A MathLang Path into a Coq Proof Skeleton

Fairouz Kamareddine[1], Joe B. Wells[1], and Christoph Zengler[2]

[1] ULTRA Group, MACS, Heriot-Watt University Edinburgh, Scotland
[2] Symbolic Computation Group, W. Schickard-Institute for Informatics, Universität Tübingen, Germany

Abstract. The computerization of mathematical texts is often a tedious and manual task. The MathLang System was developed to carry out this process in gradual steps. The idea is that the user can annotate an existing mathematical text with different types of information (grammatical/logical/rhetorical/etc.) and MathLang generates different computerised versions of this text which accommodate different levels of formality. So far there are paths from MathLang into skeletons for the Mizar and Isabelle proof checkers. In this paper we add new reasoning methods to MathLang's Document Rhetorical aspect (DRa) and develop a new path from an annotated text into a proof skeleton for the Coq proof assistant. We test and evaluate our new approach with the help of the first chapter of Landau's "Grundlagen der Analysis".

1 Background and motivation

MathLang is intended to support different degrees of formalisation and aims to make easier the partial or full formalisation of mathematical texts in some foundation. Furthermore, for documents where full formalisation is a goal, MathLang is intended to allow this to be accomplished in gradual steps. 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 is sometimes desirable for various reasons: it can be helpful with automated checking, semantics-based searching and querying, and interfacing with computer algebra systems (and other mathematical computation environments). In MathLang partial formalisation can be carried out to different degrees:

- The abstract syntax trees of symbolic formulas can be represented accurately. This is usually missing in systems like LaTeX or Presentation MathML, while more semantically oriented systems provide this to some degree. This can be used to provide editing support for algebraic rearrangements and simplifications, and can help with 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.
- A weak type system can be used to check simple grammatical conditions without checking full semantic sensibility.
- Justifications (inside proofs/between formal statements) can be linked (without always stating precisely how they are used). Uses of this feature include:
  - Extracting only those parts of a document that are relevant to specific results. (This could be useful in educational systems.)

- Checking that instances of circular reasoning are handled via induction.
- Calculating proof gaps as a first step toward fuller formalisation.

– If one commits to a foundation (or a family of foundations), one can start to use sophisticated type systems for checking more aspects of well-formedness.

The design of MathLang is (currently) divided into three *aspects*:

– The Core Grammatical aspect (CGa) [4,7] takes the best features of Weak Type Theroy [5] and MV [1] 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 is used to allow local definitions. The basic unit becomes the *step*, which can be either a definition, a statement (a phrase that asserts something), or a *block* which is merely a grouping of steps. We have nine different kinds of CGa annotations: term set noun adjective statement declaration definition step context. CGa provides a *grammar* for well-formed mathematics with grammatical categories and allows checking for basic well-formedness conditions (e.g., the origin of all names and symbols can be tracked).

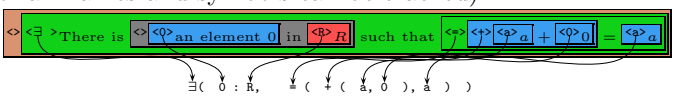


Fig. 1. Example of CGa encoding of mathematician's text

– The Text and Symbol aspect (TSa) [6,7,2] 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$_{\text{MACS}}$ with CGa data. Thanks to a notion of *souring* rules (called "souring" because it does the opposite of *syntactic sugaring*), TSa allows the structure of the mathematical text to follow the structure of mathematics as conceived by the mathematician.

– The Document Rhetorical aspect (DRa) [3,8] supports identifying portions of a text and expressing the relationships between them. Any portion of text (e.g., a phrase, a step, a block, etc.) can be given an identity and relationships can be expressed between identified pieces of text. For example, a chunk of text can be identified as a "theorem", and another as the "proof" of that theorem. Similarly, one chunk of text can be a "subsection" or "chapter" of another. This way, it is possible to do computations to check whether i) all dependencies are identified, ii) the relationships are sensible/problematic (hence whether the author should be warned), and to extract/explain the logical structure of a text. Such dependencies have been used in generating formal proof sketches and identifying the proof holes that remain to be filled.

In addition to the design of MathLang itself, we have worked on relating a MathLang text to a fully formalised version of the text. Using a CGa and DRa annotated text, we have given in [3] a procedure for producing a corresponding Mizar document, first as a proof sketch with holes and then as a fully completed proof. We have also worked on doing this with Isabelle [6]. Figure 2 diagrams the paths in MathLang. **In this paper, we make the following progress:**

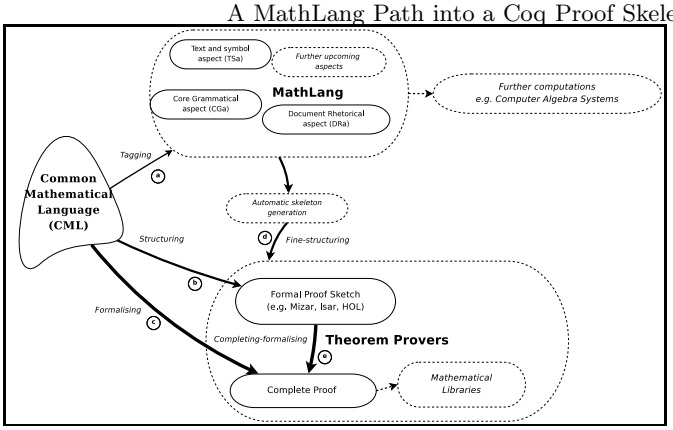


Fig. 2. Overall situation of work in MathLang

- Extending the formalisation and implementation of the DRa.
- Completing the path in MathLang in order to reach full formalisation.
- Introducing a third theorem prover (Coq) as a test bed for MathLang.
- Developing the Mizar proof skeleton of [3] into an automatically generated proof skeleton in a choice of theorem provers (Mizar, Isar and Coq, etc.). To achieve this, we give a generic algorithm for automatic proof skeleton generation which takes a DRa tree and the required prover as arguments.
- Giving hints for developing a generic algorithm to automatically convert parts of a CGa annotated text into the syntax of the prover in question.
- Giving an extensive example of how the mathematician's text passes through all the stages of MathLang from the original into the fully formalised text.

2 Extended Formalisation and Implementation of DRa

The DRa structure of a text can be represented as a tree (which is exactly the tree of the XML representation of the DRa annotated MathLang document). Due to this tree structure, we refer to an annotated part of a text as a DRa node (e.g., see figure 11). 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. For example in figure 4, the tree has 10 nodes. The root node (labelled Document) has four children nodes and five grandchildren nodes (all children of B). We distinguish between proved nodes (*theorem*, *lemma*, etc.) which have a solid line in the picture and unproved nodes (*axiom*, *definition*, etc.) which have a broken line. In order 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 returns an error if someone tries to prove an unproved node e.g. a definition or an axiom. When document D_2 references document D_1 it can reference the root node D_1 to include all of its mathematical text.

In figure 3 (taken from [3]), there are four top-level nodes: A, B, C and D, representing respectively lemma 1, its proof, corollary 2 its proof. 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, cases 1 and 2. The visual representation of this tree is on the lefthand-side of figure 4.

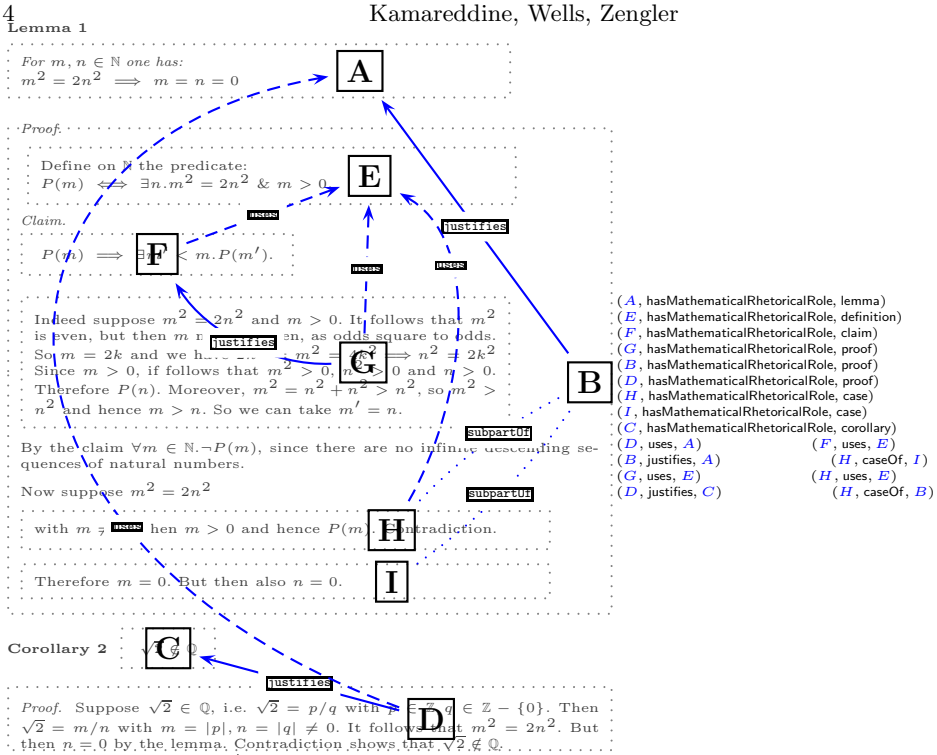


Fig. 3. Wrapping/naming chunks of text and marking relationships in DRa

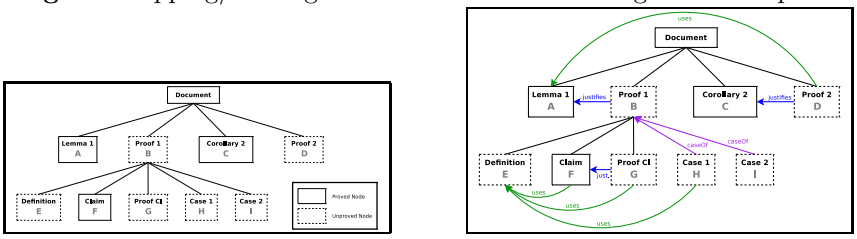


Fig. 4. Example of a tree of a document's DRa nodes and its dependency graph

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. We also have also an order of the nodes at the same level from left to right where we enumerate the children of a node form 1 to n and process them in this way. For figure 4, the pre-order yields the order A, B, E, F, G, H, I, C, D.

The DRa implementation automatically extracts a dependency graph (figure 4, righthand-side) that represents how the parts of a document are related.

Textual Order To examine the proper structure of a DRa tree we introduce the concept of textual order between two nodes in the tree. Textual order is a modification of the logical precedence presented in [3]. In this article we formalise this concept of order for the first time and show how we can process this information 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. To 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 A DRa node can also have further DRa nodes as children (e.g. B in figure 4 has the children E, F, G, H and I). We introduce different sets for a DRa node n. All these sets can be automatically generated from the user's CGa and TSa annotations of the document. Table 1 defines these sets and gives examples for the CGa annotated text in figure 5 which is the definition of the subset relation.

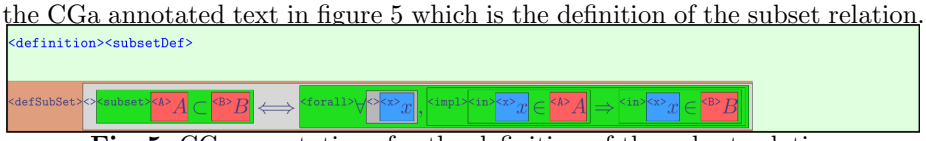


Fig. 5. CGa annotations for the definition of the subset relation

Set	Description	Example for fig. 5
$\mathcal{T}(n)$	$\{x \mid x$ 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$\}$	$\{A \subset B, x \in A, x \in B, x \in A \implies x \in B, \forall x(x \in A \implies x \in B)\}$
$\mathcal{DC}(n)$	$\{x \mid \exists q$ 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$\}$	$\{A \subset B \iff \forall x(x \in A \implies x \in B)\}$
$\mathcal{C}(n)$	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

To give further examples of $\mathcal{DC}(n)$ and $\mathcal{DF}(n)$, take this example (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 6 (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 the syntax introduced by the reader), is an identifier with a (possibly empty list of arguments) on the left-hand side followed by the symbol := and an expression on the right-hand side. The introduced symbol is the identifier of the left-hand side. For the example of figure 5 (call it ex''), the internal CGa representation is: $\subset(\texttt{A}, \texttt{B}) := \texttt{forall}(\texttt{a}, \texttt{impl}(\texttt{in}(\texttt{a}, \texttt{A}), \texttt{in}(\texttt{a}, \texttt{B})))$; the only introduced symbol is $\subset$ and hence $\mathcal{DF}(ex'') = \{\subset\}$.

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

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

Crucial to a context, step, definition, or declaration, is a set of term, set, noun, adjective, and statement. A DRa node n *uses* $\mathcal{USE}(n)$ 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 *$\bullet\mathcal{DF}(n) \cup \mathcal{DC}(n) \subseteq \mathcal{T}(n) \cup \mathcal{S}(n) \cup \mathcal{N}(n) \cup \mathcal{A}(n) \cup \mathcal{ST}(n)$ for every DRa node n. Furthermore, $\bullet\mathcal{IN}(n) \subseteq \mathcal{USE}(n)$ for every DRa node 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 6.

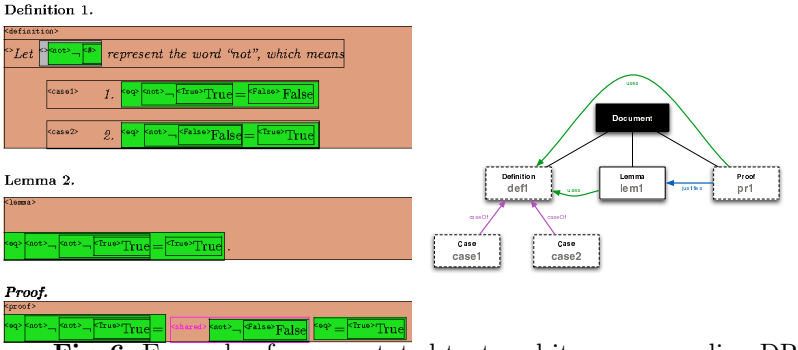


Fig. 6. Example of an annotated text and its corresponding DRa tree

Node n	$\mathcal{IN}(n)$	$\mathcal{USE}(n)$
def1	$\{\neg\} \cup \mathcal{IN}$(Case 1) $\cup\ \mathcal{IN}$(Case 2)	$\{\neg\} \cup \mathcal{USE}$(Case 1) $\cup\ \mathcal{USE}$(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

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 $\mathcal{ENV}(\mathsf{def1})$ for the node def1. The environment for case1 consists of $\mathcal{ENV}(\mathsf{def1})$ 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: $\mathcal{ENV}(\mathsf{lem1}) = \{\neg False, \neg False = True\} \cup \mathcal{ENV}(\mathsf{case2})$ and $\mathcal{ENV}(\mathsf{pr1}) = \{\neg\neg True, \neg\neg True = True\} \cup \mathcal{ENV}(\mathsf{lem1})$. With this information we derive the sets as shown in table 2 for the single nodes. We can now formalise for the first time three different kinds of textual order:

- **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 between two nodes induces a textual order. Table 3 gives standard relations and their textual order. To verify the relations of the example of figure 6 and their textual orders (Table 4), note that all five conditions hold and hence the relations are valid. For example the relation (case2, uses, lem1) is not

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

valid, because $\neg\exists x(x \in \mathcal{USE}(\mathsf{case1}) \wedge x \in \mathcal{IN}(\mathsf{lem1}))$. 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.

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 give a textual order $\prec, \preceq$ or $\leftrightarrow$. These orders 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 give a proof skeleton for a certain 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 already seen in our examples. There is also a relation between a DRa node and its children: For

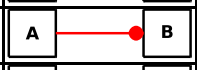
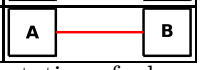
(A, uses, B)	$A \succ B$	
(A, caseOf, B)	$A \preceq B$	
$(A, \mathsf{justifies}, B)$	$A \leftrightarrow B$	

Table 5. Graphical representation of edges in the GoTO

each child c of 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 is useful e.g. in papers with a page limit, where parts of the text are relocated in the appendix but would be originally within the main text. The algorithm for producing the GoTO out of 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 4 we get the GoTO of figure 7. 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.

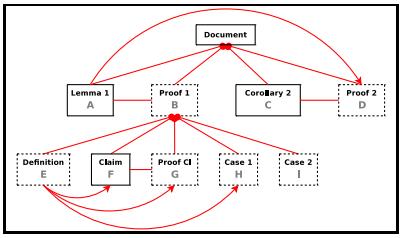


Fig. 7. Graph of Textual Order for an example DRa tree

Automatic checking of DG and GoTO We implemented two kinds of failures: warnings and errors. Currently, we check for four different kinds of failures:

i) Loops in the GoTO (error)
ii) Proof of an unproved node (error)
iii) More than one proof for a proved node (warning)
vi) Missing proof for a proved node (warning)

The checks for ii) - iv) are performed in the DG. For ii) 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 iii) and iv) 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 i) 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. But 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 8.

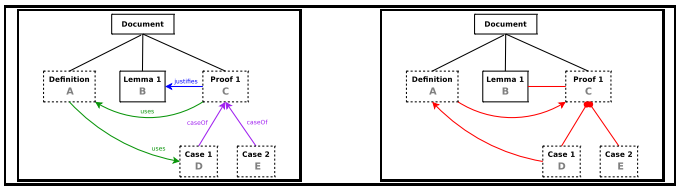


Fig. 8. Example of a not recognised loop in a DRa (left DG, right GoTO)

Apparently there is a cycle in this tree, because to be able to translate C we need to translate its children D and E. But before we can translate C we must have translated A because C uses A. But the child D of C is used by A. There we have a deadlock situation. Neither A nor C can be processed. To recognise such kinds of cycles we add certain edges to the GoTO when checking for cycles. We 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 is

done via algorithm 1. We could also add new edges for all incoming edges of the children c_i but this is not necessary because the textual order of the "childOf" relation is a directed edge from each child c_i to its parent node n. There we can use the transitivity of the edges and find a cycle anyway.

```
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

In the example of figure 8, algorithm 1 adds 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 gives figure 9. There you can clearly see a cycle between the nodes A, C and A with the edges A-C and C-A.

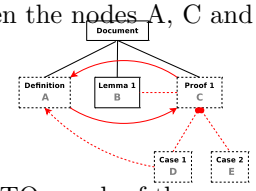


Fig. 9. GoTO graph of the example of figure 8 with added edges

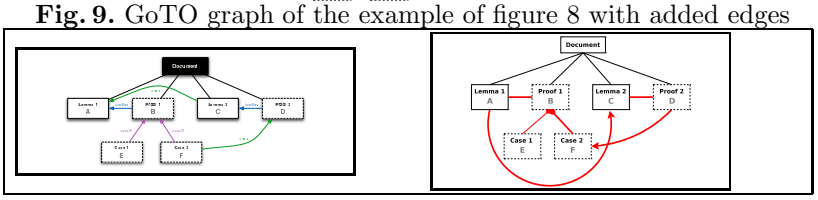


Fig. 10. Example of a loop in the GoTO (DG left, GoTO right)

Figure 10 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.

3 The Automated Skeleton Generation Algorithm

In this paper we report a generic algorithm gSGA for transforming the DRa tree into an automated proof skeleton for arbitrary theorem provers. 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.

Skeleton generation aims to stay as close as possible to the mathematician's original text. But due to some restrictions by different theorem provers the original order cannot always be respected. Our algorithm for rearranging parts of the text and generating the proof skeleton performs reordering only when necessary for the theorem prover at hand. We give examples when this can happen:

- Nested lemmas/theorems: Sometimes mathematicians define new lemmas or theorems inside proofs. Not every theorem prover can handle this (e.g. Coq). For such 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. references to already stated constructs).
- Outsourced proofs: A usual practise in mathematical writing is to outsource complex proofs which are not mandatory for the central results, in the appendix. During formalisation, these proofs need to be put in the right place.

```
while found white 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)

```
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)

The generation of the proof skeleton algorithm has two parameters: 1) the input MathLang XML file with DRa annotations and 2) a configuration file (in XML) for the theorem prover. A DRa node has one of three states: processed (black), in-process (grey) and unprocessed (white). A processed node has already been translated as a part of the proof skeleton, a node in-process is being checked, while an unprocessed node is awaiting translation. Algorithm 2 generates the output of a node where the generateOutput function is first performed for the Document root node. Then it is recursively performed for all the nodes in the DRa tree. That a node is ready to be processed depends only on the GoTO of the DRa tree. Algorithm 3 tests the three criteria below necessary for a node to be ready to be processed. Appendix A illustrates algorithm 3.

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

The transformation of the DRa annotated text into a proof skeleton has 2 steps:

- Reorder the text to satisfy the constraints of the particular theorem prover.

– Translate each DRa annotation to its corresponding syntax in the language of the theorem prover.

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

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/structural role of DRa. A single DRa node has two important properties: a name and a content. This is used in the translation. Within the configuration file we 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`. Take the example of the DRa node of figure 11. The role

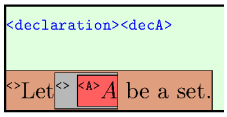


Fig. 11. An example for a single DRa node

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 is on the left-hand side and the rule for this translation is on the right-hand side:

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

The corresponding declaration and translation in Coq are:

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

Here, a single rule is embedded in an XML tag whose attribute "name" is:

```
<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/axioms:

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

The next question we 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.

```
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)

Every child of the DRa top-level node is 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 only appears 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 given in the GoTO. When a node is moved in front of its parent node, there is a $\prec$ edge between it and its former parent. So each sibling of the removed node from whom there is an incoming node is 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 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.

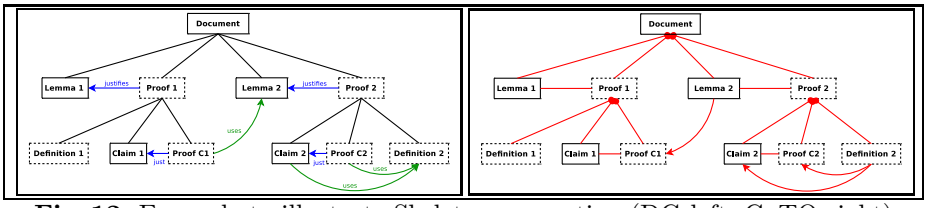


Fig. 12. Example to illustrate Skeleton generation (DG left, GoTO right)

We demonstrate the gSGA algorithm on the example from figure 12. First, 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 is in figure 13. 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 is given on the lefthandside of table 6. On the other hand, if we allow definitions to be nested but forbid

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 6. Outputs of the graphs of figures 13 (left) and 14 (right)

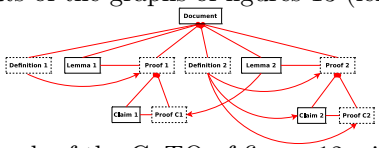


Fig. 13. A flattened graph of the GoTO of figure 12 without nested definitions

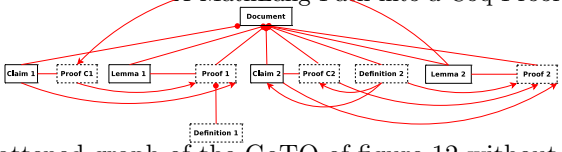


Fig. 14. A flattened graph of the GoTO of figure 12 without nested claims

nested claims, we get the graph of figure 14. 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 that its proof and Definition 2 have to be removed. The output for this graph is given on the righthandside of Table 6.

4 A full formalisation in Coq via MathLang

In this section we will take for the first time, the first chapter of Landau's book into all encoding levels in MathLang up to a full formalisation in Coq. We have given a complete CGa, TSa and DRa annotation for the chapter, have automatically generated (with gSGA) proof skeletons for Mizar and Coq, and have created a complete formalised version of the chapter in Coq. We used the MathLang TEX_{MACS} plugin to annotate the existing plaintext of the book.

To clarify the path we took, look at the overall diagram of the different paths in MathLang (figure 2). We first used path ⓐ and annotated the complete text with CGa, TSa and DRa annotations with the help of the MathLang TEX_{MACS} plugin. The second step was to automatically generate a proof skeleton of the annotated text. Using the proof skeleton and the CGa annotations we fully formalised the proofs in Coq completing the paths ⓓ and ⓔ. The final result was a complete formalised version of the first chapter of Landau's book in Coq. Appendix B explains the CGa and TSa annotations of the chapter in question.

DRa annotation The first section has five axioms which we annotate with the mathematical role "axiom", name "ax11" - "ax15" and classify as unproved nodes. In the following sections we have 6 definitions which we annotate with the mathematical role "definition", name "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 and a uniqueness part. This 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 cases which we annotate as unproved nodes with the mathematical role "case". This is translated in the Mizar "per cases" statement or in single proofs in Coq. The DRa tree for sections 1 and 2 is in figure 15.

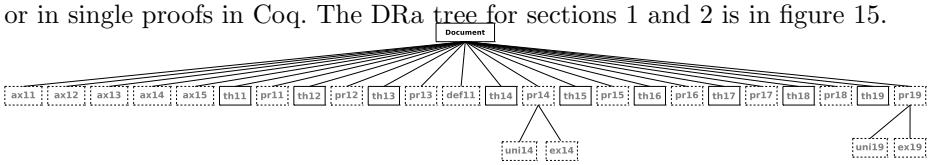


Fig. 15. The DRa tree of sections 1 and 2 of chapter 1 of Landau's book

Each proof *justifies* its corresponding theorem. Some 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 *uniqueness-PartOf* relations. Figure 16 gives the automatically produced DG of the DRa annotation of sections 1 and 2.[3]

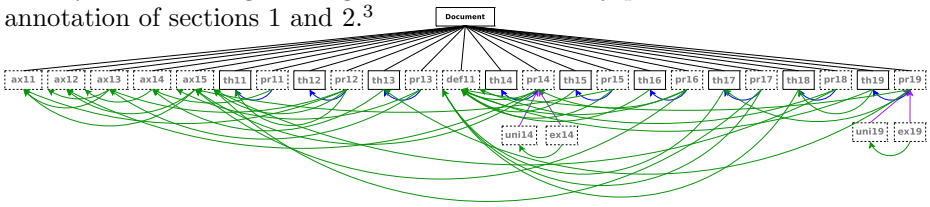


Fig. 16. The DG of sections 1 and 2 of chapter 1 of Landau's book

The GoTO is also produced automatically.[4] There are no errors or warning in the document which means no loops in the GoTO, no proofs for unproved nodes, no double proofs for a node and no missing proofs for proved nodes.

Automatically generated proof skeleton Since there are no errors in the GoTO, the proof skeleton can be produced without any warnings. We have 8 different mathematical roles in the document: axioms, definitions, theorems, proofs, cases, case, existenceParts and uniquenessParts. We distinguish 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 8 (appendix C) gives an overview of the rules 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 8 (appendix C) we give a part of the skeleton for section 4 (Mizar on the left, Coq on the right).[5]

In appendix C we show how the CGa, TSa and DRa encoding of chapter one of Landau's book is taken into a fully formalised Coq code, and we explain why formalising a mathematical text into Coq through MathLang is simpler than the formalisation of the text directly into Coq.

5 Conclusion

In this paper we gave the foundations behind the DRa annotation of a text and implemented the automatic checking of the DRa annotated text. We have also developed the proof skeleton idea given in [3] specifically for Mizar, into an automatically generated proof skeleton in a choice of theorem provers (including Mizar, Isar and Coq). To achieve this, we gave a generic algorithm for proof skeleton generation which takes the required prover as one of its arguments. We also gave hints for the development of a generic algorithm which is able to convert parts of a CGa annotated text automatically into the syntax of the theorem prover it is given as an argument. Furthermore, we have given the complete MathLang encoding and full formalisation into Coq via the MathLang

[3] The DG for the whole chapter can be found on our web page.
[4] The GoTO of the whole chapter can be found on our web page.
[5] The output of the skeletons for Mizar/Coq for the whole chapter is on our web page.

path of the first chapter of Landau's book showing how one can pass through all the path of MathLang from the version written by the mathematician into a fully formalised text. We gave examples to illustrate that this formalisation into Coq using the MathLang path is easier than a direct formalisation into Coq.

To get the full Coq formalisation of the first chapter, we took the proof skeleton for Coq and extended it with a number of hints. With these hints we were able to produce 234 lines of correct Coq lines out of the 957 lines of the complete proof. That is, we could automatically generate one fourth of the complete formalised text. This simplifies indeed the formalisation process, even for the Coq expert who can then better devote his attention to the important issues of formalisation: the proofs.

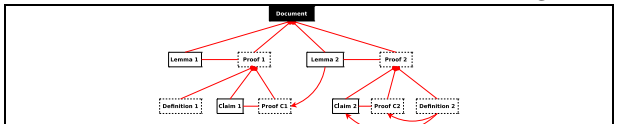
Of course there are some proofs within this chapter whose translation is not as easy and straightforward as the proof of Theorem 2 given in Appendix C. But with the help of the CGa annotations and the automatically generated proof skeleton, we have completed the Coq proofs of the whole of chapter one in a couple of hours. As we said above, 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.

References

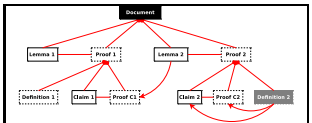
1. de Bruijn, N.: The mathematical vernacular, a language for mathematics with typed sets. In: Workshop on Programming Logic (1987), reprinted in [10, F.3]
2. Kamareddine, F., Lamar, R., Maarek, M., Wells, J.B.: Restoring natural language as a computerised mathematics input method. In: MKM '07 [9], pp. 280–295
3. Kamareddine, F., Maarek, M., Retel, K., Wells, J.B.: Gradual computerisation/-formalisation of mathematical texts into Mizar, Studies in Logic, Grammar and Rhetoric, vol. 10, pp. 95–120. University of Białystok (2007), under the auspices of the Polish Association for Logic and Philosophy of Science
4. Kamareddine, F., Maarek, M., Wells, J.B.: Toward an object-oriented structure for mathematical text. In: Mathematical Knowledge Management, 4th Int'l Conf., Proceedings. Lecture Notes in Artificial Intelligence, vol. 3863, pp. 217–233. Springer (2006)
5. Kamareddine, F., Nederpelt, R.: A refinement of de Bruijn's formal language of mathematics. J. Logic Lang. Inform. 13(3), 287–340 (2004)
6. Lamar, R.: *A MathLang Path to Isabelle*, Submitted for PhD thesis, Heriot-Watt University, Edinburgh, Scotland, 2011.
7. Maarek, M.: *Mathematical Documents Faithfully Computerised: the Grammatical and Text & Symbol Aspects of the MathLang Framework*, PhD thesis, Heriot-Watt University, Edinburgh, Scotland, 2007.
8. Retel, K.: *A computerisation path from Mathlang to Mizar*, PhD thesis, Heriot-Watt University, Edinburgh, Scotland, 2008.
9. Mathematical Knowledge Management, 6th Int'l Conf., Proceedings, Lecture Notes in Artificial Intelligence, vol. 4573. Springer Berlin / Heidelberg (2007)
10. Nederpelt, R., Geuvers, J.H., de Vrijer, R.C.: Selected Papers on Automath, Studies in Logic and the Foundations of Mathematics, vol. 133. North-Holland, Amsterdam (1994)

A Illustrating the algorithms

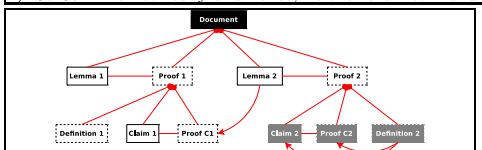
To illustrate algorithm 3, take the DRa tree of a (typical and not well structured) mathematical text with its DG and GoTO edges as in figure 12.



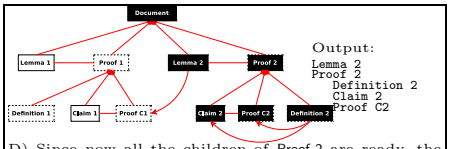
A) The root node of the document can be marked as processed and the algorithm starts at this node. The first child is Lemma 1. Criterion 1) is fulfilled, since the node has no incoming $\prec$ edges in the GoTO. Criterion 2) is fulfilled because the node has no children. For criterion 3) Proof 1 has 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.



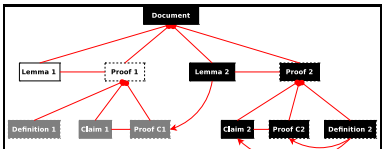
B) The next Node to check is Lemma2. Criteria 1) and 2) are fulfilled, for criterion 3) the proof Proof 2 has to be ready to be processed. Criteria 1) and 3) of the proof are fulfilled, so its children have to be ready to be processed. The first child that can be processed is Definition 2. So it is marked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



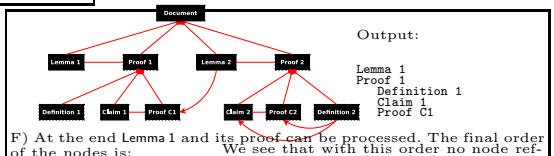
C) In a second run of the for loop for checking the children of Proof 2, Claim 2 and its proof are now ready, because Definition 2 is not white anymore but grey. This situation is the reason, why we must perform the check whether the n children of a node are ready n times exactly.



D) 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).



E) 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.



F) 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.

B CGa and TSa annotations of the first chapter

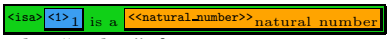
The Preface In the preface of a MathLang document we give symbols that are used but not defined in the text. These are often quantifiers or Boolean connectives and are often pre-encoded in theorem provers (e.g. Coq has special symbols for and, or, implication, etc.). The preface of the first chapter of Landau's book consists of 17 symbols and is given in the table below.

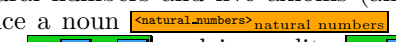
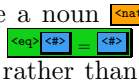
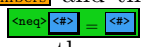
Group	Meaning	Encoding
$\forall$	for all	<forall> $\forall$ <#> <#> . <#>
$\exists$	exists	<exists> $\exists$ <#> <#> . <#>
$\exists!$	exists exactly one	<exists_one> $\exists!$ <#> <#> . <#>
$\in$	element of	<in> <#> $\in$ <#>
$\subset$	subset of	<subset> <#> $\subset$ <#>
$\{\mid\}$	set constructor	<<Set>> { <#> <#> | <#> }
$\emptyset$	empty set	<emptyset> $\emptyset$
$=$	set equality	<seteq> <#> $=$ <#>
$\neq$	set inequality	<setneq> <#> $\neq$ <#>

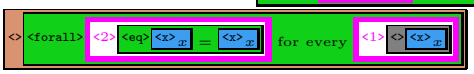
Group	Meaning	Encoding
$\wedge$	and	<and> <#> $\wedge$ <#>
$\vee$	or	<or> <#> $\vee$ <#>
$\Longrightarrow$	implication	<impl> <#> $\Longrightarrow$ <#>
$\oplus$	exclusive or	<xor> <#> $\oplus$ <#>
$:=$	is a	<isa> <#> isa <#>
1	one	<1> 1
$S(x)$	successor function	<succ> <#>
_	indexing function	<index> <#> _ <#>

Table 7. The preface for the first chapter of Landau's book

Two functions deserve explanation:

1. The "is a" function expresses that a term is an instance of a noun. E.g. the first axiom of the book is *1 is a natural number*, and its encoding is 
2. The "index" function expresses a notion like of $a_b = c$ which can be defined as $index(a, b) = c$. It takes two terms as argument and returns a term.

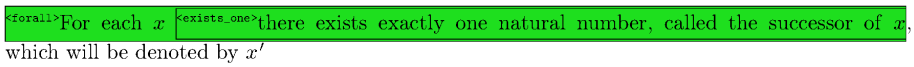
The first section The first section introduces the natural numbers, equality on natural numbers and five axioms (an extension of the Peano axioms). We introduce a noun  and the set  of natural numbers. Equality  and inequality  between natural numbers are declared rather than defined. There are three properties of equality, which we encoded. We will show one such encoding to demonstrate TSa sharing and the use of the symbols of the preface. The original statement is $x = x$ for every x which is equivalent to $\forall x(x = x)$. Since the positions are swapped in Landau's text we use the position souring annotation of this statement: 

This yields the final statement 

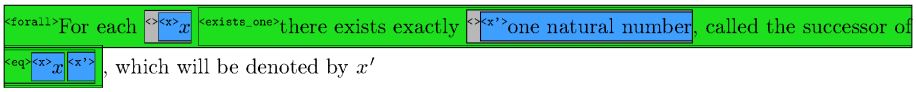
Next we show how to encode axiom 2 which illustrates that "wordy" parts can also be annotated, not only mathematical statements. Axiom 2 is:

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

The general structure of axiom 2 is:



We encode axiom 2 (which is equivalent to $\forall x(\exists! x'(succ(x) = x'))$ as:



Sections 2 - 4 Next, 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. We show how we annotate the theorem of figure 17. x and y are annotated as terms, plus as a function which takes two terms and

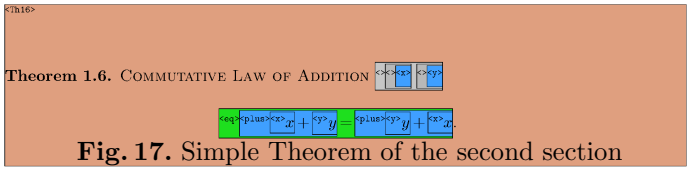


Fig. 17. Simple Theorem of the second section

returns a term. Equality between terms is a statement. Since we did not declare x and y in the preface or in a global context we use a local scoping where in the first annotated line 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.

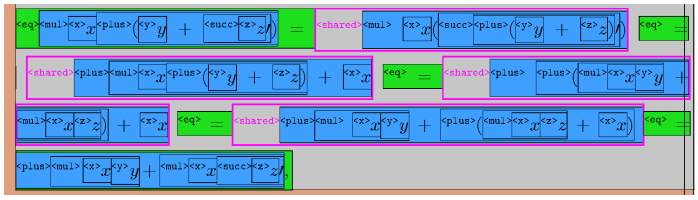


Fig. 18. Souring in chains of equations

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

To save time when annotating documents, we use our souring methods (e.g. sharing variables, cf. figure 18 and swapping positions as in $x = x$ *for every* x).

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

If $x \leq y,\ y < z$ or $x < y,\ y \leq z$
then $x < z$

We annotate the premise of the theorem as a disjunction of two conjunctions as seen in figure 19. Another use of Boolean connectives is when we have formu-

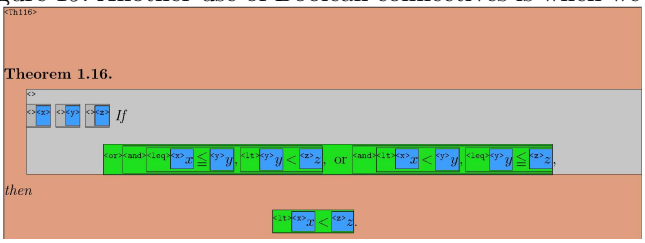


Fig. 19. The annotated Theorem 16 of the Landau's first chapter

lations 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 B).

C Completing the proofs in Coq

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

Mizar/Coq rules for the dictionary

```
theorem th131:
<th131>              Theorem th131: <th131> .
proof
<pr131>              Proof.
end;                 <pr131>
theorem th132:       Qed.
<th132>
proof                Theorem th132: <th132> .
  per cases;
  suppose            Proof.
  <pr132case1>
  end;               <pr132case1>
  suppose
  <pr132case2>       <pr132case2>
  end;
  suppose            <pr132case3>
  <pr132case3>
  end;               Qed.
end;
```

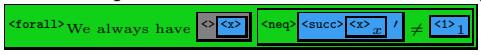
Table 8. Part of the Mizar (left) and Coq (right) output from gSGA

Currently we use the proof skeleton discussed in section 4 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 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 B). 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
 | 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:



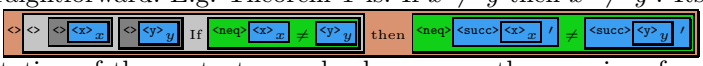
By just viewing the interpretations of the annotations we get (a) its automatically generated Coq proof skeleton (b):

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

```
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, the work needed to get the full formalisation is straightforward: E.g. Theorem 1 is: If $x \neq y$ then $x' \neq y'$. Its CGa annotation is 

The CGa annotation of the context can also be seen as the premise of an implication. So the upper statement can be translated to (again we see here that this is only a simple rewriting of the interpretations of the annotations):

```
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 look at how a simple proof 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 which are 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)
```

Landau proved the first case with the help of Axiom 3 which states, that for all x it holds that $x' \neq 1$. We 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 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 x is an induction variable we 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. That is, one fourth of the complete formalised text automatically generated. 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 easy and 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. As we said above, 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.