Literate proving
Presenting and documenting formal proofs

Paul Cairns & Jeremy Gow

UCL Intereaction Centre
University College London
London WC1E 7DP, UK
{p.cairns,j.gow}@ucl.ac.uk

Abstract. Literate proving is the analogue for literate programming in the mathematical realm. That is, the goal of literate proving is for humans to produce clear expositions of formal mathematics that could even be enjoyable for people to read whilst remaining faithful representations of the actual proofs. This paper describes maze, a generic literate proving system. Authors markup formal proof files, such as Mizar files, with arbitary XML and use maze to obtain the selected extracts and transform them for presentation, e.g. as LaTeX. To aid its use, maze has built in transformations that include pretty printing and proof sketching for inclusion in LaTeX documents. These transformations challenge the concept of faithfulness in literate proving but it is argued that this should be a distinguishing feature of literate proving from literate programming.

1 Introduction

Whilst formal languages, such as those used in formalised mathematics and programming, are ideal for communicating with a computer, they are far removed from the natural discourses that take place between humans in natural languages. Indeed, it can be argued that overcoming the distinction between human-human discourse and human-computer discourse is the entire basis for the discipline of human-computer interaction.

In formal mathematical proofs, humans may rapidly become lost in the detail and fail to understand the proof [19] or even to recognise them as a proof [4]. The corresponding difficulty of understanding programming languages has long been recognised. Documentation is considered to be a valuable resource for communicating the purpose and concepts embodied in the formal code [9]. However, it is also recognised that coding and documenting code are quite different activities and that the documentation of code is rarely of a high, readable standard.

To help overcome the division between program and documentation, Knuth devised the notion of *literate programming* [8]. Here the code and its explanation are combined together in a single document. Knuth's hope was that in doing so, programs would be faithfully described because when changes were made to the code, corresponding changes could be made to the documentation located in the same place. Of course, this is what people normally do with code comments.

However, a commented program does not make for easy reading for a human. Knuth's further step was to transform the single document in two different ways — one way to produce the compilable version of the code, the other to produce a human readable document. In this way, documentation would not only be faithful to the code but might even become an enjoyable literary work in its own right.

Formal mathematics shares many of the features of programming languages. In recent years, large repositories of formal mathematics such as the Mizar [10] and Coq libraries are being developed (and more are planned with on-going MKM). Whilst these libraries are not expected to change in the same manner as program code, there is nonetheless a need to document the formal language to help guide a reader through ideas, notation and even the trickier parts of formalisation [7]. In fact, Cruz-Filipe et al. (*ibid.*) recognise that some form of *literate proving*, that is, the weaving together of formal proofs and documentation, would be valuable.

This paper describes maze, a working system that implements literate proving. It deviates from Knuth's original implementation, instead following more modern versions of literate programming [14], as will be discussed in the next section. The way maze provides literate proving functionality is based on three key principles:

1. No changes made to the proof tool being used.
2. Minimal interference with the proof source files.
3. Minimal restrictions on the form of the proof or its presentation.

This is achieved by allowing the author to use arbitary XML markup to structure their proof source files and using comment syntax to 'hide' the markup from the proof tool. For instance, using maze an author of a Mizar article can extract parts of the article for ready inclusion in a LaTeX document, whilst still writing a normal, checkable Mizar article. The principles embodied in maze are quite general, making it widely applicable. We have so far used it to produce literate versions of proofs from the Mizar, Isabelle and Phox systems. The key point is that regardless of the underlying formal system, maze provides a good level of presentation with very little effort on the part of the author.

In order to aid discussion, the maze system will be described before we expand on the principles behind the system and literate proving in general. Most of the examples provided are taken from Mizar, specifically [18], as Mizar represents a substantial library of formal mathematics that could conceivably benefit from being presented via maze. Examples are also given for other proving systems to demonstrate the flexibility of this approach. Also, we have included as an appendix a short article presenting the proof of the irrationality of e as presented in the Mizar library. The maths in the article was generated entirely automatically from the original Mizar article [15][1] using maze.

[1] `http://www.mizar.org/JFM/`

2 From programming to proving

In Knuth's conception of literate programming, the source code for a program and the documentation of a program would be written jointly in a single document. This document could then be transformed one of two ways to give either the executable version of the program code or the typeset version of the documentation, as depicted in Figure 1. In this way, the source code as described in the documentation and the source code as compiled were one and the same — there was no duplication through cut-and-paste and no slightly different versions between that presented and that compiled. Morever, the original document did not need to follow the logical structure of the code at all but instead the documentation and the code were interleaved and interlinked as best fitted a literary exposition. For this reason, Knuth called his system `web`.

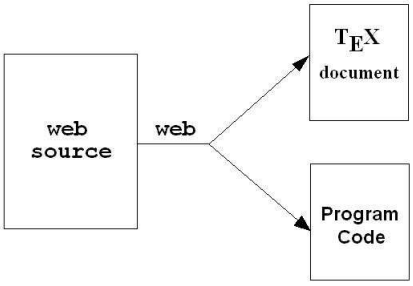

Fig. 1. The structure of Knuth's `web`

The drawback with this was that the program was actually written through a series of macros and that macro language needed to be learned. In addition, `web` and subsequent literate programming systems, such as `cweb` [13], tend to impose their own formatting. Though these native formats could no doubt be changed, they would require some intimate knowledge of the workings of the system.

Thus, the tightly coupled documentation and code of `web` imposes its own limitations. With regard to say the Mizar library, a literate proving system of this sort would also require refactoring significant parts of the Mizar library to be able to generate the original articles and their corresponding documentation.

Also, it must be noted that though the motivations of literate proving are similar to those of literate programming, the goals of literate proving are not. For example, the same piece of mathematics may be used in different ways. For instance, a description of the real line may strongly vary depending on whether it is to be used in developing the theory of fields or as a prototypical metric space. It is unlikely that similar concerns would arise in program description.

This suggests that a literate proving system would require rather looser coupling between the proofs and their documentation. A more recent literate programming system, Warp, shows a possible way forward [14]. With warp, the literate programming system is essentially a way of extracting code from a program and into manageable chunks. These chunks can be incorporated, in any order, into a larger document that describes what they are and also typesets them appropriately. This is precisely the model used in maze and is shown in Figure 2.

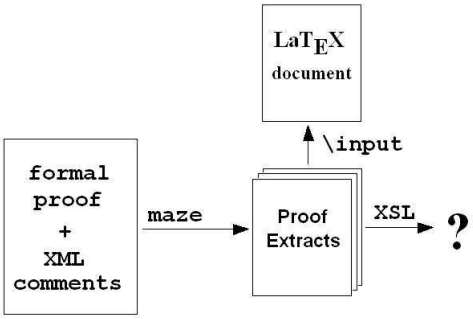

Fig. 2. The structure of maze

With maze then, the principles of literate programming are maintained. That is, there is only one version of the formal mathematics and there is a freely flexible relationship between the description of the mathematics and its logical structure. This clearly comes at the cost of a separation between the formal mathematics and the documentation. The risk then is of course that the source mathematics could change independently of its description. However, the author who cared about the description would still be obliged to update the documentation if it were to make sense.

The next section gives details of how maze extracts chunks of Mizar articles. The interested reader is refered to Thimbleby's paper on warp for a deeper discussion of the various types of literate programming systems and their relative merits and drawbacks [14].

3 maze

For maze to extract the required parts of a formal proof script, it is necessary to insert markers in the proof. Of course, any markers will necessarily stand out from the formal language. To prevent the markers from interfering in the validation of the proof, the comment tags for the article are used to hide the maze instructions from the formal system. The instruction to maze, within a

comment, is in the form of an XML tag. For example, in Mizar, the comment tag is :: and so the tags marking the beginning and end of a portion to extract would be:

```
::<maze id="extract">
...Mizar article...
::</maze>
```

As you might expect, when maze encounters this pair of tags, everything between (and including) the tags is written to the file `extract.xml`.

Note that apart from the comment tags for the particular formal language, maze requires no other knowledge of the underlying language. This makes adapting it to other formal languages relatively straightforward.

The XML style, following Warp [14], seems a natural choice. A model of maze is that it extracts any portion of an article between such tags and in the process drops the comment markers from any such tags. Any non-XML comments are marked with `<comment>` tags. From this simple process, to use the extracts in documents such as LaTeX or HTML, further transformations of the extracted parts are optionally performed using XSL. For example from the marked up section of Mizar as follows:

```
::<maze id="demo">
::<theorem><statement>
theorem Th21:
 -K = -Q implies K = Q
::</statement><proof>
  proof
    ::Uses the natural property of inverses
  --K = K & --Q = Q
   ::<ref>
   by PRE_TOPC:20;
   ::</ref>
  hence thesis;
  end;
::</proof></theorem></maze>
```

maze produces the following:

```
<maze id="demo">
<theorem>
<statement>
theorem Th21:
 -K = -Q implies K = Q
</statement>
<proof>
  proof
<comment>
Uses the natural property of inverses
</comment>
  --K = K & --Q = Q
```

```
<ref>
   by PRE_TOPC:20;
</ref>
  hence thesis;
  end;
</proof>
</theorem>
</maze>
```

However, rather than require an author to also have knowledge of XSL or to spend time on developing an appropriate stylesheet, maze has certain transformations built-in. These are described in the following subsections.

3.1 maze for LaTeX

The simplest method for getting text into LaTeX is to have maze produce plain text extracts within the LaTeX `verbatim` environment. If the target, though, is truly readable LaTeX then of course a `verbatim` version of text falls short of nicely presented mathematics in several ways. Specifically concentrating on Mizar, the Mizar language is entirely in standard ASCII text and so lacks the finesse of the rich character sets and symbols that LaTeX has. For this reason, maze is also able to produce a pretty version of the output from Mizar articles suitable for LaTeX. For example, here is a theorem and its proof that has been pretty printed from [18]:

THEOREM Th_{20} :
$Kc = Q$ iff K misses $-Q$
PROOF
$A_1 : -Q = Q'$ by STRUCT_0 :def 5;
hereby assume $Kc = Q$; then
$K \setminus Q =$ by XBOOLE_1 : 37; then
$K/ \setminus Q' =$ by SUBSET_1 : 32;
hence K misses $-Q$ by A_1, XBOOLE_0 :def 7;
END;
assume K misses $-Q$; then
$K/ \setminus -Q =$ by XBOOLE_0 :def 7; then
$K \setminus Q =$ by A_1, SUBSET_1 : 32;
hence thesis by XBOOLE_1 : 37;
END;

The pretty printing follows some simple heuristics to produce this output. These are as follows. First, single characters have been interpreted as variables and so are put into the math environment. Trailing numerals after a string of letters are assumed to be subscripts. Thirdly, keywords such as "theorem" have been put into the small caps font. Finally, characters that would normally

correspond to LaTeX control codes have been appropriately converted to appear correctly in the final document.

Though the output is obviously more "latex-y", it could not really be called pretty. Symbols that are a natural feature of Mizar such as `c=` for subset, appear peculiar in LaTeX as $c =$. Though heuristics could also be developed to handle these, they would begin to require more detailed parsing of the Mizar source and this is not without hazard or even always possible [5]. Instead, a lighter method has been chosen that places the control in the hands of the author. When extracting text for pretty LaTeX, `maze` is also able to consult what we have called a match file. In this, strings from the Mizar article appearing in the match file have replacement LaTeX code that is used instead of the string. The match file takes priority over any of the other heuristics. The result appears like this:

THEOREM Th_{20} :
$K \subseteq Q$ iff K misses $-Q$
PROOF
$A_1 : -Q = Q'$ by `struct_0` :def 5;
hereby assume $K \subseteq Q$; then
$K \setminus Q = \emptyset$ by `xboole_1` : 37; then
$K \cap Q' = \emptyset$ by `subset_1` : 32;
hence K misses $-Q$ by A_1, `xboole_0` :def 7;
END;
assume K misses $-Q$; then
$K \cap -Q = \emptyset$ by `xboole_0` :def 7; then
$K \setminus Q = \emptyset$ by A_1, `subset_1` : 32;
hence thesis by `xboole_1` : 37;
END;

In this example, the ASCII representations of some symbols have been replaced with the corresponding LaTeX symbols. Other symbols have been replaced with a more usual one such as the replacement of `{}` with $\emptyset$. Also, the match file need not be confined to mathematical symbols and here has been used to make the references to other articles appear in a different font. The match file that was used here consisted of only eight lines of text, one line per match and its replacement, and so could easily be produced by an author.

The example in appendix A uses the match file extensively to help produce a version of the Mizar proof that uses the standard summation notation for series. The appendix also uses `skip` tags to help present the proof — these work as described in the next section but for pretty printing insert ellipsis to show omitted material.

3.2 Proof sketching with maze

Literate proving, like literate programming, should strive to be about producing a document that not only elucidates the proofs presented but also makes that

experience enjoyable to the human reader. However, as is well known, formal mathematics of the sort found in the Mizar library has a tendency to be verbose and for key insights to be lost in the detail [16, 6]. To aid in exposition and enjoyment, some form of simplification is often necessary.

For this reason, maze is able to automatically generate proof sketches. The form chosen was that proposed by Wiedijk as a way to aid constructing formal proofs in Mizar [17]. The two things that distinguish a formal proof sketch from a completed Mizar proof are: references in support of proofs steps are always omitted; some proof steps are also omitted. The sketch should therefore be a summary of the most salient proof steps of the formal proof.

The omission of references is easily achieved automatically in maze since all references are indicated by the keyword `by`. In contrast, the assessment of which proof steps are most salient can only be made by the author. For this reason, a second maze specific tag was introduced, namely `skip`. Where `skip` tags are placed within `maze` tags, all of the text except the text marked up to be skipped is placed in the file specified by the `maze` tag. The result is put in the `verbatim` environment. Here is an example of a complete Mizar proof that has been sketched to appear in this article:

```
theorem
 P is dense & Q is dense & Q is open implies P /\ Q is dense
 proof
  assume that A1:P is dense and A2:Q is dense and A3:Q is open;
     [#] TS c= Cl(P /\ Q)
     proof
      now let C be Subset of TS; assume A7: C is open;
       assume x in C;
       then  Q meets C  then
       A8:Q /\ C <> {}
       Q /\ C is open
       then P meets (Q /\ C)  then
              hence (P /\ Q) meets C
      end;
    hence thesis
    end;
    then Cl(P /\ Q) = [#] TS
   hence thesis
  end;
```

Of course, this output could also be pretty printed if required. Any automation of the proof sketching necessarily needs to know something about the structure of the formal language. Currently, therefore, maze only does proof sketching for Mizar.

3.3 Tactic-style proof assistants

Having illustrated maze's functionality using Mizar, we now briefly show the versatility of the system with example based on the Phox system [12]. Phox is a

proof assistant with a tactic-style form of interaction: the user specifies how the system should transform the proof state, in a similar fashion to Isabelle, HOL and many other proof tools. This is a very different form of interaction to the declarative proof style of Mizar — nonetheless, maze can still be used for literate proving.

As with Mizar, a section of Phox proof file can be marked up freely by the author, as follows:

```
(* <maze id="square"><theorem> *)
(* Product of squares equal to square of the product. *)
(* <statement> *)
fact square.mult /\x,y (square x * square y = square (x * y)).
(* </statement><proof><skip> *)
intros.
(* </skip><rewrite> *)
unfold square.
(* </rewrite> *)
(* (x.x).(y.y) = (x.y).(x.y) *)
(* <calc> *)
rewrite -p 1 mult.assc.R.
rewrite -p 4 mult.comm.R.
(* </calc> *)
(* = (x.(x.y)).y *)
(* <final> *)
rewrite mult.assc.R.
(* <skip> *)
trivial.
(* </skip></final></proof></theorem></maze> *)
```

Here the author has chosen to mark up the short six-line proof as three steps: two `<rewrite>` steps (one line each) and a `<calc>` step (two lines), with the trivial first and last lines skipped over. They have also annotated the script with comments that declarative describe some of the proof states. This is transformed into the following XML:

```
<maze id="square">
<theorem>
<comment>
Product of squares equal to square of the product.
</comment>
<statement>
fact square.mult /\x,y (square x * square y = square (x * y)).
</statement>
<proof>
<rewrite>
unfold square.
</rewrite>
<comment>(x.x).(y.y) = (x.y).(x.y)</comment>
<calc>
rewrite -p 1 mult.assc.R.
```

```
rewrite -p 4 mult.comm.R.
</calc>
<comment>= (x.(x.y)).y</comment>
<final>
rewrite mult.assc.R.
</final>
</proof>
</theorem>
</maze>
```

This example illusrates how maze allows the author to freely annotate and structure their proofs, and consequently to present them in any way they like. This freedom requires the author to design their own presentations (or to use others designs), but allows them to design and present literate formal proofs, where the structure is tailored to suit their presentation needs.

3.4 Implementation details

maze is implemented in Java with the formal source being either passed as a command line parameter or through the standard input. A single source file may have multiple `<maze>` sections extracted, each going to an individual file as specified in the `id` attribute. If `id` has value `foo` the output file is `foo.xml`, if it is omitted the output is sent to the standard output.

The system can configured for a particular proof tool by providing it with a simple description of the system's comment syntax. The different transforms described in the paper are produced by engaging various modes via command line options. These modes are:

Raw : Extracts data without any changes.
Text : XML tags are not shown in output.
All : Data within of skip tags is extracted.
Verb : Places a LaTeX verbatim environment around the output.
Suffix : changes the filename suffix of the output file.
Help : produces a summary of the flags and match file structure.

maze is freely available along with all the source code used in the production of this article from the author or from the web-site: `www.uclic.ucl.ac.uk/paul`.

4 The principles of literate proving

First and foremost, it is worth noting that maze embodies a generic set of principles, like the Warp literate programming system [14], that are not specific to this particular implementation. These are:

1. The use of "commenting out" for maze instructions so that they are ignored by a proof checker
2. Proof extracts being XML marked up for further transformations

3. The match file concept to aid pretty printing without deep semantic knowledge of the formal language
4. The use of `skip` to aid proof sketching
5. A small and simple instruction set (currently two commands).

Of course, there are specific features such as the pretty printing of keywords in small caps and how proof sketching is performed that are specific to Mizar. Even then, it should be noted that Mizar, as a large body of formal mathematics, is being used to demonstrate literate proving but maze is not deeply entangled in that particularly library. maze can be trivially reconfigured to work with other proof tools — we have so far also used it with Phox and Isabelle. Where literate proving differs across these systems is not in the nature of the tool but actually what it means to make the different sorts of formal proofs literate: contrast the use of maze with Mizar's declarative proofs against Phox's procedural proof scripts. Literate proving allows us to attach some structured declarative information to selected procedural steps, greatly improving proof presentation for these systems.

The general concept of literate proving though does seem to have separate concerns from literate programming. Literate programming is concerned with producing programmes that are enjoyable for human readers to read. Likewise, literate proving is about producing formal proofs that are enjoyable for humans to read. Literate programming has the goal also to be faithful to the actual program, hence the close coupling of the code and the documentation. In maze, it is possible to have entirely faithful presentations of proofs but in fact it produces more readable documents if they are pretty printed and also sketched. Thus, the presentation of the proof can differ significantly from the original. This could have consequences where a person reading both literate proofs and the original proofs is unable to easily make a connection between the two. The impact of the superficial differences may be not be so superficial and only time will tell.

Tools such as maze provide a guarantee that faithfulness is preserved — the presented proof is automatically generated from the original. Of course, the author defines any match file used in pretty printing but it is to be hoped that an author would make logical or at least acceptable choices for ensuring that Mizar symbols are replaced with suitable LaTeX symbols. Where unusual choices of matching symbol have been used, the author should have some responsibility to explain their choice.

Another difference between literate programmming and proving is that the description of a program is deliberately quite specific to the tasks of that particular program. The documentation is therefore aimed at explaining how the program meets the overall tasks or how subtasks lead to the completion of the tasks. Standard algorithms such as quick sort would not be the most interesting ones to explain in a program's documentation. In contrast, literate proving could have a number of goals.

A literate proof could be along the lines of a literate program to explain how a particular formal proof represents a more traditional proof or how particular features of the formal language were used. Unlike programs though, proofs are

objects of interest to mathematicians in and of themselves not just the tasks they achieve [19]. They are communication acts that lead to other ideas and mathematicians may wish to explain a proof in different ways depending on how that proof is used in a particular domain. For example, a diagonalisation proof may be explained very differently for a mathematician than for a computer scientist. Moreover, there can be no sense in which a given explanation that is not merely about language specifics could be sufficient for all time. Fortunately, maze's implementation is such that there is no constraint to have only one explanation but it provokes interesting questions as to the role of literate proving in the wider realm of mathematical knowledge management.

5 Related Work

Several proof tools already provide some form of document generation. With the Coq system [3] comes the CoqDoc tool to transform Coq proof files into LaTeX and HTML. CoqDoc allows pretty printing of formal objects, explanatory text in comments (including LaTeX/HTML commands) and hiding of subparts of the document. maze reproduces this functionality, and with an appropriate stylesheet could produce identical presentations. However, although some presentation details may be specified, CoqDoc produces documents of a roughly fixed structure and style. In contrast, with maze the author is free to choose both the structure and style: any form of XML markup may be used and transformed to a range of presentation formats and styles. While CoqDoc is well suited to producing more readable versions of proof files in a standard LaTeX or HTML format, maze allows much more flexible document generation.

We believe literate proving to be better supported by systems that give the author freedom to markup and present formal proofs as she sees fit, rather than imposing a standard structure that is bound to the underlying proof files. Apart from CoqDoc, all other systems we are aware of suffer from similar drawbacks. For example, Isabelle/Isar allows generation of LaTeX and HTML, but of a predetermined structure and style [11]. The HELM project [2] has developed an XML generation tool for Coq, but the structure is that of the underlying proof terms, and does not provide support of a more free-form development of literate proofs.

6 Conclusion

Literate proving is the analogue of literate programming for formal proof languages and maze is a system that implements literate proving as demonstrated on the Mizar library. It is clear from maze that some straightforward principles can be used to do literate proving for any formal mathematical language but that some form of pretty printing and sketching is likely to be desirable. It is also worth noting that literate proving is not entirely a mathematical version of literate programming but rather literate descriptions of proofs could be manifold

and need updating as more mathematics is learned. Quite how literate proofs may fit into the wider body of mathematical knowledge remains to be seen.

Acknowledgements

Thanks to Harold Thimbleby for his comments on both the paper and the maze system.

References

1. A. Asperti, B. Buchberger & J.H. Davenport (2003), editors, Mathematical Knowledge Management, Proceedings of MKM 2003. LNCS **2594**, Springer.
2. A. Asperti et al (2003), Mathematical Knowledge Management in HELM. *Annals of Mathematics and Artificial Intelligence*, **38**(1):27–46.
3. Y. Bertot & P. Castéran (2004), *Coq'Art: The Calculus of Inductive Constructions.* Springer. See also `http://coq.inria.fr/`
4. P. Cairns & J. Gow (2003), A theoretical analysis of hierarchical proofs. In [1], pp175–187.
5. P. Cairns & J. Gow (2004), Using and parsing Mizar. *Electronic Notes in Theoretical Computer Sci.*, **93**:60-69.
6. P. Cairns, J. Gow & P. Collins (2003), On dynamically presenting a topology course. *Annals of Mathematics and Artificial Intelligence*, **38**:91–104.
7. L. Cruz-Filipe, H. Geuvers & F. Wiedijk (2004), C-CoRN, the constructive Coq repository at Nijmegen. In A. Asperti, G. Bancerek, A. Trybulec (eds), Mathematical Knowledge Management, Proceedings of MKM 2004. LNCS **3119**, Springer, pp88–103.
8. D. E. Knuth (1984), Literate programming. *The Computer Journal* **27**:97-111.
9. S. McConnell (1993), *Code Complete: A practical handbook of software construction.* Microsoft Press.
10. The Mizar Mathematical Library `http://mizar.org`
11. L. Paulson (1994), *Isabelle: a generic theorem prover.* Springer.
12. C. Raffalli & R. David (2002), Computer Assisted Teaching in Mathematics. *Proc. Workshop on 35 Years of Automath*, Edinburgh.
13. H. Thimbleby (1986), Experiences of 'literate programming' using cweb (a variant of Knuth's WEB). *The Computer Journal* **29**(3):201–211.
14. H. Thimbleby (2003). Explaining code for publication. *Software — Practice and Experience* **33**:975–1001.
15. F. Wiedijk (1999), Irrationality of e. *Journal of Formalized Mathematics* **11**(42).
16. F. Wiedijk (2000), *The De Bruijn Factor*, Poster at TPHOL 2000.
17. F. Wiedijk (2003), Formal Proof Sketches. Types for Proofs and Programs, proceedings of TYPES 2003. LNCS **3085**, Springer, pp378–393.
18. M. Wysocki, A. Darmochwał (1989), Subsets of topological spaces. *Journal of Formalized Mathematics* **1**(28).
19. C. Zinn (2004), *Understanding Informal Mathematical Discourse.* PhD Thesis, Arbeitsberichter des Instituts für Informatik, Friedrich-Alexander-Universität, **37**(4).

A Example: the irrationality of e

As an extended example we present a short article based on Freek Wiedijk's Mizar proof of the irrationality of e [15]. Here maze has been used to provide pretty printed extracts from a marked-up copy of the original Mizar article. Some of the extracts are sketches of the underlying formal proofs.

A.1 Overview

THEOREM
e is irrational

In the Mizar library, e is defined as the usual infinite sum. More explicitly, e is the sum of the sequence, eseq, where:

DEFINITION
func eseq $->$ Real_Sequence means
: Def_5 : for k holds it.$k = 1/(k!)$;

Briefly, the proof mainly considers the terms of eseq multiplied by $n!$ for some appropriately chosen n. In this case, if e were rational, the sum of the final n terms of eseq$\times n!$ would have to be an integer. However, this expression can be bounded by a geometric series and hence must be a positive integer strictly between 0 and 1 — a contradiction.

A.2 Bounds on the terms of eseq $\times n!$

First we require two lemmas on the terms of eseq. The proof of the first lemma is a straightforward induction on k and tells us that each term of the sequence eseq $\times n!$ is bounded by a corresponding term in the geometric series.

THEOREM Th_{39}
$x = 1/(n+1)$ implies $(n!)/((n+k+1)!) <= x \uparrow (k+1)$

The second lemma gives us a bound for the tail of the series for e. Note that the bound is in fact the sum of the geometric series from the previous lemma.

THEOREM Th_{40} :
$n > 0 \& x = 1/(n+1)$ implies $n! \times \sum_{n+1}^{\infty}(\mathrm{eseq}) <= x/(1-x)$
PROOF
⋮
$A_4 : 0 < x \& x < 1$ by A_1, A_2, REAL_2 : 127, SQUARE_1 : 2;
deffunc $F(\mathrm{Nat}) = x \uparrow (\$1 + 1)$;
consider seq being Real_Sequence such that
A_5 : for k holds seq.$k = F(k)$ from SEQ_1 :sch 1;
⋮

then A_{10} : seq is summable & $\sum$(seq) =seq.0/$(1-x)$ by A_4, A_7, SERIES_1 : 29;
$A_{11} : \sum(\text{seq}) = x/(1-x)$ by A_6, A_8, A_9, SERIES_1 : 29;
A_{12} : $(\text{eseq})^{\infty}_{k=n+1}$ is summable by Th_{24}, SERIES_1 : 15;
now let k;
$A_{13} : (n!(\#)((\text{eseq})^{\infty}_{k=n+1})).k = n! \times (((\text{eseq})^{\infty}_{k=n+1}).k)$ by SEQ_1 : 13
$. = n! \times \text{eseq}.(n+1+k)$ by SEQM_3 :def 9
$. = n! \times (1/((n+k+1)!))$ by Def_5
$. = n!/((n+k+1)!)$ by XCMPLX_1 : 100;
hence $(n!(\#)((\text{eseq})^{\infty}_{k=n+1})).k >= 0$ by Th_{34};
$\text{seq}.k = x \uparrow (k+1)$ by A_5;
hence $(n!(\#)((\text{eseq})^{\infty}_{k=n+1})).k <= \text{seq}.k$ by $A_1, A_{13}, \text{Th}_{39}$;
END;
then $\sum(n!(\#)((\text{eseq})^{\infty}_{k=n+1})) <= \sum(\text{seq})$ by A_{10}, SERIES_1 : 24;
hence $n! \times \sum^{\infty}_{n+1}(\text{eseq}) <= x/(1-x)$ by A_{11}, A_{12}, SERIES_1 : 13;
END;

A.3 The proof of the irrationality of *e*

PROOF
assume e is rational;
then consider n such that $A_1 : n >= 2 \& n! \times e$ is integer by Th_{32};
$A_2 : n! \times e = n! \times ((\sum^{n}_{1}(\text{eseq})\ ) + \sum^{\infty}_{n+1}(\text{eseq}))$
by $\text{Def}_6, \text{Th}_{24}$, SERIES_1 : 18
$. = n! \times (\sum^{n}_{1}(\text{eseq})\ ) + n! \times \sum^{\infty}_{n+1}(\text{eseq})$;
reconsider $N = n! \times e$ as Integer by A_1;
reconsider $N' = n! \times \sum^{n}_{1}(\text{eseq})$ as Integer by Th_{38};
$A_3 : n! \times \sum^{\infty}_{n+1}(\text{eseq}) = N - N'$ by A_2;
set $x = 1/(n+1)$;
$A_4 : x/(1-x) < 1$ by A_1, Th_{41};
$n > 0$ by A_1;
then $n! \times \sum^{\infty}_{n+1}(\text{eseq}) <= x/(1-x)$ by Th_{40};
then $n! \times \sum^{\infty}_{n+1}(\text{eseq}) < 0+1$ by A_4, AXIOMS : 22;
then $n! \times \sum^{\infty}_{n+1}(\text{eseq}) <= 0$ by A_3, INT_1 : 20;
hence contradiction by Th_{36};