, the abbreviation `term` is the type of all mathematical entities, the abbreviation `set` is the type of any set, `noun` is the type of any noun (and specifies no characteristics for it), and `natural_number` is the type of any mathematical entity having the characteristics described by the noun `natural_number`.[5] The behaviour of nouns in CGa is similar to that of *classes* in object-oriented programming languages. CGa also has *adjectives* which are like object-oriented *mixins* and act as functions from nouns to nouns. These linguistic levels and syntactic elements are summarised in the following definitions.

DEFINITION 1 (Linguistic levels). The syntax of CGa is based on a hierarchy of the five different linguistic levels given below. Elements from $\mathfrak{I}$ and $\mathfrak{C}$ are part of $\mathfrak{E}$, expressions are part of the phrases of $\mathfrak{P}$ and steps $\mathfrak{S}$ are built from phrases.

1. Identifier level $\mathfrak{I}$
2. Category level $\mathfrak{C}$
3. Expression level $\mathfrak{E}$
4. Phrase level $\mathfrak{P}$
5. Step level $\mathfrak{S}$

DEFINITION 2 (Syntactic elements). The syntactic elements at each level are:

1. At identifier level: term identifiers $I^{\mathcal{T}}$, set identifiers $I^{\mathcal{S}}$, noun identifiers $I^{\mathcal{N}}$, adjective identifiers $I^{\mathcal{A}}$ and statement identifiers $I^{\mathcal{P}}$.
2. At category level: term categories $\mathcal{T}$, set categories $\mathcal{S}$, noun categories $\mathcal{N}$, adjective categories $\mathcal{A}$, statement categories $\mathcal{P}$ and declaration categories $\mathcal{D}$.

[4] Our current implementation only allows ASCII characters in identifiers, but we plan to support any graphic Unicode characters.

[5] CGa has other mechanisms that allow specifying additional characteristics of the noun `natural_number` separate from its declaration, and we assume in this example that this is done.

3. At expression level: declaration expressions **DEC**, instantiation expressions **INST**, description expressions **DSC**, refinement expressions **REF** and the self expression **SEL**.
4. At phrase level: sub refinement phrases **SUB**, definition phrases **DEF**, declaration expressions **DEC** and statement expressions $\mathbb{P}$.
5. At step level: local scoping steps **LOC**, block steps **BLO** and each phrase of $\mathfrak{P}$ is a basic step.

For details of the rules of CGa see [Kamareddine *et al.*, 2006; Maarek, 2007]. Here, it is crucial to mention the following colour codings for these CGa categories: term set noun adjective statement definition declaration step context.

The types of CGa are more sophisticated than the *weak types* of WTT and allow tracking which operations are meaningful in some additional cases. Although CGa's types are more powerful than WTT's, there are still significant limitations. One limitation is that higher-order types are not allowed. For example, although CGa allows the type $(\texttt{term}, \texttt{term}) \to \texttt{term}$, which is the type of an operator that takes two arguments of type `term` and returns a result of type `term`, CGa does not allow using the type $((\texttt{term}) \to \texttt{term}, \texttt{term}) \to \texttt{term}$, which would be the type of an operator that takes another operator as its first argument. Higher-order types can be awkwardly and crudely emulated in CGa by encapsulation with noun types, but this emulation does not work well due to the fact that CGa's type polymorphism is shallow, which is another significant limitation. To work around the weakness of CGa's type polymorphism, in practice we find ourselves often giving entities the type `term` instead of a more precise type. We continue to work on making the type system more flexible without making it too complex. It is important to understand that the goal of CGa's type system is *not* to ensure full correctness, but merely to check whether the reasoning parts of a document are coherently built in a sensible way. CGa provides a kind of *grammar* for well-formed mathematics with grammatical categories and allows checking for basic well-formedness conditions (e.g., the origin of all names/symbols can be tracked).

The design of CGa is due to Kamareddine, Maarek and Wells [Kamareddine *et al.*, 2006]. The implementation of CGa is due to Maarek [Maarek, 2007].

4b The Text and Symbol aspect (TSa)

The Text and Symbol aspect (TSa) [Kamareddine *et al.*, 2004b; Kamareddine *et al.*, 2007a; Maarek, 2007; Lamar, 2011] allows integrating normal typesetting and authoring software with the mathematical structure represented with CGa. TSa allows weaving together usual mathematical authoring representations such as LaTeX, XML, or TEX_{MACS} with CGa data. Thanks to a notion of *souring* rules (called "souring" because it does the opposite of *syntactic sugar*), TSa allows the structure of the mathematical text to follow that of the CML text as conceived by the mathematician. TSa allows interleaving pieces of CGa with pieces of CML in the form of mixtures of natural language, symbolic formulas, and formatting instructions for visual presentation. The interleaving can be at any level of granularity: meanings can be associated at a coarse grain with entire paragraphs or

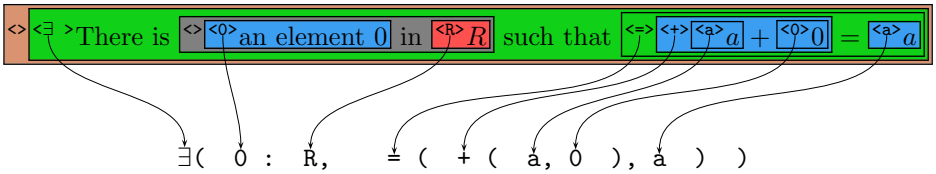


Figure 2. Example of CGa encoding of CML text

sections, or at a fine grain with individual words, phrases, and symbols. Arbitrary amounts of mathematically uninterpreted text can be included. The TSa representation is inspired by the XQuery/XPath Data Model (XDM) [WC3, 2007] used for representing the information content of XML documents. In TSa, a document d is built from the empty document ($[]$) by sequencing (d_1, d_2) and labelling $(\ell\langle d\rangle)$.

For example, the CML text and its CGa representation given in figure 2 could be represented in TSa by the following fine-grained interleaving of CGa[6] and LaTeX:

```
“There is #1 such that #2.”
  ⟨∃⟨“#1 in #2”⟨:⟨“an element $0$”⟨0⟩, “$R$”⟨R⟩⟩⟩,
     “$#1 = #2$”⟨=⟨“#1 + #2”⟨+⟨“a”⟨a⟩, “0”⟨0⟩⟩⟩, “a”⟨a⟩⟩⟩⟩
```

This example (see [Kamareddine *et al.*, 2007a]) uses the abbreviation that ℓ stands for $\ell\langle[]\rangle$. For example, “a”⟨a⟩ actually stands for “a”⟨a⟨[]⟩⟩.

Associated with TSa are methods for extracting separately the CGa and the typesetting instructions or other visual representation. E.g., from the TSa above can be extracted the following TSa representation of just the CGa portion:

$$\exists\langle:\langle 0, R\rangle, =\langle+\langle a, 0\rangle, a\rangle\rangle$$

The CGa portion of this text can be type checked and used for processing that needs to know the mathematical meaning of the text. Similarly, the following pieces of LaTeX can also be extracted:

```
“There is #1 such that #2.”
  ⟨“#1 in #2”⟨“an element $0$”, “$R$”⟩,
   “$#1 = #2$”⟨“#1 + #2”⟨“a”, “0”⟩, “a”⟩⟩
```

This tree of LaTeX typesetting instructions can be further flattened for actual processing by LaTeX into a string such as:

```
“There is an element $0$ in $R$ such that $a + 0 = a$.”
```

The idea of the TSa representation is independent of the visual formatting language used. Although we use LaTeX in our example here, in our implementations so far we have used the TEX_{MACS} internal representation and also XML.

As part of using TSa to interleave CGa and more traditional natural language and typesetting information, we needed to develop techniques for handling certain challenging CML formations where the mathematical structure and the CML representation do not nicely match. For example, in the text $0+a0 = a0 = a(0+0) =$

[6] The representation shown here omits type/category annotations that we usually include with the CGa identifiers used in the TSa representation.

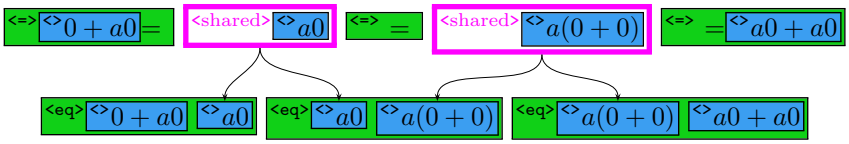


Figure 3. Example of using souring in TSa to support sharing

$a0 + a0$, the terms $a0$ and $a(0 + 0)$ are each shared between two equations. Most formal representations would require either duplicating these shared terms, like for example $0 + a0 = a0 \wedge a0 = a(0 + 0) \wedge a(0 + 0) = a0 + a0$, or explicitly abstracting the shared terms. To allow the TSa representation to be as close to CML as possible, we instead solve this by using "*souring*" annotations in the TSa representation [Kamareddine *et al.*, 2007a]. These annotations are a third kind of node label used in TSa, in addition to the CGa and formatting labels. Souring annotations are used to extract the correct mathematical meaning and the nice visual presentation in the CML style. For the above example, see figure 3.

We have developed more sophisticated annotations that can handle more complicated cases of sharing of terms between equations. Souring annotations have also been developed to support several other common CML formulations. Support for folding and mapping over lists allows using forms like $\forall a, b, c \in S.P$ as shorthand for $\forall a \in S. \forall b \in S. \forall c \in S.P$ and $\{a, b, c\}$ as shorthand for $\{a\} \cup (\{b\} \cup (\{c\} \cup \emptyset))$. We have not yet developed folding that is sophisticated enough to handle ellipsis (...) as in CML formulations like the next example (from [Sexton and Sorge, 2006]):

$$f\underbrace{[x, \ldots, x]}_{n+1 \text{ arguments}} = \frac{f^{(n)}(x)}{n!}$$

We have implemented a user interface as an extension of the TEXMACS editor for entering the TSa MathLang representation. The author can use mouse and keyboard commands to annotate CML text entered in TEXMACS with boxes representing the CGa grammatical categories in order to assign CGa identifiers and explicitly indicate mathematical meanings. The user interface allows displaying either a pure CML view which hides the TSa and CGa information, a pure CGa view, or various combined views including a view like that of figure 2. This interface allows adding souring annotations like those of figure 3. We plan to develop techniques for not just pairing a single CML presentation with its CGa meaning, but also allowing multiple parallel visual presentations such as multiple natural languages (not just English), both natural language and symbolic formulas, and presentations in different symbolic notations. We plan also to develop better software support to aid in semi-automatically converting existing CML texts into MathLang via TSa and CGa.

The design of TSa is due to Kamareddine, Maarek, and Wells with contributions by Lamar to the souring rules [Kamareddine *et al.*, 2007a; Maarek, 2007; Lamar, 2011]. The implementation is primarily by Maarek [Maarek, 2007] with contributions from Lamar [Lamar, 2011].

4c The Document Rhetorical aspect (DRa)

The Document Rhetorical aspect (DRa) [Kamareddine *et al.*, 2007c; Retel, 2009; Zengler, 2008] supports identifying portions of a text and expressing the relationships between them. Any portion of text (e.g., phrase, step, block, etc.) can be given an identity. Many kinds of relationships can be expressed between identified pieces of text. E.g., a chunk of text can be identified as a "theorem", and another can be identified as the "proof" of that theorem. Similarly, one chunk of text can be a "subsection" or "chapter" of another. Given these relationships, it becomes possible to do computations to check whether all dependencies are identified, to check whether the relationships are sensible or problematic (and whether therefore the author should be warned), and to extract and explain the logical structure of a text. Dependencies identified this way have been used in generating formal proof sketches and identifying the proof holes that remain to be filled. This paper presents further formalisation and implementation of notions related to DRa.

DRa is a system for attaching annotations to mathematical documents that indicate the roles played by different parts of a document. DRa assumes the underlying mathematical representation (which can be the MathLang aspects CGa or TSa) has some mechanism for identifying document parts.

Some DRa annotations can be unary predicates on parts; these include annotations indicating ordinary document sectioning roles such as *part*, *chapter*, *section*, etc. (like the sectioning supported by LaTeX, OMDoc, DocBook, etc.) and others indicating special mathematical roles such as *theorem*, *lemma*, *proof*, etc.

Other DRa annotations can be binary predicates on parts; these include such relationships between parts as "*justifies*", "*uses*", "*inconsistent with*", and "*example of*". Regarding the annotation of justifications, remember that a CML text is usually incomplete: a mathematical thought process makes jumps from one interesting point to the next, skipping over details. This does not mean that many mistakes can occur; these details are usually so obvious for the mathematician that a couple of words are enough (e.g., "*apply theorem 35*"). The mathematician knows that too many details hinder concentration. To allow MathLang text to be close to the CML text, DRa allows informal justifications, which can be seen as *hints* about which statements would be used in the proof of another statement.

Figure 4 gives an example (taken from [Kamareddine *et al.*, 2007b] and implemented by Retel [Retel, 2009]) where the mathematician has identified parts of the text (indicated by letters A through I in the figure). Figure 5, shows the underlying mathematical representation of some example DRa annotations for the example in figure 4. Here, the mathematician has given each identified part a structural (e.g., chapter, section, etc.) and/or mathematical (e.g., lemma, corollary, proof, etc.) rhetorical role, and has indicated the relation between wrapped chunks of texts (e.g., justifies, uses, etc.). Note that all the DRa annotations are represented as triples; this allows using the machinery of RDF [WC3, 2004] (a W3C standard that is aimed at the "semantic web") to represent and manipulate them.

The DRa structure of a text can be represented as a tree (which is exactly the

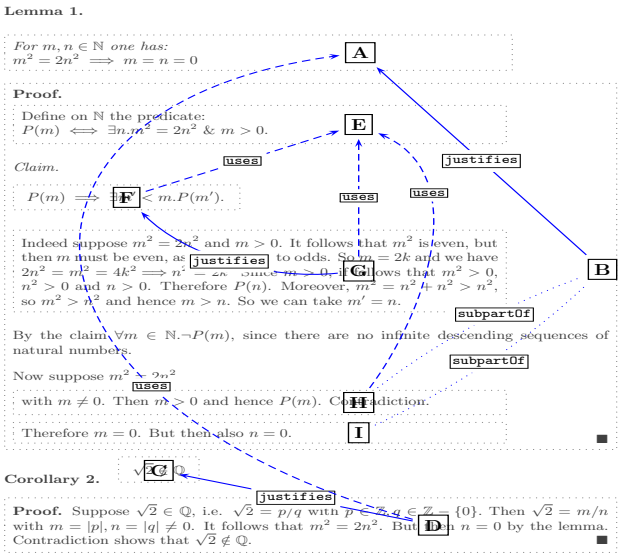


Figure 4. Wrapping/naming chunks of text and marking relationships in DRa

(A, hasMathematicalRhetoricalRole, lemma)	(B, justifies, A)
(E, hasMathematicalRhetoricalRole, definition)	(D, justifies, C)
(F, hasMathematicalRhetoricalRole, claim)	(D, uses, A)
(G, hasMathematicalRhetoricalRole, proof)	(G, uses, E)
(B, hasMathematicalRhetoricalRole, proof)	(F, uses, E)
(H, hasMathematicalRhetoricalRole, case)	(H, uses, E)
(I, hasMathematicalRhetoricalRole, case)	(H, caseOf, B)
(C, hasMathematicalRhetoricalRole, corollary)	(H, caseOf, I)
(D, hasMathematicalRhetoricalRole, proof)	

Figure 5. Example of DRa relationships between chunks of text in figure 4

tree of the XML representation of the DRa annotated MathLang document). Due to the tree structure of a DRa annotated document, we refer to an annotated part of a text as a DRa node. We see an example of such a DRa node in figure 8. The role of this node is declaration and its name is decA. Note that the content of a DRa node is the user's CGa and TSa annotation. In the DRa annotation of a document, there is a dedicated root node (the Document node) where each top-level DRa node is a child of this root node. In figure 6, we see a tree consisting of 10 nodes. The root node (labelled Document) has four children nodes and five grandchildren nodes (which are all children of B).

We distinguish between proved nodes (*theorem*, *lemma*, etc.) with a solid line in the picture and unproved nodes (*axiom*, *definition*, etc.) with a broken line. We introduce this distinction because with the current implementation of DRa the user has the possibility to create its own mathematical and structural roles. Since we want to check a DRa annotated document for validity, the information whether a node is to be proved or not is important. For example such information would result in an error if someone tries to prove an unproved node e.g. by proving

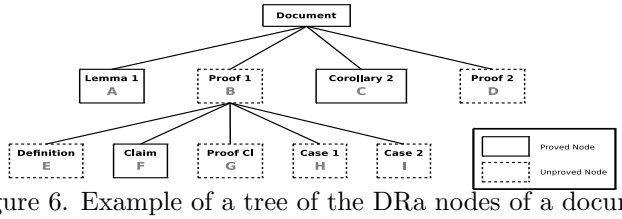


Figure 6. Example of a tree of the DRa nodes of a document

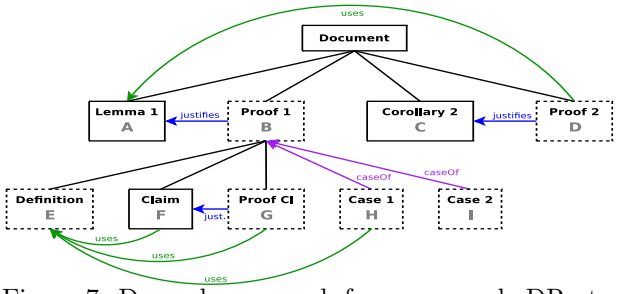


Figure 7. Dependency graph for an example DRa tree

a definition or an axiom. When document D_2 references document D_1 it can reference the root node of document D_1 to include all of its mathematical text.

In figure 4 one can see, that there are four top-level nodes: A, B, C and D, representing respectively lemma 1, a proof of lemma 1, corollary 2 and a proof of corollary 2. The proof of lemma 1 has five children: E, F, G, H, I representing respectively the definition of the predicate, a claim, the proof of the claim, case 1 and case 2. The visual representation of this tree can be seen in figure 6.

By traversing the tree in pre-order we derive the original linear order of the DRa nodes of the text. Pre-order means that the traversal starts with the root node and for each node we first visit the parent node before we visit its children. It is important to mention that we have also an order of the nodes at the same level from left to right. This means that we enumerate the children of a node form 1 to n and process them in this way. In the example of figure 6, the pre-order would yield the order A, B, E, F, G, H, I, C, D.

The DRa implementation can automatically extract a dependency graph (as seen in figure 7) that shows how the parts of a document are related.

Textual Order

To be able to examine the proper structure of a DRa tree we introduce the concept of textual order between two nodes in the tree. The concept of textual order is a modification of the logical precedence presented in [Kamareddine *et al.*, 2007c]. In what follows, we formalise this concept of order and show how it can be used to automatically generate a proof skeleton. The textual order expresses the dependencies between parts of the text. For example if a node A uses a part of a node B, then in a sequence of reasoning steps, B has to be before A. In order to

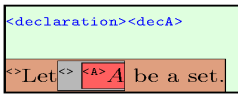


Figure 8. An example for a single DRa node

formally define textual order, we introduce some notions for DRa nodes.

Recall that the content of a DRa node is its CGa and TSa part and that we have nine kinds of CGa annotations: term set noun adjective statement declaration definition step context . A DRa node can also have further DRa nodes as children (e.g. B in figure 6 has the children E, F, G, H and I). We give different sets for a DRa node n. All these sets can be automatically generated from the user's CGa and TSa annotations of the text. Table 1 defines these sets and gives examples for the CGa annotated text in figure 9 which is the definition of the subset relation.

Figure 9. CGa annotations for the definition of the subset relation

Set	Description	Example of Fig. 9
$\mathcal{T}(n)$	$\{x \mid x$ is part of n and x is annotated as term$\}$	$\{x\}$
$\mathcal{S}(n)$	$\{x \mid x$ is part of n and x is annotated as set$\}$	$\{A,\ B\}$
$\mathcal{N}(n)$	$\{x \mid x$ is part of n and x is annotated as noun$\}$	$\{\}$
$\mathcal{A}(n)$	$\{x \mid x$ is part of n and x is annotated as adjective$\}$	$\{\}$
$\mathcal{ST}(n)$	$\{x \mid x$ is part of n and x is annotated as statement$\}$	$\{A \subset B,\ x \in A,\ x \in B,\ x \in A \Longrightarrow x \in B,\ \forall x(x \in A \Longrightarrow x \in B)\}$
$\mathcal{DC}(n)$	$\{x \mid \exists q$ part of n, q is annotated as declaration and x is the declared symbol of $q\}$	$\{x\}$
$\mathcal{DF}(n)$	$\{x \mid \exists q$ part of n, q is annotated as definition and x is the defined symbol of $q\}$	$\{\subset\}$
$\mathcal{SP}(n)$	$\{x \mid x$ is part of n and x is annotated as step$\}$	$\{A \subset B \Longleftrightarrow \forall x(x \in A \Longrightarrow x \in B)\}$
$\mathcal{C}(n)$	the set of all parts of n annotated as context	$\{\}$
$\mathcal{ENV}(n)$	$\{x \mid \exists m \neq n, m$ is a node in the pre-order path from the root node to the node n, x is a part of m, and x is annotated as statement$\}$	

Table 1. Sets for a DRa node n and examples

Let us give further examples of $\mathcal{DC}(n)$ and $\mathcal{DF}(n)$. In section 4a we had the following example of a list of declarations (call it ex):

$\ldots;\ \in(\texttt{term},\texttt{set}) : \texttt{stat};\ =(\texttt{term},\texttt{term}) : \texttt{stat};\ \texttt{natural\_number} : \texttt{noun};$
$+\,(\texttt{natural\_number},\texttt{natural\_number}) : \texttt{natural\_number};\ \ldots$

For ex, we have that $\mathcal{DC}(ex) = \{\in,\ =,\ natural\_number,\ +\}$.

Now take the example of figure 10 (and call it ex'). This example introduces the definition of $\neg$ (Definition 1). We have that $\mathcal{DF}(ex') = \{\neg\}$.

The syntax of a definition in the internal representation of CGa (which is not necessarily the same as that given by the reader), is an identifier with a (possibly empty list of arguments) on the left-hand side followed by ":=" and an expression. The introduced symbol is the identifier of the left-hand side. For the example of figure 9 (call it ex''), the introduced symbol is $\subset$, hence $\mathcal{DF}(ex'') = \{\subset\}$ and the internal CGa representation is: $\subset(\mathtt{A},\mathtt{B}) := \mathtt{forall}(\mathtt{a},\mathtt{impl}(\mathtt{in}(\mathtt{a},\mathtt{A}),\mathtt{in}(\mathtt{a},\mathtt{B})))$.

Note that $\mathcal{ENV}(n)$ is the environment of all mathematical statements that occur before the statements of n (from the root node). Note furthermore that in the CGa syntax of MathLang, a definition or a declaration can only introduce a term, set, noun, adjective, or statement. Furthermore, recall that mathematical symbols or notions can only be introduced by definition or declaration and that mathematical facts can only be introduced by a statement. We define the set $\mathcal{IN}(n)$ of *introduced symbols and facts* of a DRa node n as follows:

$$\mathcal{IN}(n) := \mathcal{DF}(n) \cup \mathcal{DC}(n) \cup \{s | s \in \mathcal{ST}(n) \wedge s \notin \mathcal{ENV}(n)\} \cup \bigcup_{c\ childOf\ n} \mathcal{IN}(c)$$

At the heart of a context, step, definition, or declaration, is a set of statement, set, noun, adjective, and term. A DRa node n *uses* the set $\mathcal{USE}(n)$ where:

$$\mathcal{USE}(n) := \mathcal{T}(n) \cup \mathcal{S}(n) \cup \mathcal{N}(n) \cup \mathcal{A}(n) \cup \mathcal{ST}(n) \cup \bigcup_{c\ childOf\ n} \mathcal{USE}(c)$$

Lemma 1. *For every DRa node n we have:*
1. $\mathcal{DF}(n) \cup \mathcal{DC}(n) \subseteq \mathcal{T}(n) \cup \mathcal{S}(n) \cup \mathcal{N}(n) \cup \mathcal{A}(n) \cup \mathcal{ST}(n)$.
2. $\mathcal{IN}(n) \subseteq \mathcal{USE}(n)$.

Proof. We prove 2 by induction on the depth of parenthood of n. If n has no children then use lemma 1. Assume the property holds for all children c of n. By lemma 1 and the induction hypothesis, we have $\mathcal{IN}(n) \subseteq \mathcal{USE}(n)$. ■

We demonstrate these notions with an example. Consider a part of a mathematical text and its corresponding DRa tree with relations as in figure 10.

We assume the document starts with an environment which contains two statements, <True>$True$ and <False>$False$. Hence $\mathcal{ENV}(\mathsf{def1}) = \{True, False\}$. When traversing the tree we start with the given environment for the node def1:

$$\mathcal{ENV}(\mathsf{def1}) = \{True, False\}$$

The environment for case1 consists of the environment of def1 and all new statements of def1. In def1 there is only the new statement $\neg$ which is added to the environment: $\mathcal{ENV}(\mathsf{case1}) = \{\neg\} \cup \mathcal{ENV}(\mathsf{def1})$. After case1 all the statements of this node are added to the environment. These are $\neg True$ and $\neg True = False$:

$$\mathcal{ENV}(\mathsf{case2}) = \{\neg True, \neg True = False\} \cup \mathcal{ENV}(\mathsf{case1})$$

We can proceed with the building of the environment in the same way and get the last two environments of lem1 and pr1:

$$\mathcal{ENV}(\mathsf{lem1}) = \{\neg False, \neg False = True\} \cup \mathcal{ENV}(\mathsf{case2})$$
$$\mathcal{ENV}(\mathsf{pr1}) = \{\neg\neg True, \neg\neg True = True\} \cup \mathcal{ENV}(\mathsf{lem1})$$

With this information we derive the sets as shown in table 2 for the single nodes.

We can now formalise three different kinds of textual order $\prec$, $\preceq$ and $\leftrightarrow$:

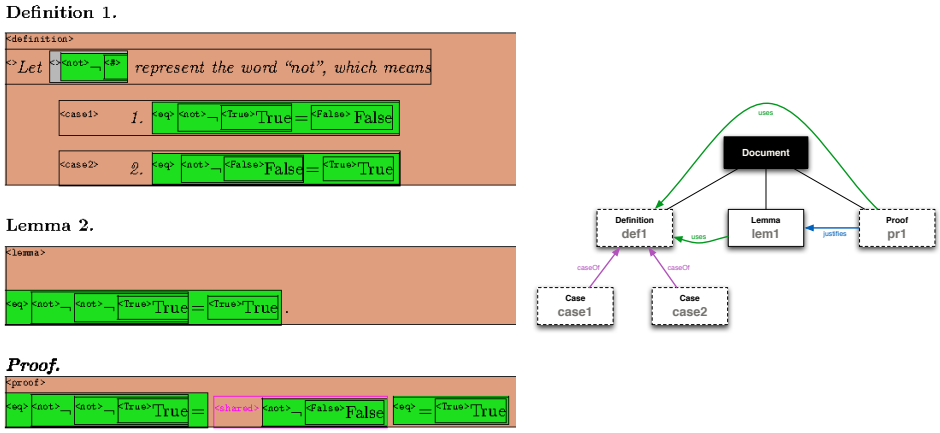


Figure 10. Example of an annotated text and its corresponding DRa tree

Node n	$\mathcal{IN}(n)$	$\mathcal{USE}(n)$
def1	$\{\neg\} \cup \mathcal{IN}(\text{Case 1}) \cup \mathcal{IN}(\text{Case 2})$	$\{\neg\} \cup \mathcal{USE}(\text{Case 1}) \cup \mathcal{USE}(\text{Case 2})$
case1	$\{\neg True, \neg True = False\}$	$\{True, False, \neg, \neg True, \neg True = False\}$
case2	$\{\neg False, \neg False = True\}$	$\{True, False, \neg, \neg False, \neg False = True\}$
lem1	$\{\neg\neg True, \neg\neg True = True\}$	$\{True, \neg, \neg True, \neg\neg True, \neg\neg True = True\}$
pr1	$\{\neg\neg True = \neg False\}$	$\{True, False, \neg, \neg True, \neg\neg True, \neg False, \neg\neg True = \neg False, \neg False = True\}$

Table 2. The sets $\mathcal{IN}$ and $\mathcal{USE}$ for the example

- **Strong textual order $\prec$:** If a node A uses a declared/defined symbol x or a statement x introduced by a node B, we say that A succeeds B and write $B \prec A$. More formally: $B \prec A := \exists x(x \in \mathcal{IN}(B) \wedge x \in \mathcal{USE}(A))$.
- **Weak textual order $\preceq$:** This order describes a subpart relation between two nodes (A is a subpart of B, written as $A \preceq B$). More formally:
  $$A \preceq B := \mathcal{IN}(A) \subseteq \mathcal{IN}(B) \wedge \mathcal{USE}(A) \subseteq \mathcal{USE}(B)$$
- **Common textual order $\leftrightarrow$:** This order describes the relation that two nodes use at least one common symbol or statement. More formally:
  $$A \leftrightarrow B := \exists x(x \in \mathcal{USE}(A) \wedge x \in \mathcal{USE}(B))$$

When $B \prec A$ (resp. $A \preceq B$) we also write $A \succ B$ (resp. $B \succeq A$). A DRa relation induces a textual order. Table 3 gives some relations and their textual order.

We can now verify the relations of the example of figure 10 and their textual orders (Table 4). It is obvious that all five conditions hold and hence the relations are valid. For example the relation (case2, uses, lem1) would not be valid, because $\neg\exists x(x \in \mathcal{USE}(\text{case1}) \wedge x \in \mathcal{IN}(\text{lem1}))$.

Note that these conditions are only of a syntactical form. There is no semantical checking if e.g. a "justifies" relation really connects a proved node and its proof.

Relation	**Meaning**	**Order**
A uses B	A uses a statement or a symbol of B	$B \prec A$
A inconsistentWith B	some statement in A contradicts a statement in B	$B \prec A$
A justifies B	A is the proof for B	$A \leftrightarrow B$
A relatesTo B	There is a connection between A and B but no dependence	$A \leftrightarrow B$
A caseOf B	A is a case of B	$A \preceq B$

Table 3. Example of DRa relations and their textual order

Relation	**Condition**	**Order**
(case1, caseOf, def1)	$\mathcal{IN}(\mathsf{case1}) \subseteq \mathcal{IN}(\mathsf{def1}) \wedge \mathcal{USE}(\mathsf{case1}) \subseteq \mathcal{USE}(\mathsf{def1})$	case1 $\preceq$ def1
(case2, caseOf, def1)	$\mathcal{IN}(\mathsf{case2}) \subseteq \mathcal{IN}(\mathsf{def1}) \wedge \mathcal{USE}(\mathsf{case2}) \subseteq \mathcal{USE}(\mathsf{def1})$	case2 $\preceq$ def1
(pr1, justifies, lem1)	$\exists x(x \in \mathcal{USE}(\mathsf{pr1}) \wedge x \in \mathcal{USE}(\mathsf{lem1}))$	pr1 $\leftrightarrow$ lem1
(lem1, uses, def1)	$\exists x(x \in \mathcal{USE}(\mathsf{lem1}) \wedge x \in \mathcal{IN}(\mathsf{def1}))$	def1 $\prec$ lem1
(pr1, uses, def1)	$\exists x(x \in \mathcal{USE}(\mathsf{pr1}) \wedge x \in \mathcal{IN}(\mathsf{def1}))$	def1 $\prec$ pr1

Table 4. Conditions for the relations of the example

The GoTO

The GoTO is the Graph of textual order. For each kind of relation in the dependency graph (DG) of a DRa tree we can provide a corresponding textual order $\prec, \preceq$ or $\leftrightarrow$. These different kinds of order can be interpreted as edges in a directed graph. So we can transform the dependency graph into a GoTO by transforming each edge of the DG. So far there are two reasons why the GoTO is produced:

1. Automatic Checking of the GoTO can reveal errors in the document (e.g. loops in the structure of the document).
2. The GoTO is used to automatically produce a proof skeleton for a prover.

To transform an edge of the DG we need to know which textual order it induces. Each relation has a specific order $\prec, \succ, \preceq, \succeq, \leftrightarrow$. Table 5 shows the graphical representation of such edges and an example relation we have seen in our examples. There is also a relation between a DRa node and its children: For each child c of

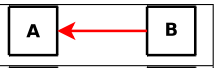
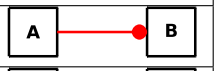
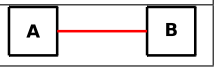
(A, uses, B)	$A \succ B$	
(A, caseOf, B)	$A \preceq B$	
$(A, \mathsf{justifies}, B)$	$A \leftrightarrow B$	

Table 5. Graphical representation of edges in the GoTO

a node n we have the edge $c \preceq n$ in the GoTO. This "childOf" relation is added automatically when producing the GoTO. But it can be added manually by the user. This can be useful e.g. in papers with a page restriction, where some parts

of the text are relocated in the appendix but would be originally within the main text. The algorithm for producing the GoTO from the DG works in two steps:

1. transform each relation of the DG into its corresponding edge in the GoTO
2. for each child c of a node n add the edge $c \preceq n$ to the GoTO

When performing this algorithm on the example of figure 7 we get the GoTO as demonstrated in figure 11. Each relation of the DG which induces a $\leftrightarrow$ textual order is replaced by the corresponding edge in the GoTO. We can see these edges between a proved node and its proof where the "justifies" relation induces a $\leftrightarrow$ order (e.g. between A and B, C and D, and F and G). The children of the node B are connected to B via $\preceq$ edges in the GoTO. For the "caseOf" relation, the user has manually specified the relation, the other edges were added automatically by the algorithm generating the GoTO. The relations which induce the order $\prec$ are transformed into the corresponding directed edges in the GoTO. We see that the direction of the nodes has changed with respect to the DG. This is because we only have "uses" relations, and for a relation (A, uses, B) we have the textual order $B \prec A$ which means, that the direction of the edge changes.

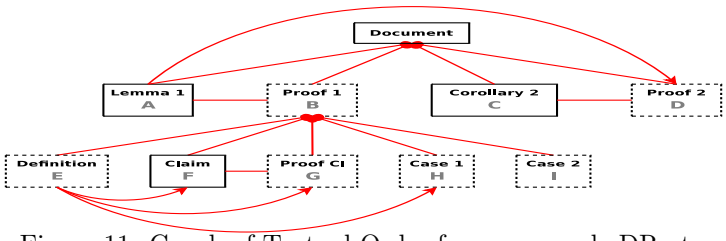


Figure 11. Graph of Textual Order for an example DRa tree

Automatic checking of DG and GoTO

We implemented two kinds of failures: warnings and errors. At the current development of DRa we check for four different kinds of failures:

1. Loops in the GoTO (error)
2. Proof of an unproved node (error)
3. More than one proof for a proved node (warning)
4. Missing proof for a proved node (warning)

The checks for 2) – 4) are performed in the DG. For 2) we check for every node of type "unproved" if there is an incoming edge of type "justifies". If so, an error is returned (e.g. when someone tries to prove an axiom or a definition). For 3) and 4) we check for each node of type "proved" if there is an incoming edge of type "justifies". If not, we return a warning (this can be a deliberate omission of the proof or just a mistake). If there is more than one proof for one node we return also a warning (most formal systems cannot handle multiple proofs).

For 1) we search for cycles in the GoTO. Therefore we have to define how we treat the three different kinds of edges. Edges of type $\prec$ and $\preceq$ are treated as directed edges. Edges of type $\leftrightarrow$ are in principal undirected edges, which means

for an edge $A \leftrightarrow B$, one can get from A to B and from B to A in the GoTO. It is vital, that within one cycle such an edge is only used in one direction. Otherwise we would have a trivial cycle between two nodes connected by a $\leftrightarrow$ edge.

As we will see in the next section, a single node in the DRa tree can first be translated when all its children nodes are ready to be translated. To reflect this circumstance we have to add certain nodes in the GoTO for the cycle check. Let us demonstrate this with an example. Consider a DG and GoTO as in figure 12.

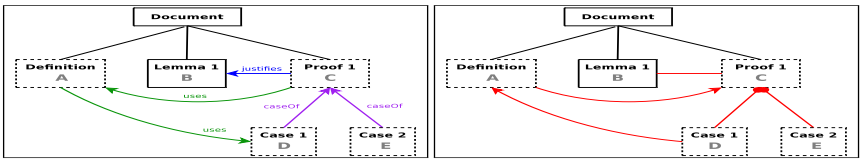


Figure 12. Example of a not recognised loop in a DRa (left DG, right GoTO)

Apparently there is a cycle in this tree, because to be able to translate C we need to translate its children D and E. Since C uses A, A must be translated before translating C. But the child D of C is used by A leading to a deadlock. Neither A nor C can be processed. To recognise such cycles we add certain edges to the GoTO when checking for cycles. Therefore we have to look at the children of a node n: hidden cycles can only evolve, when there are edges e_i from a child node c_i to a target node t_i which is not a sibling of c_i. Hence we add an edge $c_i \succ n$ for each such node e_i to the GoTO. This can be done via algorithm 1. We could also

```
foreach node n of the tree do
    foreach child c of n do
        foreach outgoing edge e of c do
            if target node t of e is no sibling of c then
                add a Strong textual precedence edge from n to t;
            end
        end
    end
end
```

Algorithm 1: Adding additional edges to the GoTO

add new edges for all incoming edges of the children c_i but this is not necessary since the textual order of the "childOf" relation is a directed edge from each child c_i to its parent node n and the transitivity of the edges helps find a cycle anyway.

In the example from figure 12, algorithm 1 would add one edge to the GoTO: The child node D of C has an outgoing node to the non-sibling node A. So a new directed edge from C to A is added which yields the result of figure 13 where a cycle between the nodes A, C and A with the edges A-C and C-A appears.

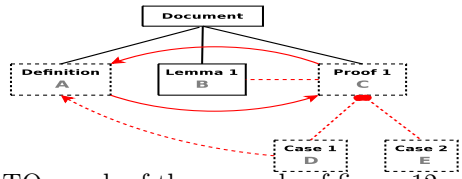


Figure 13. GoTO graph of the example of figure 12 with added edges

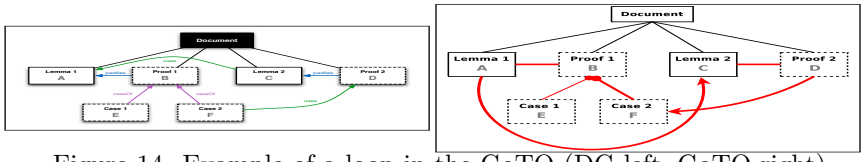


Figure 14. Example of a loop in the GoTO (DG left, GoTO right)

Figure 14 demonstrates another situation of a cycle in a DRa annotated text. The problem is mainly, that lemma 1 uses lemma 2 but the proof of lemma 2 uses a part of the proof of lemma 1. This situation would end up in a deadlock when processing the GoTO e.g. when producing the proof skeleton. We see a cycle between the nodes A, C, D, F, B and A with the edges A-C, C-D, D-F, F-B, and B-A. Here we also see why we do not need to add incoming edges to the parent nodes. For node F we have an incoming edge but due to the direction of the "childOf" edge from F to B, we can use the transivity. In both examples, an error would be returned with the corresponding nodes and edges.

The design and implementation of DRa were the subject of Retel's thesis [Retel, 2009]. Further additions have since been carried out by Zengler as reported here.

5 CONNECTING MATHLANG TO FORMAL FOUNDATIONS

Current approaches to formalising CML texts generally involve rewriting the text from scratch; there is no clear methodology in which the text can gradually change in small steps into its formal version. One of MathLang's goals is to support formalising a text in small steps that do not require radically reorganising the text. Also, a text with fully formal content should continue to be able to be presented in the same way as a less formal version originally developed by a mathematician. We envision formalisation as working by adding additional layers of information to a MathLang document to support embedding formal proofs. Ideally, there should be flexible control over how much of the additional information is presented to the reader; the additional information could form part of the visual presentation, or could exist "behind the scenes" to provide assurance of correctness.

As part of the goal of supporting formalisation in MathLang, we desire to keep MathLang independent of any particular formal foundation. However, as proofs embedded in a MathLang document become more formal, it will be necessary to tie them more closely to a particular proof system. It might be possible that fully formal documents could be kept independent of any particular foundation by allowing the most formal parts of a document to be expressed redundantly in multiple proof systems. (This is similar in spirit to the way the natural language portion of a document might be expressed simultaneously in multiple natural languages.) In this section we report on a methodology and software for connecting a MathLang document with formal versions of its content. We mainly concentrate on a formal foundation in Coq but for Mizar see [Kamareddine *et al.*, 2007b; Retel, 2009] and for Isabelle see [Lamar, 2011]. Our formalisation into Mizar, involved constructing a skeleton of a Mizar document (e.g. figure 15) from a Math-

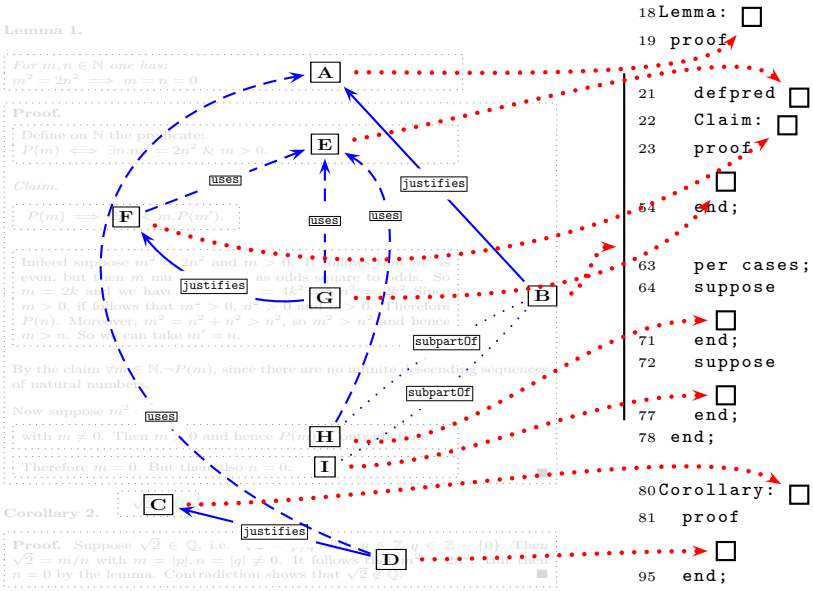


Figure 15. Generating a Mizar Text-Proper skeleton from DRa and CGa

Lang document, and then completing the Mizar skeleton separately. A Mizar document consists of an Environment-Declaration and a Text-Proper. In Mizar, the Environment-Declaration is used to generate the Environment which has the needed knowledge from MML (Mizar's Mathematical Library). The Text-Proper is checked for correctness using the knowledge in the Environment.

In this paper, we present the automation of the skeleton generation for arbitrary theorem provers and we give a generic algorithm for transforming the DRa tree into a proof skeleton. Since at this stage of formalisation we do not want to tie to any particular foundation, the algorithm is highly configurable which means it takes the desired theorem prover as an argument and generates the proof skeleton within this theorem prover. The aim of this skeleton generation is once again to stay as close as possible to the mathematician's original CML text. But due to certain restrictions for different theorem provers the original order cannot always be respected. We give some classical examples when this can happen:

- Nested lemmas/theorems: Sometimes mathematicians define new lemmas or theorems inside proofs. Not every theorem prover can handle such an approach (e.g. Coq). In the case of such theorem provers, it is necessary to "de-nest" the theorems/lemmas.
- Forward references: Sometimes a paper first gives an example for a theorem before it states the theorem. Some theorem provers (e.g. Mizar) do not support such forward references. The text has to be rewritten so that it only has backward references (i.e. to already stated mathematical constructs).
- Outsourced proofs: The practise in mathematical writing is to outsource in the appendix complex proofs that are not mandatory for the main results.

When formalising such texts, these proofs need to be put in the right place. The algorithm for re-arranging the parts of the text and generating the proof skeleton performs reordering only when necessary for the theorem prover at hand.

5a The generic automated Skeleton Generation Algorithm (gSGA)

The proof skeleton generation algorithm takes as arguments (cf. table 6):

1. the input MathLang XML file with DRa annotations; and
2. a configuration file (in XML format) for the theorem prover.

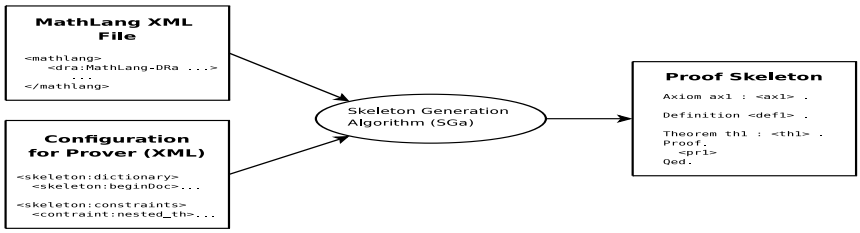


Table 6. The skeleton generation algorithm

This algorithm works on the DRa tree as seen in the last section. A DRa node can have one of three states: processed (black), in-process (grey) and unprocessed (white). A processed node has already been translated into a part of the proof skeleton, a node in-process is one that is being checked, while an unprocessed node is still awaiting translation. This information allows to identify which nodes have already been translated and which are still to be translated. The method for generating the output of a single node is shown in algorithm 2. The algorithm starts

```
while foundwhite do
    foreach child c of the node do
        if c is unprocessed && isReady(c) then
            processNode(c);
            generateOutput(c);
            foundwhite := true;
            break;
        end
    end
end
```

Algorithm 2: generateOuput(Node node)

at the Document root node, recursively searches for nodes in need of processing, and processes them so that the node at hand is translated and added to the proof skeleton. The decision whether a node is ready to be processed or not is only dependent on the GoTO of the DRa tree. A node is ready to be processed if:

1. It has no incoming $\prec$ edges (in the GoTO) of unprocessed (white) nodes.
2. All its children are ready to be processed.
3. If it is a proved node: its proof is ready to be processed.

Algorithm 3 tests these three properties of a node and returns the result. It is important when checking if each child of the n children is ready, to perform the test n times because a rearrangement can also be required for the children. If there

```
foreach incoming edge e of the node do
    if type of e is ≺ && source of e is unprocessed (white) then
        return false
    end
end
mark node n as grey;
n = number of children of the node;
for 1..n do
    foreach child c of the node do
        if c is not processed && isReady(c) then
            mark c as grey;
            break;
        end
    end
end
if still a white node is among the children of the node then
    reset all grey nodes back to white;
    return false
end
if node is a proved node then
    proof = proof of the node;
    if not isReady(proof) then
        reset all grey nodes back to white;
        return false
    end
end
reset all grey nodes back to white;
return true
```

Algorithm 3: isReady(Node node)

are still white children after n steps, then the children cannot be yet processed and so the node cannot be processed.

To illustrate algorithm 3 we look at a (typical and not well structured) mathematical text whose DG and GoTO edges are in figure 16.

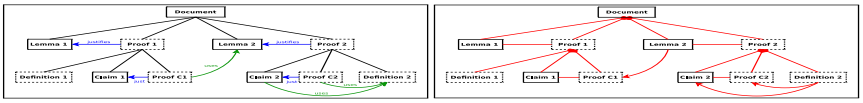
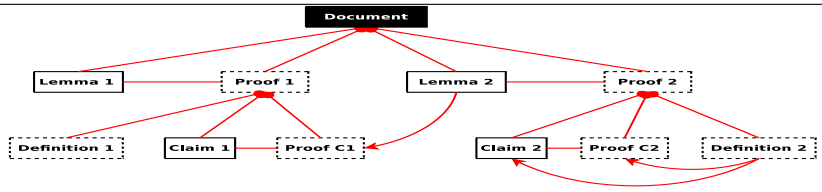
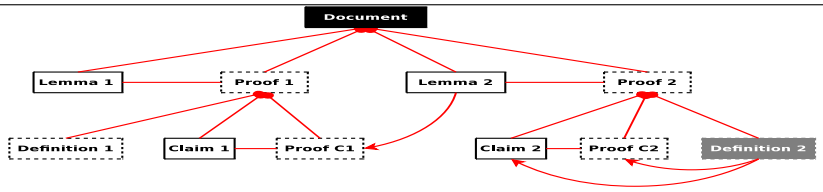


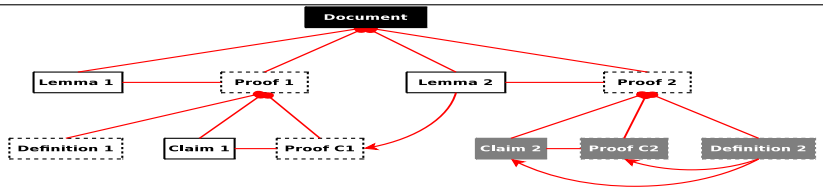
Figure 16. To illustrate Skeleton generation (DG at top, GoTO at bottom)



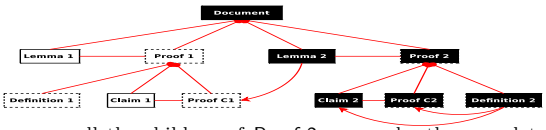
The root node of the document can be marked as processed and the algorithm starts at this node. The first child is Lemma 1. Criterion 1) is fulfilled, since the node has no incoming $\prec$ edges in the GoTO. Criterion 2) is fulfilled because the node has no children. For criterion 3) Proof 1 has to be ready to be processed before we can mark Lemma 1 as ready to be processed. Proof 1 has no incoming $\prec$ edges. So criterion 1) is fulfilled. For criterion 2) the children of the proof have to be ready to be processed. Definition 1 is ready, but the proof of Claim 1, Proof C1 has an incoming node of an unprocessed node (Lemma 2). So Claim 1 is not ready and hence, neither are Proof 1 and Lemma 1.



The next Node to check is Lemma2. Criteria 1) and 2) are fulfilled, for criterion 3) the proof Proof 2 has to be ready to be processed. Criteria 1) and 3) of the proof are fulfilled, so its children are ready to be processed. The first child that can be processed is Definition 2. So it is marked as in-process (grey).



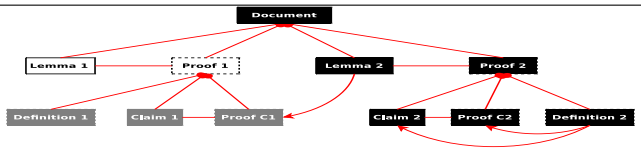
In a second run of the for loop for checking the children of Proof 2, Claim 2 and its proof are now ready, because Definition 2 is not white anymore but grey. This situation is the reason, why we must perform the check whether the n children of a node are ready n times exactly.



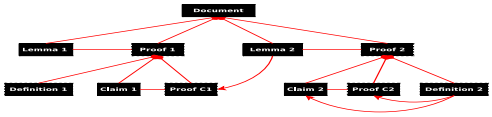
Output:

```
Lemma 2
Proof 2
   Definition 2
   Claim 2
   Proof C2
```

Since now all the children of Proof 2 are ready, the complete proof is ready and so is Lemma 2. The grey flags are unassigned and the output for Lemma 2 is generated. In this step all nodes Lemma 2, Proof 2, Claim 2, Proof C2 and Definition 2 are permanently marked as processed (black).



Since now a node has been processed, the algorithm starts again with the first white node. So Lemma 1 is checked again. Now the children of its proof can be processed because Lemma 2 is now processed and does not prevent the processing of Proof C1.



Output:

```
Lemma 1
Proof 1
   Definition 1
   Claim 1
   Proof C1
```

At the end Lemma 1 and its proof can be processed. The final order of the nodes is:

```
Lemma 2
Proof 2
   Definition 2
   Claim 2
   Proof C2
Lemma 1
Proof 1
   Definition 1
   Claim 1
   Proof C1
```

We see that with this order no node references other nodes which are not already translated. Lemma 2 is translated first. Its proof follows immediately. Definition 2 is reordered, because Claim 1 and its proof refer to it. So it has to be written in front of them. Lemma 1 can then be translated because now Lemma 2 which it refers to, is already translated.

5b The configuration of gSGA

The transformation of the DRa annotated text into a proof skeleton has two steps:

- Reorder the text to satisfy the constraints of the particular theorem prover.
- Translate each DRa annotation into the language of the theorem prover.

The configuration file for a particular theorem prover for the gSGA reflects these two steps: there is a dictionary part and a constraints part. The dictionary contains a rule for each mathematical or structural role of DRa. A single DRa node has two important properties: a name and a content. This information is used in the translation. Within the configuration file we can refer to the name of a node with `%name` and to the body with `%body`. A new line (for better readability) can be inserted with `%nl`. Consider the example of the DRa node from figure 8. The role of this node is declaration and its name is decA. The body of this node is the sentence *Let A be a set* or its CGa annotation. A translation into Mizar could be:

```
reserve <body of decA> ;
```

The rule for this translation would be:

```
reserve %body ;
```

Such a kind of declaration in Coq would be:

```
Variable <body of decA> .
```

And the for this translation would be:

```
Variable %body .
```

Here, we let a single rule be embedded in an XML tag whose attribute "name" is the corresponding keyword:

```
<skeleton:keyword name="declaration">
  reserve %body ;
</skeleton:keyword>
```

The constraints section of the configuration file for a theorem prover configures two main properties: the allowance of forward properties and of nested mathematical constructs. Forward references can be allowed via the tag:

```
<skeleton:forwardrefs>true</skeleton:forwardrefs>
```

Changing the content of the tag to "false" forbids forward references. If there is no such tag, the default value is "false".

For a configuration of nested constructs there are two possibilities:

- Either allow in general the nesting of constructs defining those exceptions for which nesting is not allowed;
- Or forbid in general the nesting of constructs defining those exceptions for which nesting is allowed.

The next configuration allows nesting in general but not for definitions and axioms:

```
<skeleton:nesting>true</skeleton:nesting>
<skeleton:nest role="definition">false</skeleton:nest>
<skeleton:nest role="axiom">false</skeleton:nest>
```

5c The flattening of the DRa graph

The next question we have to deal with, is how to perform changes to the tree when certain nestings are not allowed. We call this a flattening of the graph, because certain nodes are removed from their original position and inserted as direct children of the DRa top-level node. Algorithm 4 achieves this effect.

```
foreach (child c of the node) do
    flattenNode(c);
    if c cannot be nested then
        nodelist := transitive closure of incoming nodes of c;
        foreach node n of nodelist do
            remove n of list of children of node;
            add n in front of node as a sibling;
        end
    end
end
```

Algorithm 4: flattenNode(Node node)

We refer to every child of the DRa top-level node as a node at level 1. Every child of such a node is at level 2 and so on. If a mathematical role must not be nested, it can only appear at level 1. So we check for each node at a level greater than level 1, if its corresponding mathematical role can be nested. If not, then the node and all its required siblings are removed from this level and put in front of their parent node. Since there is no "childOf" relation between this no-longer-child and its parent node, the relation between child and parent changes form $\preceq$ to $\prec$.

The required sibling nodes are determined in the GoTO. When a node is moved in front of its parent node, there is a $\prec$ edge between this node and its former parent. Each sibling of the removed node from whom there is a incoming node must be moved with the node. This includes its children or - for a proved node - its proof. Since for these children we have to move the related nodes too, we can

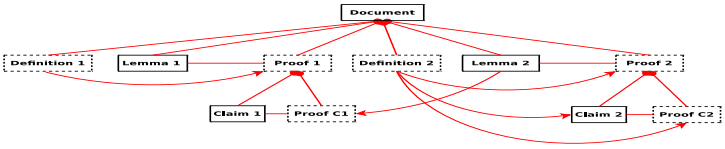


Figure 17. A flattened graph of the GoTO of figure 16 without nested definitions

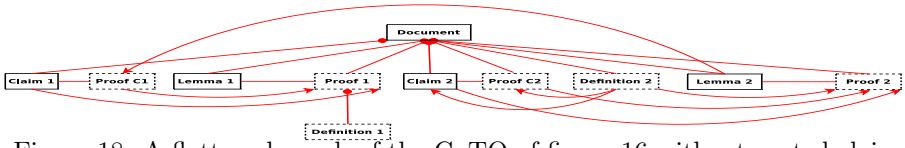


Figure 18. A flattened graph of the GoTO of figure 16 without nested claims

build the transitive closure over the incoming nodes of the node which has to be moved. All nodes in this closure have to be relocated in front of the parent node.

We demonstrate this algorithm again on the example from figure 16. For a first demonstration we assume that the nesting of definitions is not allowed. So Definition 1 and Definition 2 have to be removed from level 2 and be relocated in front of their parent nodes. The transitive closure over incoming edges in the GoTO yields no new nodes for removing (because the definitions have no incoming edges in the GoTO). The resulting new flattened graph can be seen in figure 17. We see that the two definition are now at level 1 and their edges to their former parent nodes have changed from $\preceq$ to $\prec$. The output for this graph according to the algorithm from the last section is given on the left-hand side of table 7.

Definition 1	Definition 2
Definition 2	Claim 2
Lemma 2	Proof C2
Proof 2	Lemma 2
Claim 2	Proof 2
Proof C2	Claim 1
Lemma 1	Proof C1
Proof 1	Lemma 1
Claim 1	Proof 1
Proof C1	Definition 1

Table 7. Outputs of the graphs of figures 17 (left-hand side) and 18 (right-hand side)

On the other hand, if we allow definitions to be nested but forbid nested claims, we get the graph of figure 18. The first claim which is found in the graph is Claim 1. The transitive closure yields that Proof C1 needs also to be removed since there is a $\leftrightarrow$ edge to the claim. The second claim which is found is Claim 2. The transitive closure yields again that its proof as well as Definition 2 have to be removed.

The output for this graph is given on the right-hand side of table 7.

6 CONNECTING MPL TO FORMAL FOUNDATION

6a Newman's Lemma

As a case study we specify for Newman's Lemma a feasible interactive mathematical proof development. It should be accepted by an interactive proof assistant, if these are to be accepted by a working mathematician. Table 8 gives an actual proof development in Coq for the main lemma is given. We start with the informal statement and proof.

Let A be a set and let R be a binary relation on A. R^+ is the transitive closure of R and R^* is the transitive reflexive closure of R.

Confluence of R, notation $\mathrm{CR}(R)$ (Church-Rosser property), is defined as follows (the notion $\mathrm{cr}(R,a)$ denotes confluence from $a \in A$). $\mathrm{WCR}(R)$ stands for *weak confluence*.

1. $\mathrm{cr}_R(a) \iff \forall b_1, b_2 \in A.[aR^*b_1 \mathbin{/\backslash} aR^*b_2 \Rightarrow \exists c.b_1R^*c \mathbin{/\backslash} b_2R^*c]$.
2. $\mathrm{CR}(R) \iff \forall a \in A.\mathrm{cr}_R(a)$.
3. $\mathrm{WCR}(R) \iff \forall a, b_1, b_2 \in A.[aRb_1 \mathbin{/\backslash} aRb_2 \Rightarrow \exists c.b_1R^*c \mathbin{/\backslash} b_2R^*c]$.

Newman's lemma states that for well-founded relations weak confluence implies strong confluence. The notion of well-foundedness is formulated as the possibility to prove statements by transfinite induction. Let $P \in \mathcal{P}(A)$.

4. $\mathrm{IND}_R(P) \iff \forall a \in A.(\forall y \in A.aRy \Rightarrow P(y)) \Rightarrow P(a)$.
5. $\mathrm{WF}(R) \iff \forall P \in \mathcal{P}(A).[\mathrm{IND}_R(P) \Rightarrow \forall a \in A.P(a)]$.

Lemma 3 (Main Lemma.). $\mathrm{WCR}(R) \Rightarrow \mathrm{IND}_R(cr_R)$.

Proof. Assume WCR(R). Remember $IND_R(cr_R) \Leftrightarrow$

$$(\forall a : A.(\forall y : A.a\ R\ y \rightarrow cr(R,y)) \rightarrow (cr_R(a)).$$

Let $a : A$ and assume

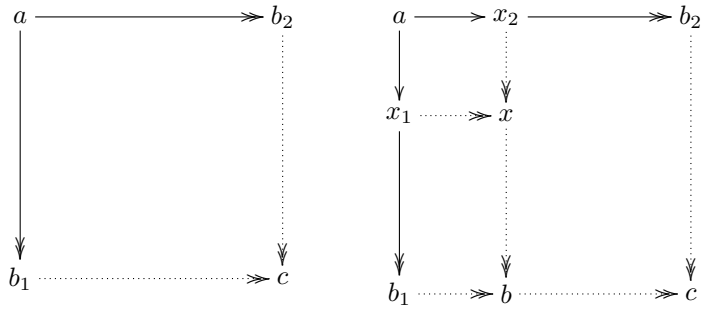
$$\forall y : A.a\ R\ y \rightarrow cr_R(y), \qquad (IH)$$

in order to show $\mathrm{cr}_R(a)$, i.e.

$$\forall b_1, b_2 : A.a\ R^*\ b_1 \mathbin{/\backslash} a\ R^*\ b_2 \rightarrow (\exists c\ A.b_1\ R^*\ c \mathbin{/\backslash} b_2\ R^*\ c).$$

So let $b_1, b_2 : A$ with $a\ R^*\ b_i$, in order to show $\exists c.b_i\ R^*\ c$.

If $a = b_1$ or $a = b_2$, then the result is trivial (take $c = b_2$ or $c = b_1$ respectively). So by lemma p7 (below) we may assume $a\ R^+\ b_i$,
which by lemma p6 (below) means $a\ R\ x_i\ R^*\ b_i$, for some x_1, x_2.



By WCR(R) there is an x such that $x_i\ R^*\ x$.
By (IH) one has $cr_R(x_1)$. So $x\ R^*\ b$ /\ $b_1\ R^*\ b$, for some b.
Again $cr_R(x_2)$. As $x_2\ R^*\ x\ R^*\ b$ one has $b\ R^*\ c$ /\ $b_2\ R^*\ c$, for some c.
Then $b_1\ R^*\ b\ R^*\ c$ and we are done. ■

PROPOSITION 3 (Newman's Lemma). *WCR(R) /\ WF(R) ⇒ CR(R).*

Proof. By WCR(R) and the main lemma we have $\mathrm{IND}_R(\mathrm{cr}_R)$. Hence by WF($R$) it follows that for $P(a) = \mathrm{cr}_R(a)$, one has $\forall a \in A.\mathrm{cr}_R(a)$. This is $CR(R)$. ■

Now we will start a proof development for Newman's lemma.

```
Variable A:Set.
Definition Bin:=[B:Set](B->B->Prop).

Inductive TC [R:(Bin A)]: (Bin A) :=
TCb:  (x,y:A)(R x y)->(TC R x y)|
TCf:  (x,y,z:A)((R x z)->(TC R z y)->(TC R x y)).

Inductive TRC [R:(Bin A)]: (Bin A) :=
TRCb:  (x:A)(TRC R x x)|
TRCf:  (x,y,z:A)((R x z) -> (TRC R z y)->(TRC R x y)).

Definition Trans [R:(Bin A)]: Prop:=
(x,y,z:A)((R x y)->(R y z)->(R x z)).

Definition IND [R: (Bin A);P:(A->Prop)]:  Prop :=
((a:A)((y:A)(a R y)-> (P y))->(P a)).

Definition cr [R:(Bin A);a:A]:=
(b1,b2:A)(TRC R a b1)/\(TRC R a b2)->(EX c:A|(TRC R b1 c)/\(TRC R b2 c)).

Definition CR [R:(Bin A)]:=(a:A)(cr R a).

Definition WCR [R:(Bin A)]:=
(a,b1,b2:A)(a R b1)->(a R b2)->(EX c:A|(TRC R b1 c)/\(TRC R b2 c)).

Definition WF [R:(Bin A)]:Prop:= (P:A->Prop)(IND R P)->(a:A)(P a).
```

```
Variable R:(Bin A).

Lemma p0:  (x,y:A)((R x y) -> (TC R x y)).
Lemma p1:  (x,y:A)((R x y) -> (TRC R x y)).
Lemma p2:  (x,y:A)((TC R x y) -> (TRC R x y)).
Lemma p3:  (Trans (TC R)).
Lemma p4:  (Trans (TRC R)).
Lemma p5:  (x,y,z:A)(R x y)->(TRC R y z)->(TRC R x z).
Lemma p6:  (x,y:A)((TC R x y)->(EX z:A | (R x z)/\(TRC R z y))).
Lemma p7:  (x,y:A)((TRC R x y)-> (eq A x y)\/(TC R x y)).
```

The proof-script for these lemmas are not shown. The main lemma is as follows.

```
Lemma main :  (WCR R)->(IND R (cr R)).
```

The proof-script in Coq is given in table 8. Now we will give an interactive mathematical proof script, for which we claim that it should essentially be acceptable by a mathematician-friendly proof-assistant. On the left we find the mathematical script, on the right the proof-state. These may contain some Coq-like statements, like "Intros", but these disappear and are replaced by mathematical statements. For an interactive version see `www.cs.kun.nl/~henk/mathmode.{ps, dvi}`. The dvi version has to be viewed in advi obtainable from `pauillac.inria.fr/~miquel`.

First we introduce some user-friendly notation.

Notation 4. For a,b:A we write

(i) a R b := (R a b).

(ii) a R+ b := (TC R a b).

(iii) a R* b := (TRC R a b).

```
Proof.  Assume WCR(R). Remember IND. Let a:A. Assume
          (y:A)((aRy)->(cr R y)).           (IH)
Remember cr.  Let a,b1,b2:A. Assume a R* bi, i=1,2.
We have
[a=b1 \/ a R+ b1],
[a=b2 \/ a R+ b2],
by lemma p6.
Case a=b1, take c=b2.  Trivial.  Hence wlog (a R+ b1).
Case a=b2, take c=b1.  Trivial.  Hence wlog (a R+ b2).
Therefore (EX xi:A|a R xi R* bi), i=1,2, by lemma p7.
Pick x1.  Pick x2.
We have (EX x.xi R* x), i=1,2, by (WCR R). Pick x.
```

```
We have (cr R x1), by IH. Hence
(EX b.b1 R* b /\ x R* b).
Pick b.  Moreover (cr R x2), by IH. Hence
(EX c.b R* c /\ b2 R* c),
by x2 R* b.  Pick c.  Since b1 R* c, by (Trans R*), we have
(bi R* c), i=1,2.  Thus c works.  QED


Newman's Lemma.  WCR(R)/\WF(R)->CR(R).

Proof.  Assume WCR(R) and WF(R). Then (IND R( cr R)),
by WCR(R) and main.  Remember CR and WF. We have
(P:(A->Prop))((a:A)((y:A)(a R y)-> (P y))->(P a)).           (+)
Apply (+) to (cr R). Then CR(R). QED
```

6b Towards a Mathematical Proof Language MPL

We will now sketch rather loosely a language that may be called MPL: *Mathematical Proof Language*. The language will need many extensions, but this kernel may be already useful.

DEFINITION 5. The phrases used in MPL for the proposed proof-assistant with interactive mathematical mode belong to the following set.

```
Assume B                        Then B [, by C]
Towards A                       Suffices
Remember t                      Wlog B, [since B \/ C]
Let x:D
Pick [in L] x                   and
Case B                          QED
Take x=t [in B]
Apply B to t
As to
```

Here `A,B, C` are propositions in context `Gamma`, `D` is a type, `x` is a variable and `t` is a term of the right type. "`Wlog`" stands for "Without loss of generality".

DEFINITION 6 (Synonyms).

```
Suffices = In order to show = We must show = Towards;
Let = Given;
Then = We have = It follows that = Hence = Moreover = Again;
and = with;
by = since.
```

Before giving a grammar for tactic statements we will give their semantics. They have a precise effect on the proof-state. In the following definition we show what the effect is of a statement on the proof-state. In some cases the tactic has a side-effect on the proof-script, as we saw in the case of Newman's lemma.

DEFINITION 7.

(i) A *proof-state* (within a context `Gamma`) is a set of statements `Delta` and a statement `A`, such that all members of `Delta` are well-formed in `Gamma` and `A` is well-formed in `Gamma, Delta`. If the proof-state is `(Delta;A)`, then the goal is to show `Delta` $\vdash$ `A`.

(ii) The *initial* proof-state of a statement `A` to be proved is of course `(∅;A)`.

(iii) A *tactic* is map from proof-states to a list of proof-states, usually having a formula or an element as extra argument.

DEFINITION 8.

```
Assume C (Delta,C->B)          =  (Delta,C;B), and ‘‘Towards B’’
                                  may be left in the script.
Let a:D (Delta,(x:D.P))        =  (Delta,a:A;P[x=a]).
Remember name (Delta;A)        =  (Delta;A’), where A’ results
                                  from A by unfolding the defined
                                  concept ‘name’.  This can be
                                  applied to an occurrence of
                                  ‘name’, by clicking on it.  Other
                                  occurrences remain closed but
                                  become transparent (as if opened).
Pick [in L] x (Delta,L;A)      =  (Delta,x:D,B(x);A), where L is
                                  a formula reference of (EX x:D.B).
Take x=name (Delta;EX x:D.A)   =  (Delta;A[x=name]),
                                  if Delta|- name:D.
Apply B to name (Delta;A)      =  (Delta,P[y=name];A), where B of
                                  the form ((y:D).P) is in Delta.
Case B (Delta;A)               =  (Delta,B;A),(Delta,C;A),
                                  if B \/ C in Delta; the second
                                  proof-state represents
                                  the next subgoal.
As to Bi (Delta;B0 /\ B1)      =  (Delta;Bi),(Delta;B(1-i)), the
                                  second proof-state represents
                                  the next subgoal;
As to B (Delta; B)             =  (Delta; B);
QED (Delta,B)                  =  <proof terminated> if B in Delta.
```

In all cases nothing happens if the side conditions are not satisfied. One should be able to refer to a statement `C` in two ways: either by naming `C` directly of by referring to a label for `C`, like “`IH`” in the proof of the main lemma above. We say that L is a formula reference of formula B if L is B or if L is a label for B. Labels are sometimes handy, but they should also be suppressed in order to keep the proof-state clean. If the argument of a tactic occurs at several places the system should complain. Then reference should be made to a unique label. It is assumed that proof-states `(Delta,A)` are in normal form, that is, if `B /\ C` is in `Delta`, then it is replaced by the pair `B,C`. If the final `QED` is accepted, then all

the statements in the proof that did not have an effect on the proof-state will be suppressed in the final lay-out of the proof (or may be kept in color orange as an option in order to learn where one did superfluous steps).

The following tactics require some automated deduction. If the proof-assistant cannot prove the claimed result, an extra proof-state will be generated so that this result will be treated as the next subgoal.

DEFINITION 9.

```
[Since B\/C] wlog C (Delta;A)   =   (Delta,C;A), if B\/C in Delta
                                    and the assistant can establish
                                    Delta|-B->A.
Then B[, by C] (Delta;A)        =   (Delta,B;A), if C is a known
                                    lemma and the assistant
                                    can establish Delta,C|-B.
Suffices B (Delta;A)            =   (Delta;B) and the assistant
                                    can establish Delta|-B->A.
May assume B (Delta,A)          =   (Delta,B;A) if the assistant
                                    can establish Delta|-~B->A and
                                    Delta|- B \/~B.
```

The tactic language MPL is defined by the following grammar.

```
formref   :=  label | form
  form+   :=  formref | form+ and formref
 tactic   :=  Assume form+ | Towards form | Remember name |
              Let var:set | Pick [in formref] var | Case form |
              Take var = term [in formref ] |
              Apply formref to term | Then form[, by form+] |
              Suffices formref | Wlog form[, since form\/form]
tactic+   :=  tactic.  | tactic, tactic+ | tactic.  tactic+
```

Here label is the proof-variable, used as a name for a statement (like IH in the proof of the main lemma), form is a Gamma, Delta inhabitant of Prop, name is any defined notion during the proof development, and var is an variable.

An extension of MPL capable of dealing with computations will be useful.

```
We have A(t).  Then A(s), since t=s.
```

Another one:

```
Then t=s, by computation.
```

It would be nice to have this in an ambiguous way: computation is meant to be pure conversion or an application of reflection. This corresponds to the actual mathematical usage:

5!=120, by computation.
In a commutative ring, $(x + y)^2 = x^2 + 2xy + y^2$, by computation.

In type theory the first equality would be an application of the conversion rule, but for the second one reflection, see e.g. [Constable, 1995], is needed.

6c Procedural statements in the implementation of MPL

As we have seen in section 6a it is handy to have statements that modify the proof state, but are not recorded as such. For example if the proof state is

$$(\texttt{Delta}; (\texttt{x} : \texttt{D})(\texttt{A(x)} -> \texttt{B(x)}),$$

then `Intros` is a fast way to generate

```
Let x:D. Assume A(x), in order to prove B(x).
```

in the proof. Another example is `Clear L` which removes formula `L` in the assumptions of the current subgoal. Also renaming variables is useful, as some statements may come from libraries and have a "wrong" choice of bound variables.

7 A FULL FORMALISATION IN COQ VIA MATHLANG: CHAPTER 1 OF LANDAU'S "GRUNDLAGEN DER ANALYSIS"

Landau's "Grundlagen der Analysis" [Landau, 1951] remains the only book which has been fully formalised in a theorem prover [van Benthem Jutting, 1977b]. This section summarises the encoding of the first chapter (natural numbers) of Landau's book into all aspects of MathLang up to a full formalisation in Coq. We give a complete CGa, TSa and DRa annotation for the chapter, we generate a proof skeleton automatically with the gSGA forboth Mizar and Coq and then we give a complete formalised version of the chapter in Coq. To accomplish this, we have used the MathLang TEX$_{\text{MACS}}$ plugin to annotate the existing plaintext of the book.

To clarify the path we took, we look once again at the overall diagram of the different paths in MathLang (figure 1). We first used path ⓐ and annotated the complete text with CGa, TSa and DRa annotations with the help of the MathLang TEX$_{\text{MACS}}$ plugin. The second step was to automatically generate a proof skeleton of the annotated text. With the help of the proof skeleton and the CGa annotations we fully formalised the proofs in Coq completing the paths ⓓ and ⓔ. The final result is a fully formalised version of the first chapter of Landau's book in Coq.

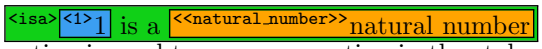
7a CGa and TSa annotations

The Preface

In the preface of a MathLang document we introduce symbols that are not defined in the text but are used throughout it. These are often quantifiers or Boolean connectives like $\wedge$ or $\vee$. These symbols are often pre-encoded in theorem provers (e.g. Coq has special symbols for the logical and, or, implication, etc.). The preface

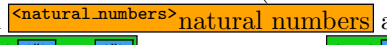
of the first chapter of Landau's book consists of 17 different symbols as given in table 7a. Two functions deserve further explanation:

1. The "is a" function is used to express that a particular term is an instance of a noun. E.g. the first axiom of the book is that *1 is a natural number*, so the encoding of this axiom is

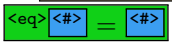


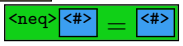
2. The "index" function is used to express a notion in the style of $a_b = c$ which can be defined as a function $index(a, b) = c$. So the index function has two terms as argument and yields a term as a result.

The first section

The first section of the first chapter introduces the natural numbers, equality on natural numbers and five axioms (an extension of the Peano axioms). We introduce a noun 

and the set 

of natural numbers. Equality 

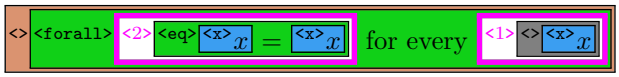
and inequality 

between natural numbers are declared rather than defined. Three properties of equality are encoded. We will show one encoding of these to recapitulate TSa annotations with sharing and to see how to use the symbols given in the preface. The original statement is

$$x = x \text{ for every } x$$

We see that this is a universal quantification of x and a well formed equivalent statement would be: $\forall x(x = x)$. Since the positions are swapped in Landau's text we use the position souring. The souring annotation of this statement is:

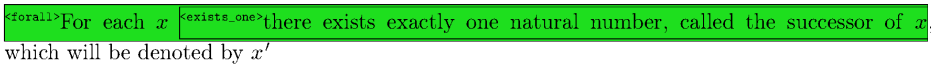
This yields the final statement



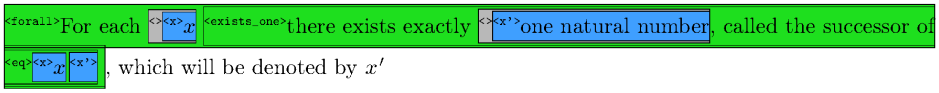
Next we show how to encode axiom 2 showing that "wordy" parts of a text can also be annotated, not only mathematical statements. The original statement is:

For each x there exists exactly one natural number, called the successor of x, which will be denoted by x'

The "for each" can be translated with a universal quantifier, the "exactly one" with the $\exists!$ quantifier. So we get the general structure:



The complete statement can be e.g. encoded corresponding to the following formal statement $\forall x(\exists! x'(succ(x) = x'))$:



Sections 2 - 4
Within the next sections, addition (section 2), ordering (section 3) and multiplication (section 4) are introduced. There are 36 theorems with proofs and 6 definitions: addition, greater than, less then, greater or equal than, less or equal than and multiplication. There are many simple structured theorems like that of figure 19. We want to examine our way of annotating these theorems. The main

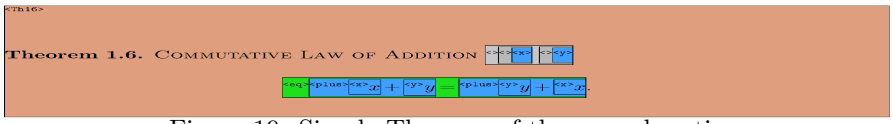


Figure 19. Simple Theorem of the second section

theorem $x + y = y + x$ is annotated in a straightforward manner. x and y are annotated as terms, plus as a function, taking two terms as arguments and yielding a term as a result. The equality between these terms is a statement. Since we did not declare x and y in the preface or in a global context we do this with a local scoping. This information is added in the first annotated line where we declare x and y as terms and put these two annotations into a context which means that this binding holds within the whole step.

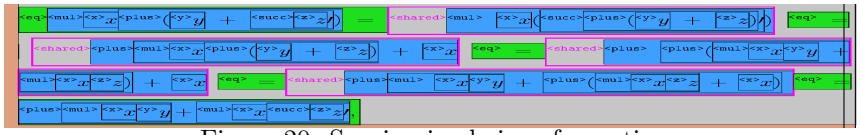


Figure 20. Souring in chains of equations

Landau often used chains of equations for proofs as in this proof for the equality of $x(y + z')$ and $xy + xz'$ in the proof of Theorem 30 of the first chapter:

$$x(y+z') = x((y+z)') = x(y+z)+x = (xy+xz) = x = xy = (xz+x) = xy+xz'$$

Here we benefit from our souring methods - especially the sharing of variables (See figure 20). There are also often hidden quantification like the one in the example $x = x$ *for every* x, where we need the souring for swapping positions. These TSa functionalities save a lot of time in annotating mathematical documents.

For some theorems we use the Boolean connectives although they are not mentioned explicitly in the text. E.g. Theorem 16 states:

$$\text{If } x \leq y,\ y < z \text{ or } x < y,\ y \leq z$$

$$\text{then } x < z$$

We annotate the premise of the theorem as a disjunction of two conjunctions as seen in figure 21. Another use of Boolean connectives is when we have formulations like *"exactly one of the following must be the case..."*. There we use the exclusive or $\oplus$ to annotate the fact that exactly one of the cases must hold. We defined the exclusive or in the preface and therefore have to take care that we find a corresponding construct in the used theorem prover (see table 7a).

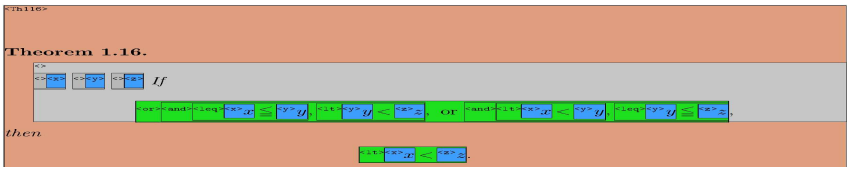


Figure 21. The annotated Theorem 16 of the Landau's first chapter

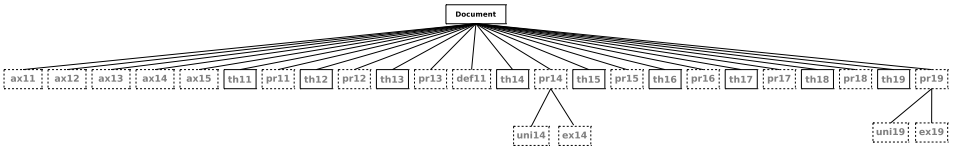


Figure 22. The DRa tree of sections 1 and 2 of chapter 1 of Landau's book

7b *DRa annotation*

The structure of Landau's "Grundlagen der Analysis" is very clear: in the first section he introduces five axioms. We annotate these axioms with the mathematical role "axiom", give them the names "ax11" - "ax15" and classify them as unproved nodes. In the following sections we have 6 definitions which we annotate with the mathematical role "definition", give them names "def11" - "def16" and classify as unproved nodes. We have 36 proved nodes with the role "theorem", named "th11" - "th136" and with proofs "pr11" - "pr136".

Some proofs are partitioned into an existential part and a uniqueness part. This partitioning can be useful e.g. for Mizar where we have keywords for these parts of a proof. In the Coq formalisation, we used this partitioning to generate two single proofs in the proof skeleton which makes it easier to formalise. Other proofs consist of different cases which we annotate as unproved nodes with the mathematical role "case". This can be translated in the Mizar "per cases" statement or in single proofs in Coq. The DRa tree for sections 1 and 2 can be seen in figure 22.

The relations are annotated in a straightforward manner. Each proof *justifies* its corresponding theorem. Some of the axioms depend on each other. Axiom 5 ("ax15") is the axiom of induction. So every proof which uses induction, *uses* also this axiom. Definition 1 ("def11") is the definition of addition. Hence every node which uses addition also *uses* this definition. Some theorems *use* other theorems via texts like: "By Theorem ...". In total we have 36 *justifies* relations, 154 *uses* relations, 6 *caseOf*, 3 *existencePartOf* and 3 *uniquenessPartOf* relations. Figures 23 and 24 give the DG and GOTO of sections 1 and 2 resp. of the whole book. The DGs and GOTOs are automatically produced from the DRa annotated text. There are no errors or warning in the document which means we have no loops in the GoTO, no proofs for unproved nodes, no double proofs for a node and no missing proofs for proved nodes.

7c *Generation of the proof skeleton*

Since there are no errors in the GoTO, the proof skeleton can be produced without warnings. We have 8 mathematical roles in the document: axioms, definitions, theorems, proofs, cases, case, existenceParts and uniquenessParts. We make a

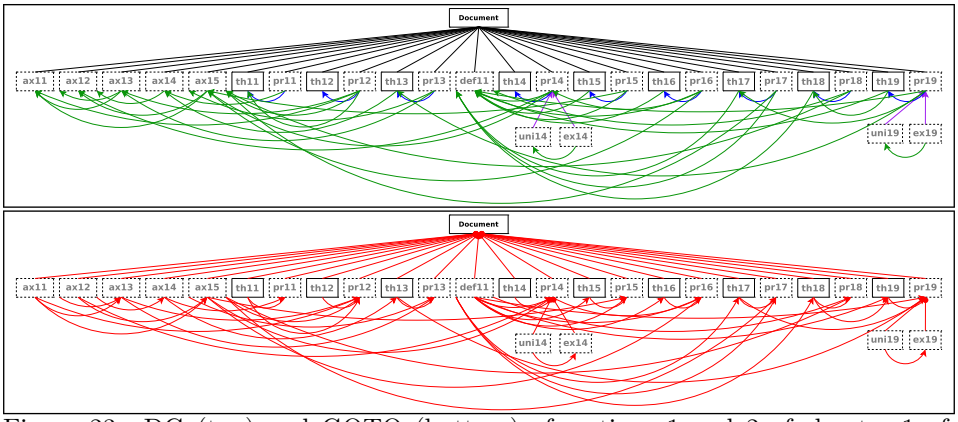


Figure 23. DG (top) and GOTO (bottom) of sections 1 and 2 of chapter 1 of Landau's book

distinction between cases and case because e.g. in Mizar we have a special keyword introducing cases (`per cases;`) and then keywords for each case (`suppose ...`). So we annotated the cases as child nodes of the case node. Table 10 gives an overview of the rules that were used to generate the Mizar and the Coq proof skeleton. Since in Coq there are no special keywords for uniqueness, existence or cases, these rules translate only the body of these nodes and add no keywords.

In table 11 we give a part of the Mizar and Coq skeletons for section 4.[7]

7d Completing the proofs in Coq

As we already explained, MathLang aims to remain as independent as possible of a particular foundation, while in addition facilitating the process of formalising mathematics in different theorem provers. Two PhD students of the MathLang project (Retel, respectively Lamar) are concerned with the MathLang paths into Mizar, respectively Isar. In this paper we study for the first time the MathLang path into Coq. We show how the CGa, TSa and DRa encoding of chapter one of Landau's book is taken into a fully formalised Coq code.

Currently we use the proof skeleton produced in the last section and fill all the `%body` parts by hand. We intend to investigate in the future how parts of the CGa and DRa annotations can be transformed automatically to Coq. In this section we explain why the process of formalising a mathematical text into Coq through MathLang is simpler than the formalisation of the text directly into Coq.

To begin with, we code the preface of the document (see table 7a). The most complicated section to code in Coq was the first one, because we had to translate the axioms in a way we can use them productively in Coq. We defined the natural numbers as an inductive set - just as Landau does in his book.

```
Inductive nats : Set :=
 | I : nats
```

[7] The complete output of the skeleton for Mizar and Coq for the whole chapter can be found in the extended article on the web pages of the authors.

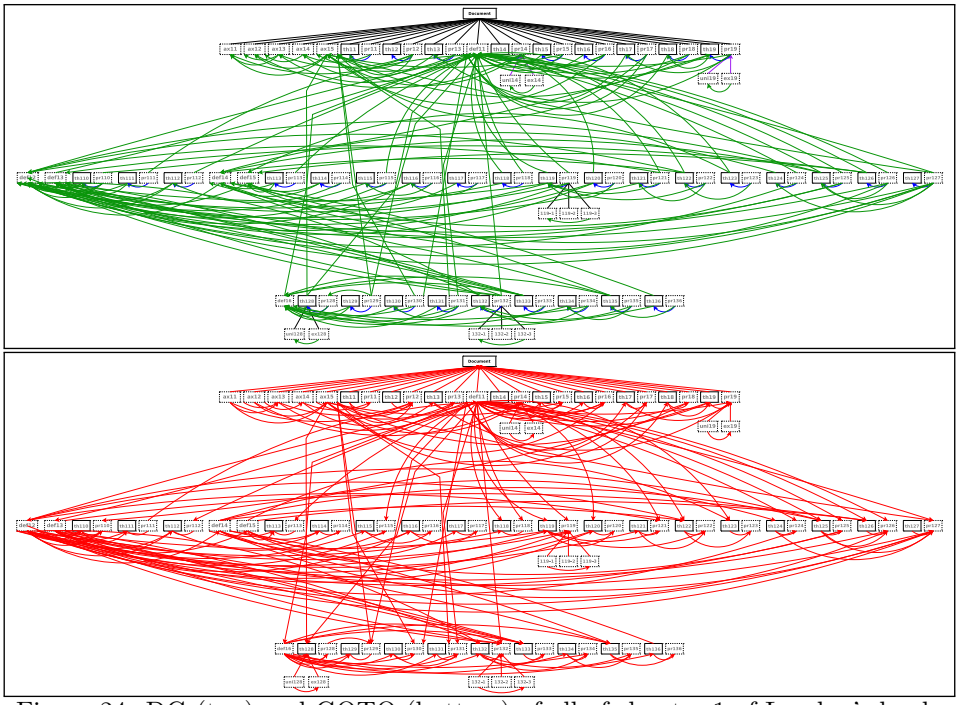
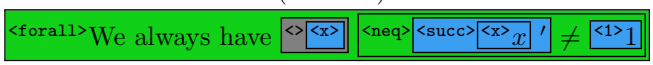


Figure 24. DG (top) and GOTO (bottom) of all of chapter 1 of Landau's book

```
| succ : nats -> nats
```

Then we translate axioms 2 - 4 almost literally from our CGa annotations. For example the annotation of Axiom 3 ("ax13") in our document is:



By just viewing the interpretations of the annotations we get:

```
forall x (neq (succ(x), 1))
```
(a)

The automatically generated Coq proof skeleton for this axiom is:

```
Axiom ax13 : <ax13> .
```
(b)

Now, we simply replace the `<ax13>` placeholder of (b) with the literal translation of the interpretations in (a) to get the valid Coq axiom (this literal translation could also be done by an algorithm that we plan to implement soon):

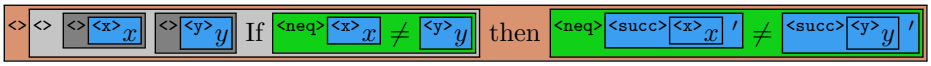
```
Axiom ax13 : forall x:nats, neq (succ x) I .
```

The other axioms could be completed in a similar way and as seen, this is a very simple process that can be carried out using automated tools that reduce the burden on the user (the proof skeleton is automated, the interpretations are obtained automatically from the CGa annotations which are simple to do, and for many parts of the text, the combination of the proof skeleton with the interpretations can also be automated).

Similarly for the theorems of chapter 1 of Landau's book, full formalisation is straightforward: E.g. Theorem 1 is written by Landau as:

$$\text{If } x \neq y \text{ then } x' \neq y'$$

Its annotation in MathLang CGa is:



The CGa annotation of the context (called local scoping) can also be seen as the premise of an implication. So the upper statement can be translated via a simple rewriting of the interpretations of the annotations to:

```
decl(x), decl(y) : neq x y -> neq (succ x) (succ y)
```

And when we compare this line with its Coq translation we see again, it is just a literal transcription of the interpretation parts of CGa and therefore could be easily performed by an algorithm.

```
Theorem th11 (x y:nats) : neq x y -> neq (succ x) (succ y) .
```

From the 36 theorems of the chapter 28 could be translated literally into their corresponding Coq theorems.

Now, we want also to look at a simple proof and how it can be translated into Coq. The encoding of Theorem 2 of the first chapter in Coq is

```
theorem th12 (x:nats) : neq (succ x) x .
```

Landau proves this theorem with induction. He first shows, that $1' \neq 1$ and then that with the assumption of $x' \neq x$ it also holds that $(x')' \neq x'$.

Since we defined the natural numbers as an inductive set, we can also do our proof in the Landau style. We introduce the variable x and eliminate it, which yields two subgoals that we need to prove. These subgoals are exactly the induction basis and the induction step.

```
Proof.
intro x. elim x.

2 subgoals
x : nats
____________________________________(1/2)
neq (succ I) I
____________________________________________________________(2/2)
forall n : nats, neq (succ n) n -> neq (succ (succ n)) (succ n)
```

To prove the first case, Landau used Axiom 3 which states, that for all x it holds that $x' \neq 1$. We can just apply this axiom in Coq to prove the first case:

```
apply ax13.

1 subgoal
x : nats
____________________________________________________________(1/1)
forall n : nats, neq (succ n) n -> neq (succ (succ n)) (succ n)
```

The next step is to introduce n as natural number and the induction hypothesis:

```
intros N H.

1 subgoal
x : nats
n : nats
H : neq (succ n) n
______________________________________(1/1)
neq (succ (succ n)) (succ n)
```

We see that this is exactly the second case of Landau's proof. He proved this case with Theorem 1 - we do the same:

```
apply th11.

1 subgoal
x : nats
n : nats
H : neq (succ n) n
______________________________________(1/1)
neq (succ n) n
```

And of course this is exactly the induction hypotheses which we already have as an assumption and we can finish the proof:

```
assumption.

Proof completed.
```

The complete theorem and its proof in Coq finally look like this:

```
Theorem th12 (x:nats) : neq (succ x) x .

Proof.
intro x. elim x.
apply ax13.
intros n H.
apply th11.
assumption.
Qed.
```

We also used another hint for translating from the CGa part to the Coq formalisation. When we have a Theorem of the following kind:

```
Theorem th11 (x y:nats) : neq x y -> neq (succ x) (succ y) .
```

This is equivalent to:

```
Theorem th11 : forall x y:nats, neq x y -> neq (succ x) (succ y) .
```

A proof of such a theorem always starts with the introduction of the universal quantified variables, so in this case x and y. In terms of Coq this means:

```
intros x y.
```

We can do this for every proof. If it is a proof by induction we can also choose the induction variable in the next step. For example if we have an induction variable x we would write:

```
elim x.
```

We took the proof skeleton for Coq and extended it with these hints and the straightforward encoding of the 28 theorems. The result can be found in the extended article on the authors' web pages. With the help of these hints we were able to produce 234 lines of correct Coq lines. The completed proof has 957

lines. In other words, we could automatically generate one fourth of the complete formalised text. This is a large simplification of the formalisation process, even for an expert in Coq who can then better devote his attention to the important issues of formalisation: the proofs.

Of course there are some proofs within this chapter whose translation is not as straightforward as the proof of Theorem 2 given above. But with the help of the CGa annotations and the automatically generated proof skeleton, we have completed the Coq proofs of the whole of chapter one in a couple of hours. Moreover, the combination of interpretations and proof skeletons can be implemented so that it leads for parts of the text, into automatically generated Coq proofs. This will speed further the formalisation and again will remove more burdens from the user. The complete Coq proof of chapter 1 of landau's book can again be found in the extended article on the authors' web pages.

8 CONCLUSION

MathLang and MPL are long-term projects and we expect there will be years of design, implementation, and evaluation, followed by repeated redesign, reimplementation, and re-evaluation. There are many areas which we have identified as needing more work and investigation. One area is improvements to the MathLang and MPL software (currently MathLang is based on the $\TeX_{\text{MACS}}$ editor) to make it easier to enter information for the core MathLang aspects (currently CGa, TSa and DRa). This is likely to include work on semi-automatically recognising the mathematical meaning of natural language text. A second area is further designing and developing the portions of MathLang and MPL needed for better support of formalisation. An issue here is how much expertise in any particular target proof system will be needed for authoring. It may be possible to arrange things in MathLang and MPL to make it easy for an expert in a proof system to collaborate with an ordinary mathematician in completing a formalisation. A third area where work is needed is in the overall evaluation process needed to ensure MathLang and MPL meet actual needs. This will require testing MathLang and MPL with ordinary mathematicians, mathematics students, and other users. And there are additional areas where work will be needed, including areas we have not yet anticipated.

The MathLang and MPL projects aim for a number of outcomes. MathLang aims to support mathematics as practised by the ordinary mathematician, which is generally not formalised, as well as work toward full formalisation. MPL aims to improve the interactive mathematical mode for proof assistants so that they can be user-friendly. We expect that after further improvements on the MathLang and MPL designs and software, writing MathLang documents (without formalising them) will be easy for ordinary mathematicians. MathLang and MPL aim to support various kinds of consistency checking even for non-formalised mathematics. MathLang and MPL will be independent of any particular logical foundation of mathematics; individual documents will be able to be formal in one or more

particular foundations, or not formalised.

MathLang and MPL hope to open a new useful era of collaboration between ordinary mathematicians, logicians (who ordinarily stay apart from other mathematicians), and computer science researchers working in such areas as theorem proving and mathematical knowledge management who can develop tools to link them together. MathLang and MPL's document representation are intended to help with various kinds of automated computerised processing of mathematical knowledge. It should be possible to link MathLang and MPL documents together to form a public library of reusable mathematics. MathLang and MPL aim to better support translation between natural languages of mathematical texts and multi-lingual texts. They also aim to better support the differing uses of mathematical knowledge by different kinds of people, including ordinary practising mathematicians, students, computer scientists, logicians, linguists, etc.

BIBLIOGRAPHY

[Abbott *et al.*, 1996] J. Abbott, A. van Leeuwen, and A. Strotmann. Objectives of openmath. Technical Report 12, RIACA (Research Institute for Applications of Computer Algebra), 1996. The TR archives of RIACA are incomplete. Earlier versions of this paper can be found at the "old OpenMath Home Pages" archived at the Uni. Köln.

[Autexier *et al.*, 2010] Serge Autexier, Petr Sojka, and Masakazu Suzuki. Foreword to the special issue on authoring, digitalization and management of mathematical knowledge. *Mathematics in Computer Science*, 3(3):225–226, 2010.

[Barendregt *et al.*, 2013] H Barendregt, Will Dekker, and Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2013.

[Barendregt, 2003] Henk Barendregt. Towards an interactive mathematical proof mode. In Kamareddine [2003], pages 25–36.

[Bundy *et al.*, 1990] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The oyster-clam system. In Mark E. Stickel, editor, *CADE*, volume 449 of *Lecture Notes in Computer Science*, pages 647–648. Springer, 1990.

[Cantor, 1895] Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre (part 1). *Mathematische Annalen*, 46:481–512, 1895.

[Cantor, 1897] Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre (part 2). *Mathematische Annalen*, 49:207–246, 1897.

[Cauchy, 1821] Augustin-Louis Cauchy. *Cours d'Analyse de l'École Royale Polytechnique*. Debure, Paris, 1821. Also in *Œuvres Complètes* (2), volume III, Gauthier-Villars, Paris, 1897.

[Constable and others, 1986] R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[Constable, 1995] Robert L. Constable. Using reflection to explain and enhance type theory. In H. Schwichtenberg, editor, *Proof and Computation*, Computer and System Sciences 139, pages 109–144. Springer, 1995.

[de Bruijn, 1987] N.G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In *Workshop on Programming Logic*, 1987. Reprinted in [Nederpelt *et al.*, 1994, F.3].

[Dedekind, 1872] Richard Dedekind. *Stetigkeit und irrationale Zahlen*. Vieweg & Sohn, Braunschweig, 1872. Fourth edition published in 1912.

[Frege, 1879] Gottlob Frege. *Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Nebert, Halle, 1879. Can be found on pp. 1–82 in [van Heijenoort, 1967].

[Frege, 1893] Gottlob Frege. *Grundgesetze der Arithmetik*, volume 1. Hermann Pohle, Jena, 1893. Republished 1962 (Olms, Hildesheim).

[Frege, 1903] Gottlob Frege. *Grundgesetze der Arithmetik*, volume 2. Hermann Pohle, Jena, 1903. Republished 1962 (Olms, Hildesheim).

[Gierz *et al.*, 1980] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. W. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, 1980.

[Gordon and Melham, 1993] M. Gordon and T. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Heath, 1956] Thomas L. Heath. *The 13 Books of Euclid's Elements*. Dover, 1956. In 3 volumes. Sir Thomas Heath originally published this in 1908.

[Kamareddine and Nederpelt, 2004] Fairouz Kamareddine and Rob Nederpelt. A refinement of de Bruijn's formal language of mathematics. *J. Logic Lang. Inform.*, 13(3):287–340, 2004.

[Kamareddine and Wells, 2008] Fairouz Kamareddine and J. B. Wells. Computerizing mathematical text with mathlang. *Electron. Notes Theor. Comput. Sci.*, 205:5–30, 2008.

[Kamareddine *et al.*, 2004a] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory from Its Origins Until Today*, volume 29 of *Kluwer Applied Logic Series*. Kluwer Academic Publishers, May 2004.

[Kamareddine *et al.*, 2004b] Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Flexible encoding of mathematics on the computer. In *Mathematical Knowledge Management, 3rd Int'l Conf., Proceedings*, volume 3119 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2004.

[Kamareddine *et al.*, 2004c] Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Mathlang: Experience-driven development of a new mathematical language. In *Proc. [MKMNET] Mathematical Knowledge Management Symposium*, volume 93 of *ENTCS*, pages 138–160, Edinburgh, UK (2003-11-25/---29), February 2004. Elsevier Science.

[Kamareddine *et al.*, 2006] Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Toward an object-oriented structure for mathematical text. In *Mathematical Knowledge Management, 4th Int'l Conf., Proceedings*, volume 3863 of *Lecture Notes in Artificial Intelligence*, pages 217–233. Springer, 2006.

[Kamareddine *et al.*, 2007a] Fairouz Kamareddine, Robert Lamar, Manuel Maarek, and J. B. Wells. Restoring natural language as a computerised mathematics input method. In MKM '07 [2007], pages 280–295.

[Kamareddine *et al.*, 2007b] Fairouz Kamareddine, Manuel Maarek, Krzysztof Retel, and J. B. Wells. Gradual computerisation/formalisation of mathematical texts into Mizar. In Roman Matuszewski and Anna Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 95–120. University of Białystok, 2007. Under the auspices of the Polish Association for Logic and Philosophy of Science.

[Kamareddine *et al.*, 2007c] Fairouz Kamareddine, Manuel Maarek, Krzysztof Retel, and J. B. Wells. Narrative structure of mathematical texts. In MKM '07 [2007], pages 296–311.

[Kamareddine, 2003] Fairouz Kamareddine, editor. *Thirty Five Years of Automating Mathematics*, volume 28 of *Kluwer Applied Logic Series*. Kluwer Academic Publishers, November 2003.

[Kanahori *et al.*, 2006] Toshihiro Kanahori, Alan Sexton, Volker Sorge, and Masakazu Suzuki. Capturing abstract matrices from paper. In *Mathematical Knowledge Management, 5th Int'l Conf., Proceedings*, volume 4108 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2006.

[Kohlhase, 2006] Michael Kohlhase. *An Open Markup Format for Mathematical Documents, OMDoc (Version 1.2)*, volume 4180 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2006.

[Lamar, 2011] Robert Lamar. *A Partial Translation Path from MathLang to Isabelle*. PhD thesis, Heriot-Watt University, Edinburgh, Scotland, May 2011.

[Landau, 1930] Edmund Landau. *Grundlagen der Analysis*. Chelsea, 1930.

[Landau, 1951] Edmund Landau. *Foundations of Analysis*. Chelsea, 1951. Translation of [Landau, 1930] by F. Steinhardt.

[Maarek, 2007] Manuel Maarek. *Mathematical Documents Faithfully Computerised: the Grammatical and Text & Symbol Aspects of the MathLang Framework*. PhD thesis, Heriot-Watt University, Edinburgh, Scotland, june 2007.

[MKM '07, 2007] *Towards Mechanized Mathematical Assistants (Calculemus 2007 and MKM 2007 Joint Proceedings)*, volume 4573 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.

[Nederpelt *et al.*, 1994] Rob Nederpelt, J. H. Geuvers, and Roel C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1994.

[Nederpelt, 2002] Rob Nederpelt. Weak Type Theory: a formal language for mathematics. Technical Report 02-05, Eindhoven University of Technology, 2002.

[Nipkow *et al.*, 2002] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[Peano, 1889] Giuseppe Peano. *Árithmetices Principia, Nova Methodo Exposita*. Bocca, Turin, 1889. An English translation can be found on pp. 83–97 in [van Heijenoort, 1967].

[Retel, 2009] Krzysztof Retel. *Gradual Computerisation and verification of Mathematics: MathLang's Path into Mizar*. PhD thesis, Heriot-Watt University, Edinburgh, Scotland, April 2009.

[Rudnicki, 1992] P. Rudnicki. An overview of the Mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[Sexton and Sorge, 2006] Alan Sexton and Volker Sorge. The ellipsis in mathematical documents. Talk overhead images presented at the IMA (Institute for Mathematics and its Applications, University of Minnesota) "Hot Topic" Workshop The Evolution of Mathematical Communication in the Age of Digital Libraries held on 2006-12-08/---09, 2006.

[Siekmann *et al.*, 2002] Jörg H. Siekmann, Christoph Benzmüller, Vladimir Brezhnev, Lassaad Cheikhrouhou, Armin Fiedler, Andreas Franke, Helmut Horacek, Michael Kohlhase, Andreas Meier, Erica Melis, Markus Moschner, Immanuel Normann, Martin Pollet, Volker Sorge, Carsten Ullrich, Claus-Peter Wirth, and Jürgen Zimmer. Proof development with omega. In Andrei Voronkov, editor, *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 144–149. Springer, 2002.

[Siekmann *et al.*, 2003] Siekmann, Benzmüllerand Fiedler, Meier, Normann, and Pollet. Proof development with ΩMEGA: The irrationality of $\sqrt{2}$. In Kamareddine [2003], pages 271–314.

[Team, 1999–2003] Coq Development Team. The coq proof assistant reference manual. INRIA, 1999–2003.

[van Benthem Jutting, 1977a] Lambert S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System*. PhD thesis, Eindhoven, 1977. Partially reprinted in [Nederpelt *et al.*, 1994, B.5,D.2,D.3,D.5,E.2].

[van Benthem Jutting, 1977b] Lambert S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system*. PhD thesis, Eindhoven, 1977.

[van der Hoeven, 2004] Joris van der Hoeven. GNU TeXmacs. *SIGSAM Bulletin*, 38(1):24–25, 2004.

[van Heijenoort, 1967] J. van Heijenoort. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, 1967.

[W3C, 2003] W3C. Mathematical markup language (MathML) version 2.0. W3C Recommendation, October 2003. W3C (World Wide Web Consortium).

[WC3, 2004] WC3. RDF Primer. W3C Recommendation, February 2004. W3C (World Wide Web Consortium).

[WC3, 2007] WC3. XQuery 1.0 and XPath 2.0 data model (XDM). W3C Recommendation, 2007. W3C (World Wide Web Consortium).

[Whitehead and Russel, 1910–1913] Alfred North Whitehead and Bertrand Russel. *Principia Mathematica*. Cambridge University Press, 1910–1913. In three volumes published from 1910 through 1913. Second edition published from 1925 through 1927. Abridged edition published in 1962.

[Wiedijk, 2006] F. Wiedijk, editor. *The Seventeen Provers of the World, foreword by Dana S. Scott*, volume 3600 of *LNCS*. Springer Berlin, Heidelberg, 2006.

[Zengler, 2008] Christoph Zengler. Research report. Technical report, Heriot-Watt University, November 2008.

[Zermelo, 1908] Ernst Zermelo. Untersuchungen über die Grundlagen der Mengenlehre (part 1). *Mathematische Annalen*, 65:261–281, 1908. An English translation can be found on pp. 199–215 in [van Heijenoort, 1967].

```
Lemma main :
(WCR R)->(IND R (cr R)).
Proof.
Intros.
Unfold IND.
Intro a.
Intro IH.
Unfold cr.
Intuition.

Assert
(a=b1\verb+\/+(TC R a b1))\verb+/\+
(a=b2\verb+\/+(TC R a b2)).
Split.
Apply p7.

Assumption.
Apply p7.
Assumption.
Tactic Definition
Get x :=
Elim x; Intros; Clear x.
Get H0; Get H3.
Exists b2.
Split.
Rewrite <- H0.
Assumption.
Apply TRC\_b.
Get H4.
Exists b1.
Split.
Apply TRC_b.
Rewrite <- H3.
Assumption.

Assert
(EX x1|(a R x1)/\(TRC R x1 b1)).
Apply p6.
Assumption.
Assert
(EX x2|(a R x2)/\(TRC R x2 b2)).
Apply p6.
Assumption.
Tactic Definition
Pick x y :=
Elim x; Intro y; Intros; Clear x.
Pick H4 x1; Pick H5 x2.
Intuition.

Assert
(EX x|(TRC R x1 x)/\(TRC R x2 x)).
Unfold WCR in H.
Apply (H a x1 x2).
Assumption.
Assumption.
Pick H4 x.
Intuition.
Assert (cr R x1).
Apply IH.
Assumption.
Assert
(EX b|(TRC R b1 b)/\(TRC R x b)).
Unfold cr in H.
Apply (IH x1).
Assumption. Split.
Assumption.
Assumption.
Pick H4 x.
Intuition.
Assert (cr R x1).
Apply IH.
Assumption.

Assert
(EX b|(TRC R b1 b)/\(TRC R x b)).
Unfold cr in H.
Apply (IH x1).
Assumption. Split.
Assumption.
Assumption.
Pick H10 b.
Intuition.
Assert (cr R x2).
Apply IH.
Assumption.

Assert
(EX c:A|(TRC R b2 c)/\(TRC R b c)).
Apply (H11 b2 b).
Split.
Assumption.
Apply (p4  x2 x b).
Assumption.
Assumption.
Pick H13 c.
Intuition.
Exists c.

Intuition.
Apply (p4 b1 b c).
Assumption.
Assumption.
Qed.

Theorem newman :
((WF R)/\(WCR R))->(CR R).
Proof.
Intros.
Intuition.
Assert (Ind R (cr R)).
Apply main.
Assumption.
Unfold CR.
Unfold WF in H0.
Apply (H0 (cr R)).
Assumption.
Qed.
```

Table 8. The Coq Script of Newman's Lemma

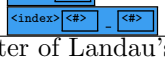
Group		Meaning	Encoding
Quantifiers	$\forall$	for all	
	$\exists$	exists	
	$\exists!$	exists exactly one	
Boolean connectives	$\wedge$	and	
	$\vee$	or	
	$\Longrightarrow$	implication	
	$\oplus$	exclusive or	
Set theory	$\in$	element of	
	$\subset$	subset of	
	$\{\mid\}$	constructor for a set	
	$\emptyset$	empty set	
	$=$	equality of sets	
	$\neq$	inequality of sets	
Special functions	$:=$	is a	
	1	one	
	$S(x)$	the successor function	
	_	function for indexing	

Table 9. The preface for the first chapter of Landau's book

Role	Mizar rule	Coq rule
axiom	`%name : %body ;`	`Axiom %name : %body .`
definition	`definition %name : %nl %body %nl end;`	`Definition : %body .`
theorem	`theorem %name: %nl %body`	`Theorem %name : %body .`
proof	`proof %nl %body  %nl end;`	`Proof %name : %body .`
cases	`per cases; %nl`	`%body`
case	`suppose %nl %body %nl end;`	`%body`
existencePart	`existence %nl %body`	`%body`
uniquenessPart	`uniqueness %nl %body`	`%body`

Table 10. The Mizar and Coq rules for the dictionary

```
theorem th131:
<th131>
proof
<pr131>
end;

theorem th132:
<th132>
proof
  per cases;
  suppose
  <pr132case1>
  end;
  suppose
  <pr132case2>
  end;
  suppose
  <pr132case3>
  end;
end;
```

```
Theorem th131: <th131> .

Proof.
<pr131>
Qed.

Theorem th132: <th132> .

Proof.

<pr132case1>

<pr132case2>

<pr132case3>

Qed.
```

Table 11. Part of the Mizar (left) and Coq (right) output from gSGA