Assisted Proof Document Authoring

David Aspinall[1], Christoph Lüth[2], and Burkhart Wolff[3]

[1] LFCS, School of Informatics, The University of Edinburgh, U.K.
[2] Department of Mathematics and Computer Science, Universität Bremen, Germany
[3] Department of Computer Science, ETH Zürich, Switzerland

Abstract. Recently, significant advances have been made in formalised mathematical texts for large, demanding proofs. But although such large developments are possible, they still take an inordinate amount of effort and time, and there is a significant gap between the resulting formalised machine-checkable proof scripts and the corresponding human-readable mathematical texts. We present an authoring system for formal proof which addresses these concerns. It is based on a *central document* format which, in the tradition of literate programming, allows one to extract either a formal proof script or a human-readable document; the two may have differing structure and detail levels, but are developed together in a synchronised way. Additionally, we introduce ways to assist production of the central document, by allowing tools to contribute *backflow* to update and extend it. Our authoring system builds on the new PG Kit architecture for Proof General, bringing the extra advantage that it works in a uniform interface, generically across various interactive theorem provers.

1 Introduction

While computer-supported proof assistants are increasingly accepted in computer science, in particular in the field of formal methods, their potential for mathematical practice is only beginning to be recognised [20]. Several substantial proofs reaching hundreds or thousands of pages like the Four Colour Problem [6] or the Prime Number Theorem [11] have been formalised with the aid of systems like Coq or Isabelle, and others like the Kepler Conjecture are currently under development [13]. It has been suggested that computer assistants could be generally accepted in mathematical practice if authors with no prior expertise in theorem could formalise mathematical proof texts at an effectivity estimated at one page of mathematical text per day [6].

This formalisation rate is not reached by contemporary systems, and there are two important areas that need work. First, we need to provide systems with a higher degree of automation so that more trivialities can be discharged with less work. Second, we need to increase the user's productivity by making it easier to construct formal proofs, assisting the writing process. The first point has been a focus for theorem prover development in the last few years, but the second point has received less attention. In general, interface technology has been quite neglected; many interfaces still use arcane command line syntax and basic text editors, which do not reach the same levels of productivity as e.g., integrated development environments (IDEs) used in software development. Modern IDEs for programming provide sophisticated mechanisms to assist writing code, for example, constructing templates automatically from graphical models or helping the user to search for library functions and documentation very swiftly. Clearly, much more could be done to assist the user in proof document authoring.

We start by taking a single proof document as the central purpose of the development: so-called *document-centred authoring.* In our sense, an *authoring system* is a set of tools which assist the user in constructing the central document, maintaining the consistency of the development and documentation under change, and generating the *views* which allow fine-grained interactive exploration of the proof detail, animating proof checking in various ways. A machine-checkable proof script and a human-readable document describing its content are just two different views of one document. The authoring assistance should allow powerful graphical user interface techniques such as drag-and-drop and point-and-click [7, 18, 1], going beyond mere text editing. Moreover, an authoring system assists the user by allowing other tools, in particular the prover itself, to edit the document as well, thus increasing productivity. We call the mechanism for this *backflow.*

The context of this work is a software framework for conducting interactive proof called the *Proof General Kit* (PG Kit). The main new contribution, as presented in this paper, is the extension for assisted authoring with backflow.

Outline. Sect. 2 motivates document-centred authoring and backflow. In Sect. 3 we describe the PG Kit architecture, and its extensions for authoring. To demonstrate the viability of our approach, we develop use cases for literate proving and script generation in Sect. 4 and Sect. 5 respectively. Sect. 6 concludes with a survey of related work and an outlook on future work.

2 Document-Centred Authoring and Backflow

A *proof script* is a formal text which can be run through a proof assistant to mechanically check the validity of the proofs therein. A proof script usually does not contain the proofs themselves, just enough information to construct them. Some formal proof languages do contain structuring mechanisms inspired by human proofs, but, unfortunately, the formal syntax and in particular the level of detail required by a proof assistant usually still precludes a proof written in such a language to be accepted as a "textbook proof" by non-expert human readers. In contrast, we call a proof document *human-readable* if it is aimed at a presentation close to textbook proofs or journal papers; typically it may contain a higher level of abstraction, leave out repetitive arguments, and even omit logical steps considered distracting.

Considerable effort has been devoted to bridging the gap between human-readable and machine-checkable proof, either by making practical mathematical

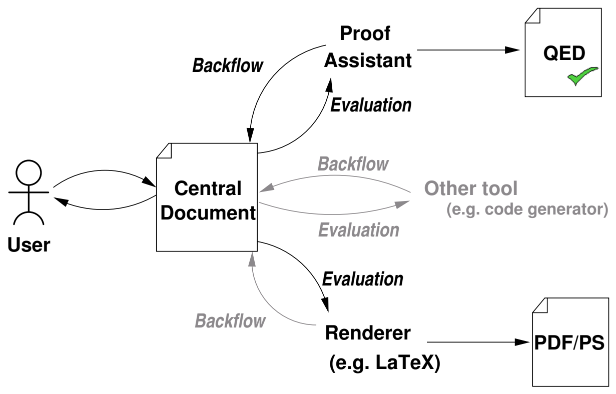


Fig. 1. Control and document flow for document-centred authoring

language more strict and hence machine-checkable, or by making prover input languages less formal, more abstract, and hence more human-readable (see the related work in Sect. 6.1). But the principal dilemma of different requirements from both human readers and proof assistants remains.

An alternative approach is to accept the dichotomy of presentation levels, and adapt techniques similar to those of literate programming [16] to weave structured, human-readable annotations into formal proof scripts. This has already been used, for example, in literate specification environments such as HOL-Z [8] and Isar's integrated LaTeX output mechanism [24], where terms, formulas or proof-states can be generated into the output during the LaTeX-rendering phase.

The underlying idea of these and other approaches, which also underlies our own, is that we have one *central document* from which we can extract both human-readable text and a machine-checkable proof script. This is the *document-centred* approach as depicted in Fig. 1. The user edits the document in a suitable editing environment, and the document can be evaluated by various tools, such as the *proof assistant* which checks that the document contains valid proofs, a *renderer* (e.g., LaTeX) which typesets or renders the document into human-readable documentation readable outwith the system, or other tools, for example a code generator to construct executable versions of specifications.

Our main contribution to this setting is to allow the possibility of *backflow* from each tool into the central document, i.e. each tool can generate text which in turn becomes part of the document. In contrast to the mentioned Isar mechanism, backflow is supported during editing and not during LaTeX-rendering. In the case of the prover, the backflow can generate parts of the central proof document to assist in writing the proof script. We concentrate on this case in Sects. 4 and 5 below, but note that the backflow can equally well originate from tools other than the prover. For example, the LaTeX component may have generated cross-references in previous runs which were offered in a context sensitive way when editing the central document.

Importantly, in contrast to classical literate programming, the backflow assistance has to be *interactive*. The user needs an immediate reaction from the system, such as searching for applicable lemmas or inspecting terms and their types. Interactive development also means that authors can annotate their proofs while developing them, which is easier than to annotate them afterwards.

Thus, the technical challenges we face when implementing a system to support assisted document-centred proof document authoring are *interactivity*, with the document being developed *incrementally*, *synchronisation* of the different views of the document, and the *coordination* of information flow between the different views. Our system architecture is designed to meet these challenges.

3 PGIP and the PG Kit Architecture

The *Proof General Kit* (PG Kit) is a software framework for conducting interactive proof. It evolved from the *Proof General* project, which constructed a generic interface to numerous interactive theorem provers in a piecemeal approach, by individual customisation for communication with each proof assistant. PG Kit is instead based on a uniform mechanism, specifying the syntax of messages exchanged between components and the protocol governing message exchanges. This section introduces just what is needed; full details are elsewhere [2, 3, 4].

Fig. 2 shows the component-based PG Kit architecture, which closely mirrors the document flow of Fig. 1. The *broker* middleware component handles the central document; it is responsible for managing the synchronisation between different views. The *display* components on the left-hand side interact with the user. On the right side, we have *proof assistants* or other tools. Each display may implement a different interaction paradigm: a text editor (e.g., Emacs) based on textual input and cryptic key sequences; a GUI (e.g., PGWin [3]) using graphical techniques such as drag-and-drop and point-and-click to construct proofs, or a generic IDE (e.g., Eclipse [26]) with sophisticated navigation and project management, using both graphical and textual interaction.

3.1 The Message Protocol PGIP

The mechanism for directing proof used by PG Kit is known as the **PGIP** protocol, for *Proof General Interactive Proof.* The order of message exchanges is given by an informal specification [2] and enforced dynamically by the central broker component. The syntax of PGIP messages is defined by an XML schema

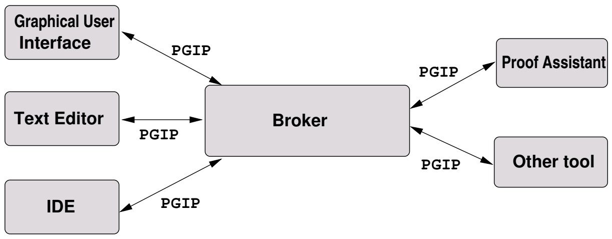


Fig. 2. PG Kit System Architecture

written in RELAX NG [21]. There is a secondary schema called **PGML**, for *Proof General Markup Language*, which is used for annotating concrete syntax within messages (for example, to generate clickable regions) and to represent mathematical symbols.[1] To define the protocol, we distinguish several kinds of messages, including: *display commands* which are sent from the display to the broker, arising from user input; *display messages* sent to the display from the prover or broker, which contain output for the user, and *prover commands* which are sent to the prover and affect the internal (proof-relevant) state of the prover.

Messages are exchanged over channels implemented as Unix pipes or sockets. Compared with simple RPC mechanisms like XML RPC, PGIP message exchange is more permissive, allowing multiple responses. We need this because interactive provers may send a lot of information while a proof proceeds, and a proof may be slow or even diverge (e.g., in proof search). It is essential that this feedback is displayed eagerly so the user can take action as soon as possible.

3.2 The Central Document

The central document is the main artefact of the system. The two principal views on the central document are the machine-checked *proof script* consisting of prover commands, and the human-readable *documentation.* These are extracted from relevant parts of the central document. Note that all document content is in principle free-form and manually generated, but the backflow concept allows tools to assist the user in constructing both proof script and documentation.

PGIP manipulates the central document in an unspecified concrete syntax (subject to a few constraints) by *marking up* the contents with PGIP commands that give the document the structure needed. Fig. 3 shows a proof script in a fictional simple tactical language and its markup in PGIP.[2]

```
goal "length (tl xs) = length xs - 1"
 /* proof by case distinction, then it's trivial */
 case_tac "xs" THEN simp_tac THEN simp_tac
 qed "Simple"
```

```
<opengoal>goal "length (tl xs) = length xs − 1"</opengoal>
<comment> /∗ proof by case distinction, then it's trivial ∗/</comment>
<proofstep> case_tac "xs" THEN simp_tac THEN simp_tac</proofstep>
<closegoal> qed "Simple"</closegoal>
```

Fig. 3. A proof script and its PGIP markup

The proof script mark-up is more fine-grained than the documentation, because it needs to be evaluated interactively. Typical proof script markup are the elements <`opengoal`>, <`proofstep`> and <`closegoal`>, which start a proof,

[1] MathML is another possibility, but PGML should be easier for existing systems.

[2] To save head scratching: this is provable with `tl []=[]` and `(0-1)::nat=0`.

perform a proof step, and end a proof, but also markup for the start and end of a theory etc. Documentation is marked up as a `<litcomment>` element (not yet shown), and proper comments, for the author's eyes only, are marked up as `<comment>`. The corresponding (trivial) proof document could read:

> **Lemma** 3.14 (Simple): "length (tl xs) = length xs - 1".
> **Proof:** trivial.

It is not very enlightening for the human reader, neither revealing the main argument, the proof structure, nor the lemmas and definitions used in the proof. We will show refinements of this running example later on.

3.3 Interaction and Authoring

When the broker reads a document or when the user edits the document, it is first parsed, causing the PGIP markup to be inserted. On the marked up document, we can perform the following operations:

– *Interactive evaluation* by the prover. The broker does this on user request, by sending parts of the script to the prover for evaluation (using a simple linear notion of dependency, or a more fine grained dependency analysis if supported by the prover); it corresponds to 'stepping through the proof', and supports the incremental development of proof scripts.
– *Extracting the proof script* by removing all annotation comments and PGIP markup. We obtain a proof script which we can feed directly to the prover, without broker intervention, to *validate* it.
– *Extract the documentation* by extracting all literate comments and (possibly) interleaving formatted prover commands. We obtain a document which we can render to produce a human-readable documentation.

The document may be updated by user editing as usual, or by the *backflow* mechanism. Backflow is characterised by what part of the document it targets: *documentation backflow* contains documentation content, e.g. a display of the current proof state or a named theorem or constant definition, which becomes part of the documentation; whereas *script backflow* contains proof script content, which is provided to assist the user in constructing a proof (typically by the prover itself), and which in turn becomes prover input.

Documentation backflow is treated specially. To help with synchronisation we want to record requests for documentation backflow in the document itself. Then we can regenerate those parts of documentation when the proof is rerun or adjusted: the proof state display which we have inserted previously might have changed, for example. In the same way that the broker tracks the status of prover commands (described in [4]), it can track the status of those parts of the document generated by backflow to see if they are up-to-date or not.

3.4 Proof Commands and Operations

The proof script part of a document contains a sequence of *prover commands* in PGIP, but not all prover commands can appear in a proof script. We distinguish *proper* commands which can appear from *improper* commands which cannot.

The broker does not know about the concrete syntax of the system, so we provide a way to construct them by filling in configurable templates with identifiers and raw text. An <operationsconfig> configuration message provides a prover-specific set of *prover types* and *prover operations*. The prover types (not to be confused with the theorem prover's *logical* notion of type, if it has one!) are used to provide context menus, icons, and drag-and-drop actions (cf [18]). Prover operations may be used to build up commands by textual substitution. They can be bound to input events, and may then be invoked by a menu item or drag-and-drop.

The improper commands are used for controlling and inspecting the prover's state, and cannot appear in the proof script being developed. A standard improper command is <undostep> which undoes the last proof step in a development. In the next section we introduce the idea of allowing configurable improper commands to generate *backflow* for feeding back into the document. This is a much more powerful way of generating prover commands than the <operationsconfig> templates because it can be context sensitive and involve arbitrary external tools.

3.5 Extending PGIP: Interactions for Authoring

The extensions for assisted authoring comprise the <litcomment> element and the backflow. They are not part of the original design, but backwards compatible.

As mentioned above, the documentation is generated from parts of the document which the prover does not see. We call these parts *literate comments* to distinguish them from ordinary comments which are not part of the documentation. A literate comment contains either text or documentation backflow-generating *directives*. A directive contains the PGIP command which generates the documentation (these exist already in PGIP as the `proofctxt` entity), and the resulting markup in PGML. An example of a `proofctxt` element is <showproofstate> which embeds the current proofstate in the document. Here is a fragment of the RELAX grammar for the new commands:[3]

```
litcomment = element litcomment { format_attr?, (text | directive)* }
directive = element directive { (proofctxt, pgml) }
format_attr = attribute format { token }
```

The format attribute can be used to specify the output format if the prover supports more than one output format, e.g. LaTeX, HTML or plain text. We also allow all proper proof commands to have an optional `nodisplay` attribute, e.g. for <opengoal>:

```
opengoal = element opengoal { display_attr?, thmname_attr?, text }
display_attr = attribute nodisplay { xsd:boolean }
```

The `nodisplay` attribute allows us to suppress proof commands for document output (e.g., to replace "`by simp_tac`" with "*Proof is obvious*").

[3] See [21] to better understand the format of these rules.

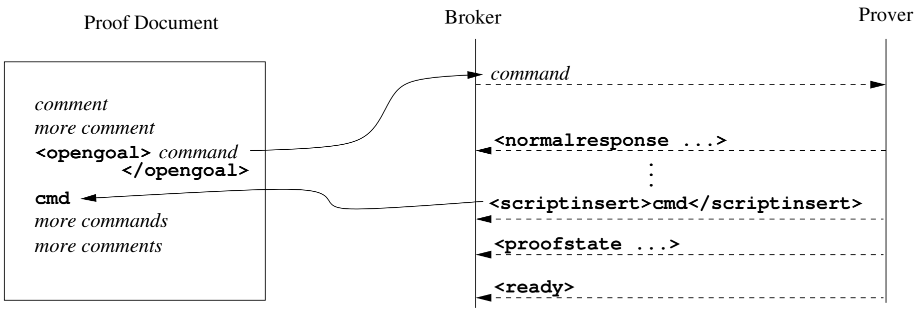


Fig. 4. Backflow

The other new element is <`scriptinsert`>, for script backflow:

scriptinsert = **element** scriptinsert { metavarid_attr?, text }
metavarid_attr = **attribute** metavarid { token }

To see how <`scriptinsert`> works, consider the usual PGIP protocol: after a prover command is sent to the prover, the prover may send a number of prover messages such as <`normalresponse`> or <`proofstate`>, followed by a final <`ready`> message to indicate its availability. A <`scriptinsert`> sent by the prover causes the text of the message to be inserted into the central document at the current point of processing, after being parsed. Fig. 4 illustrates this. The optional `metavarid` attribute specifies an alternative location in the document.

Having described PGIP and its extensions, we next show use cases which demonstrate the extensions at work, to clarify their use and show their viability.

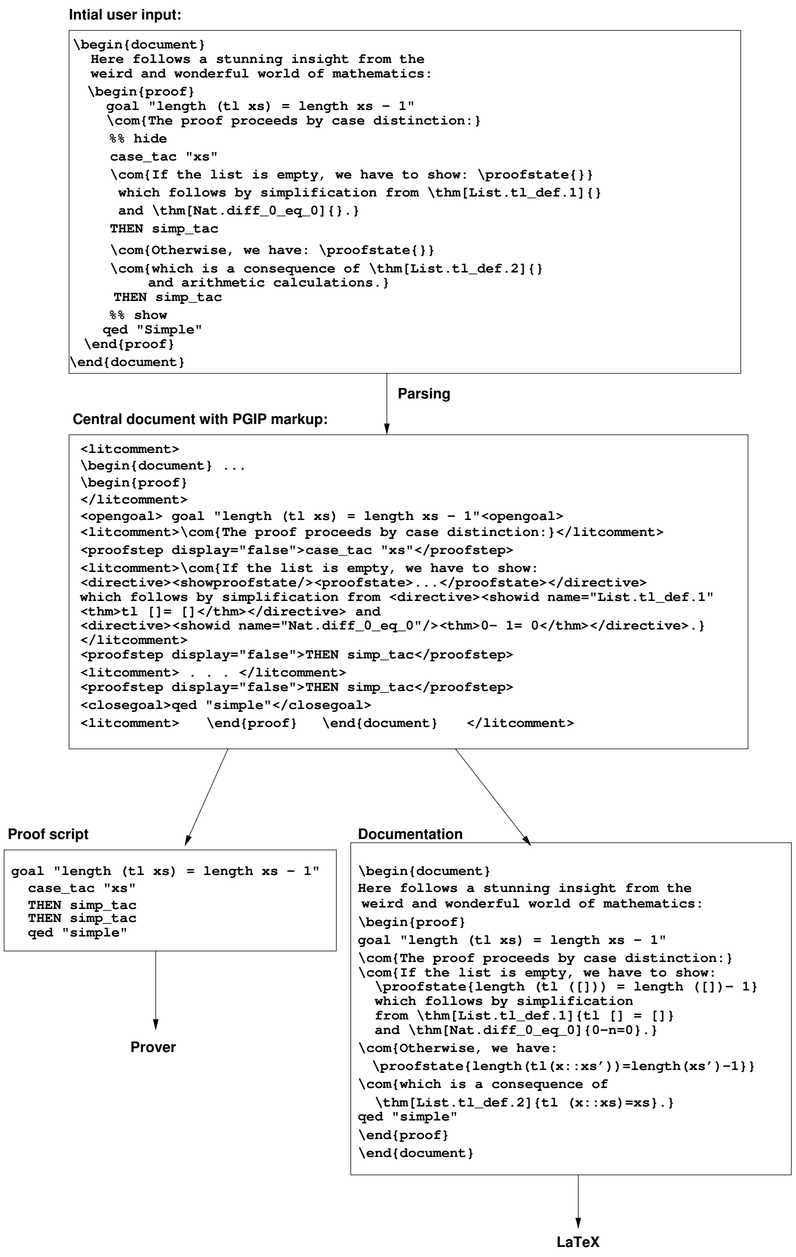
4 Literate Proving Made Easy

This section demonstrates how documentation backflow can provide literate proving facilities in a generic system architecture.

Consider a prover with a simple tactic language used to write the proof in Fig. 3 shown previously. Neither the structure nor the relevant definitions and lemmas are obvious from this script, and, as typical for LCF-style provers, the intermediate proof states of the underlying reasoning are completely implicit.

We are now going to add literate proving facilities to this prover, based on LaTeX. First, proofs are enclosed in a `proof` environment, and `\com` marks literate comments (inside proofs). The content of literate comments is just usual LaTeX code. We also need concrete syntax for the directives, e.g. `\proofstate`. Finally, there are *pragmas* (comments which have a side-effect while processing the script), such as:

```
%% declare_config cname [= expr]
%% hide [cname]
%% show [cname]
```



Intial user input:

```
\begin{document}
  Here follows a stunning insight from the
  weird and wonderful world of mathematics:
 \begin{proof}
    goal "length (tl xs) = length xs - 1"
    \com{The proof proceeds by case distinction:}
    %% hide
    case_tac "xs"
    \com{If the list is empty, we have to show: \proofstate{}}
     which follows by simplification from \thm[List.tl_def.1]{}
     and \thm[Nat.diff_0_eq_0]{}.}
    THEN simp_tac
    \com{Otherwise, we have: \proofstate{}}
    \com{which is a consequence of \thm[List.tl_def.2]{}
         and arithmetic calculations.}
     THEN simp_tac
    %% show
   qed "Simple"
 \end{proof}
\end{document}
```

Central document with PGIP markup:

```
<litcomment>
\begin{document} ...
\begin{proof}
</litcomment>
<opengoal> goal "length (tl xs) = length xs - 1"<opengoal>
<litcomment>\com{The proof proceeds by case distinction:}</litcomment>
<proofstep display="false">case_tac "xs"</proofstep>
<litcomment>\com{If the list is empty, we have to show:
<directive><showproofstate/><proofstate>...</proofstate></directive>
which follows by simplification from <directive><showid name="List.tl_def.1"
<thm>tl []= []</thm></directive> and
<directive><showid name="Nat.diff_0_eq_0"/><thm>0- 1= 0</thm></directive>.}
</litcomment>
<proofstep display="false">THEN simp_tac</proofstep>
<litcomment> . . . </litcomment>
<proofstep display="false">THEN simp_tac</proofstep>
<closegoal>qed "simple"</closegoal>
<litcomment>   \end{proof}   \end{document}   </litcomment>
```

Proof script

```
goal "length (tl xs) = length xs - 1"
  case_tac "xs"
  THEN simp_tac
  THEN simp_tac
  qed "simple"
```

Documentation

```
\begin{document}
Here follows a stunning insight from the
 weird and wonderful world of mathematics:
\begin{proof}
goal "length (tl xs) = length xs - 1"
\com{The proof proceeds by case distinction:}
\com{If the list is empty, we have to show:
  \proofstate{length (tl ([])) = length ([])- 1}
  which follows by simplification
  from \thm[List.tl_def.1]{tl [] = []}
  and \thm[Nat.diff_0_eq_0]{0-n=0}.}
\com{Otherwise, we have:
  \proofstate{length(tl(x::xs'))=length(xs')-1}}
\com{which is a consequence of
  \thm[List.tl_def.2]{tl (x::xs)=xs}.}
qed "simple"
\end{proof}
\end{document}
```

Fig. 5. Literate proving

These can be used to set or reset the `nodisplay` attribute. The `hide` and `show` pragmas may optionally have a *configuration name* like `short` or `detailed` that allows for the generation of different versions of a proof document during rendering. Configuration names may be expressed in terms of other previously declared configuration names, e.g.

```
    %% declare_config both = short or detailed
```

The resulting LaTeX document and the concrete document flow is shown in Fig. 5. As we can see, the parsing adds the necessary markup to the central document. The proof is run in the broker by just stepping through it, skipping over comments, and filling in the proofstate or the references to lemmas in the literate comment. After that, the second literate comment in the proof reads (we have elided the actual proofstate and displayed theorems):

```
<litcomment>\com{If the list is empty, we have to show:
<directive><showproofstate/><proofstate>...</proofstate></directive>
which follows by simplification from
<directive><showid name="List.tl_def.1"/><term>...</term></directive> and
<directive><showid name="Nat.diff_0_eq_0"/><term>...</term></directive>.
```

Notice that the proofstate (the elided part) is encapsulated by the <`proofstate`> element, such that if we rerun the script, it will be replaced by the then current proofstate. The same holds for lemmas or theorems like <`thm`>. From this document, we can extract both a proof script and LaTeX documentation easily.

Our approach is quite generic: we just need to integrate the parser for some concrete syntax; in principle, it should be possible to generate the XML-formats used by OpenOffice, for example. Of course, this kind of literate programming is enhanced if the prover can generated typeset output to embed in the document.

5 Script Backflow

Here is our sample proof again, this time in Isabelle/Isar, which makes the structure of the case distinction quite explicit:

```
lemma "length (tl xs) = length xs − 1":
proof (cases xs)
  case Nil thus ?thesis by simp
  txt{* If the list is empty, we have to show: @{proofstate}
      which follows from simplification with @{thm Nat.diff_0_eq_0}
      and @{thm{List.tl_def.1}.*}
next
  case (Cons y ys) thus ?thesis by simp txt{* ... *}
qed
```

However, the text is much longer than the tactical proof above, and even though this verbosity is exactly what makes Isar proofs easier to read and more stable to maintain, proofs become laborious to type at the outset. The only input inherently *required* of the user (after starting the proof) is the decision to perform a case distinction (`cases`), and the name of the variable "`xs`". Given this, the prover can choose the right case distinction rule according to the type of the variable; this results in the patterns, their local variables, etc., which are transferred to the PG broker as a template via backflow:

```
proof (cases xs)
      case Nil thus ?thesis <proof>
  next
      case (Cons y ys) thus ?thesis <proof>
done
```

Afterwards, the user can continue to fill in the two actual proofs (for the placeholder `<proof>`) in the case branches. For the subproof shown, the prover can also yield the list of lemmas used in the simplifications included in the template:

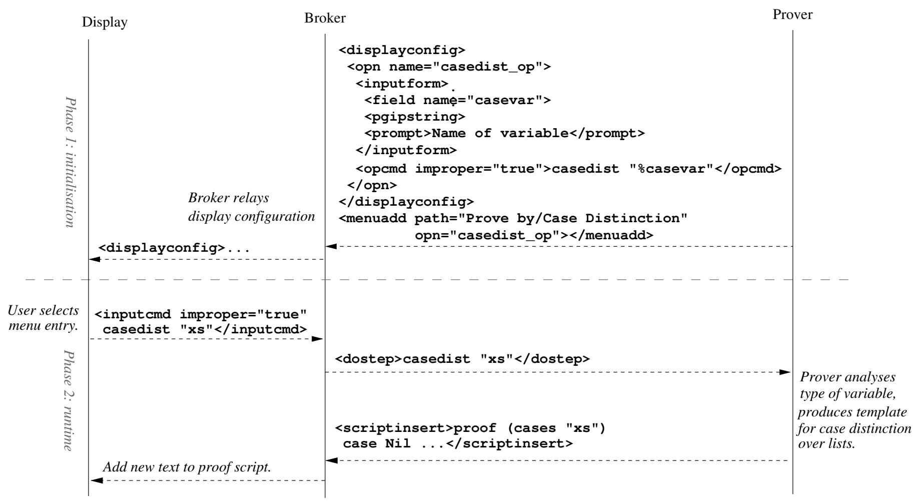
```
txt{* which follows from simplification with @{thm Nat.diff_0_eq_0}
     and @{thm{List.tl_def.1}. *}
```

(the prover could even be more clever and try to fill in obvious proofs by checking whether simplification or other automatic proof patterns would succeed, and documenting appropriately). In a graphical interface, the proof template above would be generated by a mouse-click for the selection of the proof method `cases` and two keystrokes to type `xs`. These three user actions replace the tedious typing of the complete text. Other proof methods such as proof by induction could be generated using the same backflow mechanisms.

The protocol for script backflow. In detail, the above interaction between prover, broker and display proceeds as follows. Suppose a user event such as a menu select, mouse click or drag-and-drop has occurred. The prover operation (as described in Sect. 3.4) triggered by this event causes the display to send a prover command, which is marked as proper or improper, to the broker. If it is a proper prover command, the broker inserts it into the proof script; if it is an improper prover command, it is transferred to the prover directly.

We need the prover to configure the displays to bind events to prover operations; this is done once, in the initial startup phase (operation configurations in PGIP are intended for displays). Then whenever the event occurs, the display evaluates the operation. This may require more input from the user, e.g. the variable name in the case distinction example above; to this end, operations can have *inputforms* configured which describe this additional input, e.g. here a one-line string input with prompt "Name of variable". Finally, the prover evaluates the command, which results in a backflow to the broker.

Fig. 6 shows the resulting flow of messages, slightly abridged. In the configuration phase, the prover sets up an operation `casedist_op`, which requires a string (the name of the variable) as user input. The operation is bound to a menu entry *Case distinction* in the menu *Prove by* (there will be other menus and sub-menus). When users activate that menu, the operation is executed and they will be asked to input the variable name, e.g. by typing in an input form or by pointing to it in the proofstate display. The placeholder `%casevar` in the operation command is replaced with the input, and the command is sent to the broker. According to the configuration (the attribute `improper`), it is marked


Display
Broker
Prover
Phase 1: initialisation
Broker relays display configuration
User selects menu entry.
Phase 2: runtime
Prover analyses type of variable, produces template for case distinction over lists.
Add new text to proof script.

```
<displayconfig>
 <opn name="casedist_op">
  <inputform>
   <field name="casevar">
   <pgipstring>
   <prompt>Name of variable</prompt>
  </inputform>
  <opcmd improper="true">casedist "%casevar"</opcmd>
 </opn>
</displayconfig>
<menuadd path="Prove by/Case Distinction"
         opn="casedist_op"></menuadd>
```

```
<displayconfig>...
```

```
<inputcmd improper="true"
 casedist "xs"</inputcmd>
```

```
<dostep>casedist "xs"</dostep>
```

```
<scriptinsert>proof (cases "xs")
 case Nil ...</scriptinsert>
```

Fig. 6. Message exchange for backflow in the case distinction situation

as an improper command, so the broker relays the command to the prover. The prover analyses the type of the variable, decides case distinction on lists is appropriate, with one case for the empty and non-empty lists each, and generates the corresponding backflow. The generated proof template text is inserted into the proof script.

PGIP has a generic display model, where simpler displays (e.g. text-based ones) are free to ignore configurations which only make sense for graphical displays. The only adjustment needed here over the description in [3] is to allow improper commands as well as proper ones. Note how only the prover needs knowledge about the logical structure of the proof, the types involved and so on; from the broker's and displays' point of view, the protocol is completely generic.

5.1 Calculational Proof

Calculational proof is probably the most well-known proof presentation paradigm as it is taught in school mathematics. Here is an example in Isar (which uses Isabelle's axiomatic type class mechanism to restrict instances of the type variable `'a` to those satisfying the group axioms):

```
theorem group_right_one: "x * one = (x::'a::group)"
proof -
  have "x * one = x * (inverse x * x)"
    by (simp only: group_left_inverse)
  also have "... = x * inverse x * x"
    by (simp only: group_assoc)
  also have "... = one * x"
    by (simp only: group_right_inverse)
  also have "... = x"
    by (simp only: group_left_one)
  finally show ?thesis .
qed
```

This is already quite readable, and the generation of proof presentations that abstract the proof technical details fully or up to the names of the used lemmas in each step are straightforward. As an Isar proof text, this proof pattern requires the user to type all the intermediate proof stages; they may be abbreviated by meta-variables `?X1`,..., `?Xn`, but it is still cumbersome. GUI-supported backflow helps here substantially: the user states only the overall goal, selects *calculational proof*, and sets a *focus* on a subterm (e.g., `x * inverse x`) serving as the redex of a theorem, and a theorem (e.g., `group_left_inverse`).

In this scenario, the construction of backflow is quite complex and requires the development of specialised tactic support. The main problem is to generate proof scripts that are as *general* and *reusable* as possible, ideally avoiding positional referencing by using general methods such as

```
by (simp only: group_assoc)
```

instead of a left-to-right one-step application such as

```
by (rule_tac P=% x. x * (inverse x * x) = x in subst[OF group_assoc.assoc]).
```

The technique of *proof abstraction* is based on generate-and-test heuristics for successful proof attempts with the fall-back of the least general proof method.

5.2 Window Inferencing

Logically, calculational proof depends on the transitivity of equality which allows us to string together a sequence of lemmas the form $t_i = t_{i+1}$ for $i = 1, \ldots, n$ to one theorem $t_1 = t_{n+1}$. Window inferencing [12] is a generalisation of calculational proof where instead of an equation we have a non-disjoint *family* of binary relations. Window inferencing also allows us to apply rules to subterms of the current proof state; this is referred to as *opening a window* on that subterm, and it may produce additional assumptions (e.g. opening a window on the positive branch of a conditional adds the condition as an assumption). When closing a window, implicit monotonicity reasoning is executed to validate replacing a focus with the result of the sub-derivation in a window at the next higher level.

Previous work has shown how window inferencing can be implemented as a tactic in Isabelle, using a dedicated GUI for window inferencing [19]. Here, we can achieve the same thing using *annotated terms* in PGML and backflow in PGIP. The special input field `%selected` can be used in operations to denote the selected subterm (on displays that do not support subterm selection, these operations will be ignored). The operation to open a window then sends the command `open_win %selected`, which causes the command `open_win` p to be sent to prover (via the broker, as in Fig. 6), and the prover constructs the relevant subterm and context from p. The path p is in the prover's internal abstract syntax representation of the term, it only makes sense to the prover and needs to be post-processed to render a PGML string. Again, with only modest support from the prover, we can add a very useful high-level feature for assisting document authoring.

6 Conclusions

We have presented a new component-based system architecture for authoring mathematical documents together with formal proofs. It extends the generic PG Kit infrastructure for interactive proof. The novel concept in extending proof script editing to authoring is the support of component backflow on the protocol level PGIP as well as in its implementation in the broker.

The implementation of our design is ongoing. The broker architecture, with an Emacs-based and an Eclipse-based display, has been developed and is available as a prototype [3, 4]. The authoring extensions have been added to this prototype and support from provers (in particular, Isabelle) is anticipated in future development versions.

6.1 Related Work

The basic idea of the document-centred approach can be traced back to Knuth's work on literate programming [16]. In the context of formal proof and formalisation of mathematics, the field can be divided into two fundamentally different approaches: one tries to make formal proofs more human-readable, or one tries to make textbook-proofs more formal or at least intuit their underlying formal structure. In the former line of research stands Automath [9], Mizar [23], and its descendants like Isabelle/Isar [25] or Coq's integrated documentation facility `coqdoc` that can extract a document offline in various formats. Théry's approach [22] bridges the gap by defining an XML format for manually annotating statements in mathematical papers to link them to formal counterparts, wherein proofs must be supplied; consistency is checked in a prover. Similar approaches include Weak Type Theory [15], MathLang [14]), or the Dialog project [5]. In a sense in the opposite direction, Kohlhase [17] works on the existing mathematical corpus of LaTeX papers and tries to capture their semantic content automatically with additional markup.

Although we take formal proof as the starting point, our document-centred approach eases the task of reconstructing a human-readable format during formal proof development, using the information available via backflow from the presentation of terms and proofstates, or the information from certain automated proof strategies or advanced techniques like proof planning [10]. Of course, the resulting annotations are merely organised text, kept consistent by using references to theorems, etc., which are resolved late in the presentation process. Integrating with the complementary approach of [15] with respect to these annotations is worth investigating.

6.2 Outlook and Future Work

This paper describes authoring facilities on the document level. An important future direction is to study large and richly connected developments, spanning multiple proof script files and proof modules, and supporting reordering in producing the human-readable documentation. The framework partly addresses this at the moment because there are PGIP elements describing file-level commands and dependencies between prover commands (relying on information from the theorem prover), so to extend the example in Sect. 4 we can add commands like `\openscriptfile{example.thy}` and `\closescriptfile` to indicate destination script files; several files may then be produced on processing.

Another interesting use case for our architecture would be to have the prover insert proof objects into the document via backflow. Here, a proof object would just be formal object which can be reconstructed by the prover on demand to show the validity of the proof. This would allow a proof to be more or less completely informal except for the embedded proof objects, which could be used to validate the formal content.

Finally, we want to conduct usability studies to substantiate the claim that assisted authoring increases productivity compared to unassisted editing. A good evaluation methodology would be to investigate usability for mid-sized proofs using well-known HCI techniques (e.g., keystroke-measures), as well as to collect subjective experience reports from larger proof authoring projects.

Acknowledgements. We are grateful to our collaborators on the Proof General Kit project and the developers of Isabelle for their support of our experiments in interface development. For this paper, several useful suggestions for improvement were made by the MKM'05 referees.

References

1. J.-R. Abrial and D. Cansell. Click'n'prove: Interactive proofs within set theory. In D. Basin and B. Wolff, editors, *TPHOLs 2003*, LNCS 2758, p. 1–24. Springer Verlag, 2003.
2. D. Aspinall and C. Lüth. Commentary on PGIP. Available from `http://proofgeneral.inf.ed.ac.uk/kit/`, September 2003.
3. D. Aspinall and C. Lüth. Proof General meets IsaWin. In D. Aspinall and C. Lüth, editors, *User Interfaces for Theorem Provers UITP'03. ENTCS* 103:C, 2003.
4. D. Aspinall and C. Lüth. Parsing, editing, proving: The PGIP display protocol. In *User Interfaces for Theorem Provers UITP'05*, Apr. 2005.
5. S. Autexier, C. Benzmüller, A. Fiedler, H. Horacek, and Q. Bao Vo. Assertion level proof representation with underspecification. In F. Kamareddine, editor, *Proc. MKM Symposium MKM'2003*, Edinburgh, Nov. 2003.
6. J. Avigad. Notes on a formalization of the prime number theorem. Technical report, Carnegie Mellon, 2004.
7. Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In M. Hagiya and J. C. Mitchell, editors, *Proce. of the International Symposium on Theoretical Aspects of Computer Software*, LNCS 789, p. 141–160. Springer Verlag, 1994.

8. A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.
9. N. G. de Bruijn. A survey of project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, p. 589– 606. Academic Press, 1980.
10. L. Dixon and J. Fleuriot. A proof-centric approach to mathematical assistants. *Journal of Applied Logic: Special Issue on Mathematics Assistance Systems*, 2005. To appear.
11. G. Gonthier. A computer-checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge, 2004. `http://research.microsoft.microsoft.com/ gonthier/4colproof.pdf`.
12. J. Grundy. Transformational hierarchical reasoning. *Computer Journal*, 39:291–302, 1996.
13. T. C. Hales. The Flyspeck project page. `http://www.math.pitt.edu/ thales/flyspeck/index.html`.
14. F. Kamareddine, M. Maarek, and J. B. Wells. Flexible encoding of mathematics on the computer. In A. t. Asperti, editor, *Mathematical Knowledge Management MKM 2004*, LNCS 3119, p. 160– 174. Springer Verlag, 2004.
15. F. Kamareddine and R. Nederpelt. A refinement of deBruijn's formal language of mathematics. *Journal of Logic, Language and Information*, 13(3):287– 340, 2004.
16. D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
17. M. Kohlhase. Semantic markup for TEX/LATEX. In *Informal Proc. Mathematical User Interfaces, Math UI '04*, 2004.
18. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167– 189, 1999.
19. C. Lüth and B. Wolff. TAS — a generic window inference system. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, LNCS 1869, p. 405–422. Springer Verlag, 2000.
20. D. Mackenzie. What in the name of Euclid is going on here? *Science*, 307:1402–1403, 2005.
21. RELAX NG XML schema language, 2003. Home page at `http://www.relaxng.org/`.
22. L. Théry. Formal proof authoring: An experiment. In *Informal Proc. User Interfaces for Theorem Provers, UITP '03*, 2003.
23. A. Trybulec et al. The Mizar project, 1973. See web page hosted at `http://mizar.org`, University of Bialystok, Poland.
24. M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99*, LNCS 1690. Springer Verlag, 1999.
25. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2001.
26. D. Winterstein, D. Aspinall, and C. Lüth. Proof General/Eclipse. In *User Interfaces for Theorem Provers UITP'05*, 2005.