Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 103 (2004) 121–138
www.elsevier.com/locate/entcs

Colouring Proofs: A Lightweight Approach to Adding Formal Structure to Proofs

Laurent Théry[1]

Dipartimento di Informatica
Università di L'Aquila, Italy

Abstract

In this paper we propose a proof format to write formal proofs motivated by a formalisation of floating-point numbers. This proof format aims at being adequate for both proof presentation and mechanised proof checking. We also present a simple graphical interface to support this proof format.

Keywords: mechanised proof checking, proof presentation, XML

1 Introduction

The work presented in this paper comes from a long-term collaboration [8] on applying theorem proving technology to mechanically validate new algorithms for floating-point numbers. In the computer arithmetic community researchers often publish detailed proofs of the correctness of their new algorithms. Most of the time these proofs are intricate and require a great deal of tedious checking for a skeptic reader. An example of such a proof is the 19 page proof given in [10]. Pioneering works [14,19,22] have shown how effectively theorem proving systems can be used to check proofs. In our case, proofs are encoded and formally verified in the Coq prover [15]. In systems like Coq, proofs are represented by proof scripts. A script is composed of a set of elementary commands called tactics. Tactics guide the prover in the search of the formal proof. In that respect, proof scripts are very different from the usual proofs on paper.

[1] Email: Laurent.Thery@di.univaq.it

doi:10.1016/j.entcs.2004.07.002

Our modus operandi to mechanically check a proof within Coq has nearly remained unchanged since the beginning. The proof is first written carefully on paper with as many details as possible. Then, this document is used as a guide to produce the script for the proof system. At this stage, the proof on paper and the script usually take different roads. The proof on paper is reworked, mainly shortened, so to be inserted into some technical report. The script is also modified. Good practice recommends to reorganize it in order to increase reusability and robustness.

The fact that there is no direct connection between the published proof and its proof script has at least three unpleasant consequences. First of all, the published proof cannot be claimed to have been mechanically checked. What has been checked is that the conclusion of the proof holds. For example, nothing ensures that the published proof does not contain errors. Fundamental errors are most likely to have been discovered during the checking process. Still, having just a tiny error in a published proof is very annoying, in particular when the main claim of the paper is that all the results have been mechanically checked.

The second consequence is that published proofs are of limited interest for maintaining the formal development. As provers are constantly evolving, keeping the scripts up to date is a real problem. The published proof tells very little on how the corresponding proof script is structured. Being able to quickly find out what a specific subpart of the script is supposed to do is what is really needed when maintaining scripts.

The third consequence that is closely related to the second one is a lack of flexibility. It is often the case that, when something has been proved under particular assumptions, one would like to see the effect of slightly modifying them. Typically in floating-point arithmetic, one could try to change the base of the arithmetic or the rounding mode. Doing such experiments with a mechanised proof is very simple. One simply needs to change the assumptions and rerun the scripts. The fact that a proof is not valid anymore is detected when the application of a tactic fails. Finding out the corresponding step in the proof on paper is not immediate and requires some non-trivial expertise.

In this paper we present a proof format which enforces a direct link between the proof on paper and its mechanised version. The paper is structured as follows. The format is introduced in Section 2. An example of a proof in our format is given in Section 3. In Section 4 a graphical interface to build proofs in our format is presented. Finally we relate our format with other approaches in Section 5 before concluding in Section 6.

2 The Proof Format

In order to find an adequate proof format that would reduce the gap between the structure of proofs on paper and the structure of proof scripts, we had three sources of inspiration. The first one is the paper [17] by Leslie Lamport that describes how formal proofs should be written. Reading that paper, it is clear that the main feature of formal proofs is the *structure*. A formal proof should explicitly expose how the initial problem is decomposed into elementary subproblems, until obvious statements are obtained. One of the main requirements of a formal proof format is then to highlight the proof structure.

The second source of inspiration comes from the reading of different proofs in arithmetic. Because they are mostly dealing with transforming inequalities, the proofs can often be understood by just reading the formulae and forgetting the text around. In our format, we push this observation to the extreme and take as an assumption that only formulae matter.

The last source of inspiration comes from verification condition generators like Why [11]. These systems are used to prove the correctness of programs. They take as an input a program annotated with logical assertions and output a list of conditions. Proving all these conditions ensures that the annotations in the program are valid. We would like to have a similar mechanism for proofs on paper. Running the tool on a proof should generate a list of formal conditions that needs to be checked in order to ensure that what is written in the proof is correct.

The definition of the actual proof format derives directly from these considerations. An appropriate representation of proofs to highlight their structure is known since decades. It is the *natural deduction style* proposed by Prawitz [21]. In the following we use some of the rules of natural deduction to illustrate how our format works. For the actual description of the format we use a mark-up language à la XML [4].

2.1 The format for formulae

In our format, formulae are expressed in a typed first-order logic. The syntax is directly inspired by the language to write logical assertions in Why. The language contains the usual logical connectors: conjunction `and`, disjunction `or`, implication `->`, negation `not`. For example, the formula

$$\neg(x \vee y) \Rightarrow \neg x \wedge \neg y$$

is written in our format as

```
not(x or y) -> not(x) and not(y)
```

The precedence for our logical connectors is the usual one. Arbitrary n-ary predicates are represented by

predicateName(*term*$_1$,...,*term*$_n$)

Equalities, inequalities and interval predicates have their usual infix syntax. For example, a basic property of interval predicates is written as

```
(x <= y <= z and x <> y and y <> z)  -> x < y < z
```

A formula can be universally or existentially quantified using the keywords `forall` and `exists` respectively. The variable bound by the quantification is required to be typed. For example, the formula

```
forall x: int. exists y: int. x < y
```

indicates that there are infinitely many integer numbers.

Terms are variables, numerical constants and arbitrary n-ary function applications with the syntax

functionName(*term*$_1$,...,*term*$_n$)

The infix syntax is also available for the common arithmetic operations `+`, `-`, `*`, `/`, `%`. For example, the formula

```
forall x, y, z, n:  nat.
       exp(x, n) = exp(y, n) + exp(z, n) ->
          n <= 2 or x * y * z = 0
```

is a famous ex-conjecture.

Two aspects of our format for formulae are worth commenting. First, we could have chosen the Coq syntax for formulae but have decided to use a language that is independent of a particular prover. We believe that this independence is a key issue for maintaining large formal developments on the long run. Second, we have chosen a textual representation. A natural alternative would be to use MathML [5]. As a matter of fact, an earlier version of our proof format [26] did use a light version of MathML. For example, the formula

$$\neg(x \vee y) \Rightarrow \neg x \wedge \neg y$$

was represented as

```
<imp>
    <neg>
        <or> <v><n>x</n></v> <v><n>y</n></v> </or>
    </neg>
    <and>
        <not> <v><n>x</n></v> </not>
        <not> <v><n>y</n></v> </not>
    </and>
```

```
</imp>
```

The unfortunate consequence of that choice was that users could not be asked to type formulae directly anymore. Some support should be provided. As we could not find any satisfactory and freely available component to edit MathML, we decided to step back and represent our formulae with the more concise textual representation.

2.2 The format for proofs

In our format, we use tags to describe the proof structure. Proofs and subproofs are marked with the tag `p`. Proofs can be given a name with the tag `n`. The conclusion is denoted by the tag `c`. For example, the introduction rule for the conjunction in natural deduction

$$\frac{A \qquad B}{A \wedge B}$$

is represented in our format as

```
<p>
    <n>and Intro</n>
    <p>
        <c>A</c>
    </p>
    <p>
        <c>B</c>
    </p>
    <c>A and B</c>
</p>
```

Indentation is used here for readability only. No specific layout is imposed by the format. Also the relative positions of the subgoals and the conclusion are free. Putting the conclusion first gives a goal-directed flavour to the proof. What is proved is given before explaining why it holds. Putting the conclusion last gives the more usual forward style. Most of the time a proof on paper is carried out using the forward style, but for key steps like the application of an inductive principle the conclusion may be given first. Our format easily accommodates both styles.

Assumptions are represented by the tag `h`. The tag contains the formula that is assumed. This is illustrated with the rule for case analysis

$$\frac{A \vee B \qquad \begin{array}{c}[A]\\ \cdots \\ C\end{array} \qquad \begin{array}{c}[B]\\ \cdots \\ C\end{array}}{C}$$

and its encoding

```
<p>
    <n>or Elim</n>
    <p>
        <c>A or B</c>
    </p>
    <p>
        <h>A</h>
        <c>C</c>
    </p>
    <p>
        <h>B</h>
        <c>C</c>
    </p>
    <c>C</c>
</p>
```

No limit is put on the number of assumptions and subproofs a proof can hold.

Local variables are represented with the tag `v`. As our language is typed, the text enclosed by the tag must have the form *name*: *type*. We illustrate the use of this tag with the universal introduction rule

$$\frac{P(n)}{\forall x.\, P(x)}$$

In our language x must be typed. If we consider predicates over integers, the representation of the introduction rule is the following:

```
<p>
    <n>forall Intro</n>
    <p>
        <v>n:int</v>
        <c>P(n)</c>
    </p>
    <c>forall x: int. P(x)</c>
</p>
```

A proof can have as many local variables as needed.

Two extra mechanisms have also been added. The first one lets the user name the hypothesis. This is already possible by combining the tags `h` and `n`, for example

```
<h>A <n>H1</n> </h>
```

However, putting the name inside the text of the assumption is not a good idea. It gives a very limited naming schema, no extra text can be added. To get a more general one, an extra tag `f` is used to indicate the value of the assumption. Named assumptions should then be written as

```
<h> <f>A</f> <n>H1</n> </h>
```

The second mechanism makes it possible to explain why the conclusion of a proof holds, i.e. to give a justification. This is done with an extra tag `j`. Justifications hold names. For example,

```
<p> <j> <n>exp_plus</n> </j> <c>C</c> </p>
```

indicates that the conclusion is a consequence of the theorem `exp_plus`. Names in a justification can either refer to other proofs or to local assumptions.

To sum up, our proof format is composed of seven tags: `p` for proofs, `c` for conclusions, `h` for hypotheses, `v` for variables, `n` for names, `f` for formulae, `j` for justifications. A proof in our format is structured as follows:

- There is a top tag `p`.
- Each tag `p` in the proof has exactly one subtag `c`, at most one subtag `n`, at most one subtag `j` and some (possibly zero) subtags `v`, `h` and `p`.
- Each tag `h` in the proof has at most one subtag `f` and at most one tag `n`.
- Each tag `j` in the proof has only subtags `n`.

2.3 Generating proof obligations

In program verification, in order to generate the conditions one needs to use elaborate techniques such as computing the weakest preconditions. For our format, the generation is much simpler. The algorithm is illustrated here for Coq but could be easily adapted to other systems. It consists in a traversal of the tagged structure from top to bottom and from left to right. Each time a proof tag is encountered, its subproofs are first recursively processed before generating the condition associated with the tag.

Each condition is represented by a lemma. A piece of script is also added to do the necessary book-keeping and get the appropriate assumptions. To give a more concrete example, consider the following proof

```
<p>
    <h> <f>A</f> <n>h1</n> </h>
    <p>
        <n>p1</n>
        <c>B</c>
    </p>
    <p>
        <n>p2</n>
        <h> <f>C</f> <n>h2</n></h>
        <c>D</c>
    </p>
    ...
</p>
```

The first condition that is generated corresponds to `p1` and is represented by the lemma `c_p1`. The condition has the following form:

```
Lemma c_p1:  A -> B.
Proof.
Intros h1.
Apply ok.
```

```
Qed.
```

The subproof `p1` has been declared in a context where the assumption A is visible. Its conclusion is B. So the actual statement it proves is $A \Rightarrow B$. The piece of script that is added

```
Intros h1.
Apply ok.
```

first introduces assumptions. In Coq, assumptions can be named. The name given in the document can then be faithfully reflected inside Coq. The tactic `Intros h1` introduces the first assumption A with the name `h1`. The final tactic `Apply ok` is generated so that the proof is always accepted by the prover [2]. This simple trick gives typechecking for free: any error in the initial document is automatically detected by the prover when processing the generated file. As each lemma represents a single step in the proof, tracking the origin of an error in the initial document is easy. The actual task of formally checking the proof consists in replacing all the applications of the axiom `ok` with appropriate tactics.

For the second subproof `p2`, the condition has the following form:

```
Lemma c_p2: A -> C -> D.
Proof.
Intros h1.
Generalize (c_p1 h1); Intros p1.
Intros h2.
Apply ok.
Qed.
```

There are two assumptions in the statement of the lemma: A comes from the context and C is a local assumption of `p2`. For the proof script, the tactic `Intros h1` introduces the first assumption A of the lemma with the name `h1`.

The second tactic is more elaborate. It represents the fact that `p1` is visible from `p2`. In order to introduce `p1`, the lemma `c_p1` first needs to be instantiated with the context that is common between `p1` and `p2`, i.e. the assumption A whose name is `h1`. The expression `(c_p1 h1)` does exactly that and corresponds to a proof of B. Before the application of the tactic `Generalize (c_p1 h1)`, the goal is $C \Rightarrow D$, after it is $B \Rightarrow C \Rightarrow D$. The next tactic `Intros p1` introduces B with the name `p1`. The result of these two tactics is the expected one: the fact that B is true can be used with the name `p1` inside the proof `p2`.

[2] This tactic represents the application of a predefined axiom `ok` that states that everything is true.

The third tactic simply introduces C with the name `h2` in the assumption list. Note that the generator takes a special care in introducing assumptions in the proper order. The assumptions introduced last are the ones displayed next to the conclusion. In our example, the closest assumption is `h2`, then `p1` and then `h1`.

Finally, when all introductions are done, the order of the assumptions is possibly reorganized to take into account the references given by the justifications. This improves the performance of automatic tactics that usually privilege the last introduced assumptions.

In the generation, the main degree of freedom concerns the visibility of the different subproofs. In natural deduction, if a proof P has n subproofs p_1, p_2, ... p_n, these subproofs are considered independent. For example, one cannot use the fact that the conclusion of p_1 is true in the proof p_2 without copying the whole proof. In a textual proof, the proof p_1 is read before the proof p_2. It seems then more natural to have a less restrictive policy. For this reason, the visibility rule we have implemented in our generation is that the conclusion of p_i is visible inside p_j for $i < j$ and invisible outside P. A similar approach has been adopted by Richard Bornat [3] for the box style proposed by Fitch in [12].

3 An Example

So far we have presented our format and have shown how the conditions are generated. We still need to show how proofs on paper can be translated into our format. The idea behind our format is that this can be done mostly by just adding tags to the text of the proof. We illustrate this with a proof of an elementary property of the exponential function over integers:

```
Property exp_pos:
Let x and y be two integers,
if 0 < x, to prove exp(x,y) > 0 we have three cases:
        If 0 < y then exp(x,y) >= x
        If 0 = y then exp(x,y) = 1
        If 0 > y then exp(x,y) = 1/exp(x,-y)
```

We are going to progressively integrate the tags into the text. The easiest tag to add is the one for names. Here in the proof there is only one name `exp_pos`:

```
Property <n>exp_pos</n>:
Let x and y be two integers,
if 0 < x, to prove exp(x,y) > 0 we have three cases:
        If 0 < y then exp(x,y) >= x
        If 0 = y then exp(x,y) = 1
        If 0 > y then exp(x,y) = 1/exp(x,-y)
```

The second step is to tag all local variables, here there are two variables:

```
Property <n>exp_pos</n>:
Let <v>x</v> and <v>y</v> be two integers,
```

```
if 0 < x, to prove exp(x,y) > 0 we have three cases:
        If 0 < y then exp(x,y) >= x
        If 0 = y then exp(x,y) = 1
        If 0 > y then exp(x,y) = 1/exp(x,-y)
```

The next step is to take each formula in the proof and determine if it is a hypothesis or a conclusion. In our example, we have four assumptions and four conclusions:

```
Property <n>exp_pos</n>:
Let <v>x</v> and <v>y</v> be two integers,
if <h>0 < x</h>, to prove <c>exp(x,y) > 0</c> we have three cases:
        If <h>0 < y</h> then <c>exp(x,y) >= x</c>
        If <h>0 = y</h> then <c>exp(x,y) = 1</c>
        If <h>0 > y</h> then <c>exp(x,y) = 1/exp(x,-y)</c>
```

The last step is the most delicate one. It consists in determining the scope of each conclusion. In our example, there is a proof with three subproofs.

```
<p>Property <n>exp_pos</n>:
Let <v>x</v> and <v>y</v> be two integers,
if <h>0 < x</h>, to prove <c>exp(x,y) > 0</c> we have three cases:
        <p>If <h>0 < y</h> then <c>exp(x,y) >= x</c></p>
        <p>If <h>0 = y</h> then <c>exp(x,y) = 1</c></p>
        <p>If <h>0 > y</h> then <c>exp(x,y) = 1/exp(x,-y)</c></p>
</p>
```

We are almost done. In our language variables must be typed. The type information for variables should in fact be put when adding the tags `v`. So the final version of the proof is:

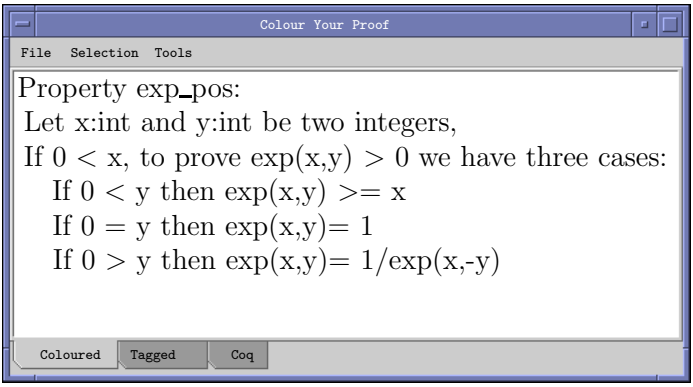
```
<p>Property <n>exp_pos</n>:
Let <v>x: int</v> and <v>y: int</v> be two integers,
if <h>0 < x</h>, to prove <c>exp(x,y) > 0</c> we have three cases:
        <p>If <h>0 < y</h> then <c>exp(x,y) >= x</c></p>
        <p>If <h>0 = y</h> then <c>exp(x,y) = 1</c></p>
        <p>If <h>0 > y</h> then <c>exp(x,y) = 1/exp(x,-y)</c></p>
</p>
```

The generated conditions are given in Appendix A. Note that since subproofs and assumptions were not named, the system has automatically generated ad-hoc names.

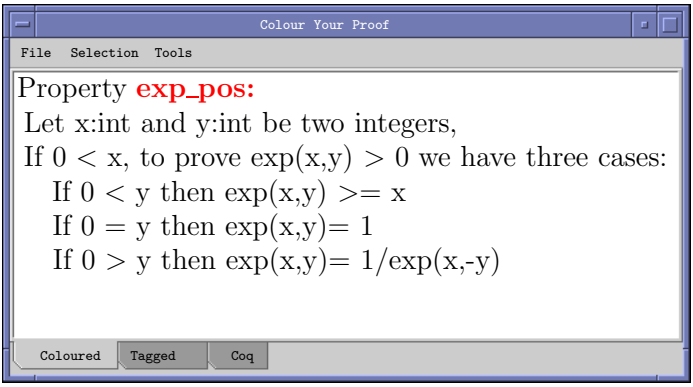
4 The Cyp Graphical Interface

A graphical interface has been developed to build proofs in our format [3]. The interface is composed of a single window. Any text can be read in this window. For example, reading the initial tag-free proof text of Section 3 gives:

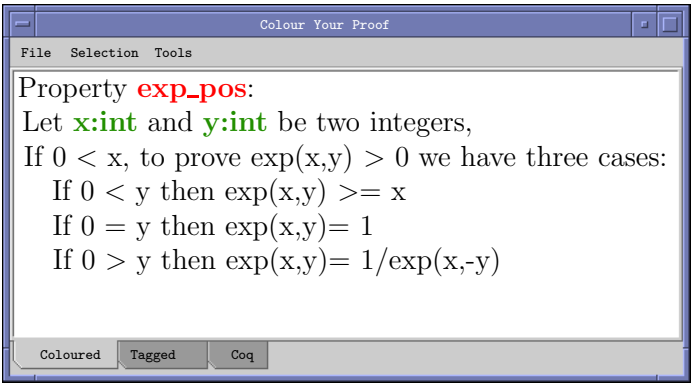
[3] The interface is freely available at ftp://ftp-sop.inria.fr/lemme/cyp/index.html.



To set a tag, the user simply needs to select the region with the mouse and press the corresponding item in the menu *Selection*. For example, selecting the text "exp_pos" and pressing on the item *n* gives the following result:

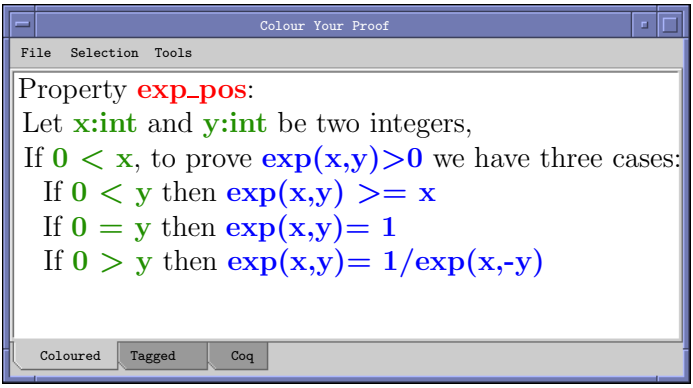


Colours are used to indicate the presence of a tag, here red is used for names. Following the same steps as in Section 3, we then need to select each variable and press on the item v:

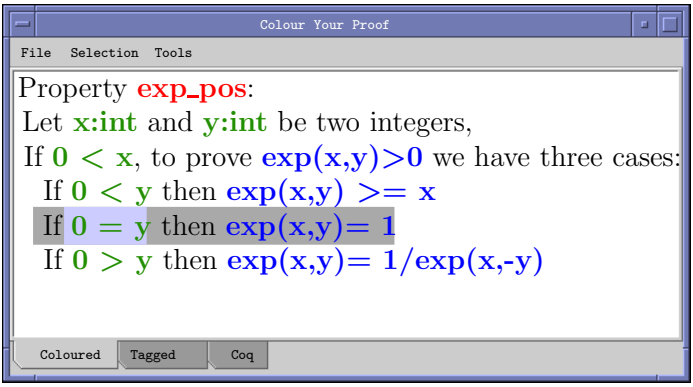


Once again the modification is indicated by a change of colour. This gives a nice metaphor to what it means to turn a proof in our format: it just consists in colouring the proof.

If we continue the process, the complete coloured proof is the following:



Conclusions are coloured in blue and assumptions in green. In order to keep the number of colours as small as possible, we have decided to use the same colour for variables and assumptions. Like this, we have just three colours: red for names, green for what is assumed and blue for what is proved. Note also that proof tags are not visible in the picture above. This is because proofs can be nested. So colouring a proof would automatically hide the colour of its subproofs. Our solution is to colour at most one subproof at a time: the smallest subproof, if it exists, that contains the current selection. For example, if in the final coloured proof the assumption of the second subproof is selected, the whole second subproof is coloured in grey:



Finally the labels `Coloured`, `Tagged` and `Coq` at the bottom of the window allow the user to select the output he/she wants. So far, we have been using the coloured output. Pressing the `Tagged` label gives the tagged version of the text. It is the one that is saved on disk. The `Coq` label gives the list of the generated conditions.

5 Related Works

We are aware that the gap between proofs on paper and their corresponding mechanised versions in Coq is mainly due to the fact that the tactic language of Coq has been thought as a little programming language whose goal is to build proofs. As in programming, conciseness and genericity in proof scripts are then privileged. This is not particular to Coq but common to all provers based on tactics. Scripts in such provers usually contain very few formulae. As coined in [18], adding formulae in a script increases the *viscosity* of the script. This means that it reduces the reusability of scripts and makes them less robust to modifications. The simple fact that formulae do not usually appear in proof scripts shows how inadequate scripts are to reflect the usual proofs.

There have been attempts to get a more natural language to interact with provers. The first and most impressive one by far is the Mizar project [20]. Other interesting attempts include [1,24,27,30]. Following the terminology used in [13], these systems propose a declarative style of proving, while systems like Coq offer a procedural approach. Declarative scripts usually contain lots of formulae and are then closer to proofs on paper. Unfortunately, these systems impose some strong restrictions on the way proofs should be written. In Mizar, for example, the proof has to be given with the level of detail imposed by the system. As the system has very little automation, proof scripts are often too

detailed for a human reader. A recent proposal [28] aims at relaxing this constraint. In Isar [27], some basic constructs are hard-wired. An example is the proof by case analysis, where the presentation of the different cases in the document has to follow the exact order in which the object was declared.

Other interesting approaches include attempts to accommodate both procedural and declarative styles [9,16,29], extract proof texts from tactics [6], extract proof texts from proof objects [2,7,23]. Note that when provers have proof objects, it would be possible to automatically convert proof objects into our format in a very similar way as in [2,7]. The result would most probably be far too detailed. However, with some support for improving the presentation while keeping the proof script consistent, this reverse engineering activity could be an effective way to get readable proofs.

6 Conclusion

In this paper we have presented a very simple and flexible format for writing formal proofs. This format is independent of a particular prover. Writing proofs is meant to be as natural as possible. With respect to the usual way of writing proofs, the author is only asked to explicitly indicate the proof structure. At each step it is then clear what the assumptions are and what the conclusion is. We have also presented a very simple user interface to help writing proofs in this format. With this interface, proofs are translated into our format by a simple colouring process.

Our approach is generic but with a special application in mind, i.e. the proofs in computer arithmetic. Its effectiveness relies strongly on the assumption that only looking at the terms in the proof is nearly sufficient to understand the entire proof. This is the case for proofs of computer arithmetic that mainly manipulate inequalities, but it is clearly not the case for any pencil and paper proof. An interesting question is whether the information that is added explicitly with tags in the proof could be synthetised instead. Doing this in full generality would require some non-trivial natural language analysis. A possibility would be to use special keywords or impose some predefined layout style, but this is exactly what we wanted to avoid using tags. The information that makes the proof mechanically checkable should interfere as little as possible with the way the proof is presented.

The separation of the activity of proof writing from the activity of proof checking has been motivated by our own experiment. We do not claim that this separation should always be the rule. In our case, it makes it possible to conciliate the necessity of publishing the proof in a human-readable form with the tedious task of mechanically checking the proof in all the low-level

details. In that respect we consider the mechanic checking of the proof just as a way to increase the confidence in the proof and not in any case as a way to substitute the refereeing process.

In the introduction we have described three drawbacks of the usual loose connection between the published proof and its machine-checked version. The first drawback was that the computer proof usually only checks the final statement of the published proof. In our framework we get a tighter connection. Every step of the published proof has been checked and for each step there is a corresponding lemma in the formal development. The second drawback was the difficulty of maintaining proofs. To ease maintenance a good practice is to always split big proofs into smaller pieces. Our generating process enforces this practice as every lemma only covers a single step of the proof on paper. The last drawback was the difficulty of experimenting with slightly different versions of the final statement. In our case, the variations can be done directly on the published proof by changing the statement. When re-running the proof script, the lemmas that the prover fails to re-establish directly correspond to the steps that need to be fixed in the published proof.

Some more work is still needed in order to turn our experiment into a realistic approach. First, the format has to be tested intensively against large proof developments. We are planning to use it to re-engineer our formalisation of floating-point numbers [8]. Second, the conditions that are generated are rarely provable automatically by Coq. Even if full automation is not our main goal, more tactics need to be developed in order to get a reasonable ratio of conditions proved automatically. Finally, we are investigating the possibility of using directly a scientific editor such as TeXmacs [25] to write proofs. This would give us for free the usual display for mathematical expressions. With a textual representation, notations as the one for integration or the one for matrices are known to be difficult to render.

References

[1] Andreas Abel, Bor-Yuh Evan Chang, and Frank Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic. In *PTP'01*, Siena, Italy, 2001.

[2] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. HELM and the Semantic Math-Web. In *TPHOLs'01*, number 2152 in LNCS, pages 59–74, Edinburgh, Scotland, 2001.

[3] Richard Bornat. Rendering Tree Proofs in Box Form. In *UITP'03*, pages 46–61, http://www.informatik.uni-bremen.de/uitp03/proceedings.pdf.

[4] Tim Bray, Jean Paoli, and C. Michael Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation.

[5] David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier. Mathematical Markup Language (MathML) version 2.0. W3C Recommendation.

[6] Avra Cohn. Proof Accounts in HOL. Unpublished draft, 1988.

[7] Yann Coscoy, Gilles Kahn, and Laurent Théry. Extracting text from proofs. In *TLCA*, number 902 in LNCS, pages 109–123, Edinburgh, Scotland, 1995.

[8] Marc Daumas, Laurence Rideau, and Laurent Théry. A Generic Library for Floating-Point Numbers and its Application to Exact Computing. In *TPHOLs'01*, number 2152 in LNCS, pages 169–184, Edinburgh, Scotland, 2001.

[9] David Delahaye. Free-Style Theorem Proving. In *TPHOLs'02*, number 2410 in LNCS, pages 164–181, Hampton, VA, USA, 2002.

[10] James Demmel and Yozo Hida. Accurate floating point summation. Available at http://www.cs.berkeley.edu/~demmel/AccurateSummation.ps.

[11] Jean-Christophe Filliâtre. Proof of Imperative Programs in Type Theory. In *TYPES'98*, number 1657 in LNCS, Eindhoven, Netherlands, 1998.

[12] Frederic B. Fitch. *Symbolic Logic: an introduction*. Ronald Press Company, 1952.

[13] John Harrison. Proof style. In *TYPES'96*, number 1512 in LNCS, pages 154–172, Aussois, France, 1996.

[14] John Harrison. A Machine-Checked Theory of Floating Point Arithmetic. In *TPHOLs'99*, number 1690 in LNCS, pages 113–130, Nice, France, 1999.

[15] G rard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant: A Tutorial: Version 6.1. Technical Report 204, INRIA, 1997.

[16] Gueorgui I. Jojgov, Rob P. Nederpelt, and Mark Scheffer. Faithfully reflecting the structure of informal mathematical proofs into formal type theories. In *MKM Symposium'03*, ENTCS, Edinburgh, Scotland, 2003.

[17] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, 1995.

[18] Nicholas Merriam and Michael Harrison. What is wrong with GUIs for theorem provers? In *UITP'97*, INRIA Report, Sophia-Antipolis, France, 1997.

[19] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA, Langley Research Center, 1995.

[20] Mizar. *Journal of Formalized Mathematics*. http://mizar.org/JFM/.

[21] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.

[22] David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.

[23] Jörg H. Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, and Martin Pollet. Proof development with OMEGA: Sqrt 2 is irrational. In *LPAR'02*, volume 2514 of *LNCS*, pages 367–387, 2002.

[24] Don Syme. Three Tactic Theorem Proving. In *TPHOLs'99*, number 1690 in LNCS, pages 203–220, Nice, France, 1999.

[25] TeXmacs. A Free Scientific Text Editor. http://www.texmacs.org/.

[26] Laurent Théry. Formal Proof Authoring: an Experiment. In *UITP'03*, pages 143–159, Rome, Italy, 2003.

[27] Markus Wenzel. A Generic Interpretative Approach to Readable Formal Proof Documents. In *TPHOLs'99*, number 1690 in LNCS, pages 167–184, Nice, France, 1999.

[28] Freek Wiedijk. Formal proof sketches.
Available at http://www.cs.kun.nl/~freek/notes/sketches2.ps.gz.

[29] Freek Wiedijk. Mizar Light for HOL Light. In *TPHOLs'01*, number 2152 in LNCS, pages 378–393, Edinburgh, Scotland, 2001.

[30] Vincent Zammit. On the Implementation of an Extensible Declarative Proof Language. In *TPHOLs'99*, number 1690 in LNCS, pages 185–202, Nice, France, 1999.

A Generated Conditions

```
Lemma c_gp_1: (x: Z) (y: Z) (Zlt ZERO x) -> (Zlt ZERO y) ->
     (Zge (exp x y) x).
Proof.
Intros x y gh_1.
Intros gh_1_1.
Apply ok.
Qed.

Lemma c_gp_2: (x: Z) (y: Z) (Zlt ZERO x) -> (`0` = y) ->
  (exp x y) = `1`.
Proof.
Intros x y gh_1.
Generalize (c_gp_1 x y gh_1); Intros gp_1.
Intros gh_2_1.
Apply ok.
Qed.

Lemma c_gp_3: (x: Z) (y: Z) (Zlt ZERO x) -> (Zgt ZERO y) ->
      (exp x y) = (Zdiv `1` (exp x (Zopp y))).
Proof.
Intros x y gh_1.
Generalize (c_gp_1 x y gh_1); Intros gp_1.
Generalize (c_gp_2 x y gh_1); Intros gp_2.
Intros gh_3_1.
Apply ok.
Qed.

Lemma exp_pos: (x: Z) (y: Z) (Zlt ZERO x) -> (Zgt (exp x y) `0`).
Proof.
Intros x y gh_1.
Generalize (c_gp_1 x y gh_1); Intros gp_1.
Generalize (c_gp_2 x y gh_1); Intros gp_2.
Generalize (c_gp_3 x y gh_1); Intros gp_3.
Apply ok.
Qed.
```