The PracTEX Journal, 2007, No. 1
Article revision 2007-1-31/

Writing and Checking Complete Proofs in TEX

Bob Neveln
Bob Alps

Abstract TEX files are text files which are readable by other programs. Mathematical proofs written using TEX can be checked by a Python program provided they are expressed in a sufficiently strict proof language. Such a language can be constructed using only a few extensions beyond the syntax of Morse's book [5], one being the incorporation of explicit theorem number references into the syntax. Such a program has been applied to and successfully checked the theorems in a significant initial segment of a book length mathematical manuscript.

1 Introduction

The present work is an unplanned side-effect of a book project by the authors [7]. As work on the book progressed proofs were written more and more carefully. Programs in Python were developed to check the mathematical syntax, then to renumber theorems following insertions or deletions and finally to check the proofs written.

These developments were possible because the book is written in TEX using a formal mathematical language. Although most mathematical text is intended to be formalizable, usually in terms of first order predicate logic, it is almost never formal as written. Checking proofs written in a conventional style would consequently require a formalization step requiring clarification of the author's intentions on many details. Checking proofs written in a formal language obviates these difficulties. In the work presented here we use a syntax derived from that of A. P. Morse. In his book, *A Theory of Sets* [5], he presented a formal syntax which was used to express all the definitions and theorems in the book. ([6] contains a mathematical exposition of this syntax.) A key feature of his treatment of

mathematical language was the inclusion of definitions themselves into the formal syntax, see [1]. The first theorem of the book was given a complete proof, but no attempt was made to continue the presentation of complete proofs. Indeed with the small set of inference rules given this would not have been feasible.

The formal syntax of Morse's book enabled the creation of a program capable of parsing its language and checking some of its theorems using an expanded inference rule set, as early as 1966 at Sandia Laboratories, [3]. Soon after that most of the mathematics in the book was checked by W.W. Bledsoe working at MIT.

This paper describes additions to Morse's syntax implemented in TEX and Python programs which together enable writing and checking complete proofs. The resulting environment is a work in progress.

2 Tools and Files

Unix utilities are based on the idea that it is good to have many tools each of which does a single task well. The environment described here to enable writing and checking complete proofs consists of many different TEX files and Python programs. As noted by Richter in [8], it is easy to write Python scripts which conveniently operate on TEX files. Those described here include:

- a program which checks the syntax of the mathematics in the TEX file,
- a program which renumbers the propositions,
- a program which adds horizontal space to mathematical expressions not ideally rendered by TEX,
- a program which checks a proof whose number is given as a command line argument.

The logic on which proofs depend is supplied in a variety of ways. Some logic is built into the parser which parses $(x < y < z)$ for example as $(x < y \wedge y < z)$. Some logic is built into the checking program which uses the commutative and associative properties of "and" as well as the transitivity of numerous relations including logical implication and set inclusion. Most of the logic resides in a file of rules of inference which is consulted in a blind linear search each time a step of

the proof to be checked is attempted. Another file consists of propositions which are generally recognized as obvious such as

$$(x \in A \wedge A \subset B \rightarrow x \in B)$$

Further logic consisting of material which is at least as elementary, but ordinarily "below the radar" of everyday mathematics is listed in a special appendix added to the work being checked. It uses the logic developed in [2].

The tools in the set create an environment in which a work cycle related to an ongoing paper or book, consisting of steps like that involved in writing a computer program:

1. Add the statement of a theorem and its proof to a TEX file.
2. Run TEX to get a viewable DVI file and detect TEX errors.
3. Run the parser to find mathematical syntax errors.
4. Run the check program to find logical errors and gaps in the proof.

At the very end it is also useful to run a program which uses the parser to add horizontal spacing at points where TEX would otherwise crowd symbols.

Because all the logical steps which can be checked at this time are quite small, the process is both arduous and tedious.

3 Proof Syntax

The basis for the proof syntax is the mathematical syntax of Tony Morse's book, [5]. Changes to Morse's mathematical syntax including additional abbreviation schemes and restrictions on the format of bound variable forms are introduced but do not alter the mathematical language markedly. To get a notation capable of expressing complete proofs just a few additional elements suffice.

3.1 Mathematical Syntax

Morse's mathematical syntax can be done in TEX if two main obstacles are overcome.

1. Morse's language contains some special symbols. These can be created with Metafont.
2. It also contains a great many operators with non-italicized Latin letters. In plain TEX there are 32 of these defined using macros such as

   ```
   \def\sin{\mathop{\rm sin}}
   ```

   We have found the need for hundreds of these. They have been stored in a separate include file.

3.2 Reference Numbering

An important element in the proof syntax described here is the inclusion of theorem numbers themselves into the syntax.

An example from the manuscript [7] follows:

```
\tabc 1.17 $(b\in\bfun \Iff \Patch_0 b\in\U)$
\lineb Proof:
\notea 1$(b \in \bfun$
\linec $\c\Patch_0 b\in \SI \rng b\setdif\dmn b$\By 1.16
\linec $\c\Patch_0 b \in \U)$ \By 01.14
\lineb
\notea 2$(\Patch_0 b\in\U$
\linec$\c\ex\Patch_0 b$ \By 01.8
\linec$\c b\in\bfun)$\By 1.13
\lineb \Bye .1, .2
\lineb
```

In this example a theorem numbered 1.17 is stated and proved. The statement involves the plain TEX macro '\in' as well as other macros such as '\c' for '\rightarrow' and '\Iff' for '\leftrightarrow'. The '\tabb', '\notea' and '\By' macros perform space formatting, but also serve as reference handles for the checking program. For example Theorem 1.16, which is referred to at the end of the second line of the proof, must be identified by a '\tabc' macro. The '\lineb' and '\lineb' macros have only a space formatting role. The '\Bye' macro prints QED and indicates that the theorem itself is to be checked.

References such as the closing '.1' and '.2' refer to the notes tagged by the '\notea' macros. The zero-plus references 01.14 and 01.8 point to the file of "obvious" theorems.

Propositions which are referenced must have a traditional number-dot-number identification which is used to invoke them in proofs. This numbering convention is similar to that produced by LaTeX but less flexible. It is used instead of LaTeX because its use requires slightly less labor and the labor involved in specifying references is a large component of the work of specifying a complete proof.

A Python program is needed for renumbering the theorems when theorems are inserted, deleted, or moved.

3.3 Significant Punctuation

Reference notations may include punctuation. The punctuation marks must be identical to the corresponding mark in the rule of inference. If rules are marked in such a way that rules of a similar nature get similar punctuation, then a meaning is associated with the punctuation mark. The semi-colon for example is used in references that have a major premise followed by minor premises. For example if in note 5 below we prove a result q by using a theorem $(p \to q)$ which is numbered 1.23 and we have previously obtained p in note 3 then we might have the following note to establish q:

Note 5 $(-a \in \mathrm{Z})$ ‡ 1.23; .3

In order for this note to be checked there must be a theorem 1.23 such as

Thm 1.23 $(x \in \mathrm{Z} \to -x \in \mathrm{Z})$

a previous note 3

Note 3 $(a \in \mathrm{Z})$

and a rule of inference which has the form

From: $(p \to q)$; p

Infer: q

The semi-colon in the reference limits the number of rules which match a given inference. The intended meaning of the semi-colon is that it sets the "major premise" apart from the "minor premises." At present approximately 190 of the stored inference rules use the semi-colon to separate major and minor premises. Another example of such a rule is the following rule:

From: $(p \rightarrow q \leftrightarrow r); q$

Infer: $(p \rightarrow r)$

Further developments towards a syntax of reference expressions will no doubt be found useful.

3.4 Given-Hence Blocks

Notes which are not proven but which are merely "given" may be introduced using ‡ G, in place of a proof reference. These remain in force until a "hence" referring to them is encountered. The "hence" attaches the given notes to the "henced note" as explicit hypotheses. The "henced" note is tagged using ‡ H as a proof reference. For example we might have:

Note 2 $(x \in A)$	‡ G
..	
Note 7 $(x \in B)$	‡ .2,....
Note 8 $(x \in A \rightarrow x \in B)$	‡ .7 H .2

The variables introduced in each Given note are local to that block. Reference may be made to notes 2-7 only from within that block, only so long as note 2 is in force in other words.

3.5 Local Definitions

Sometimes it is useful to introduce locally defined variables. To do this we may "set" a variable to a described object. A note of this form is justified by ‡S and it retains validity as long as the variables on which it depends do. For example given a non-empty set A it is useful to have a name for a member of A.

Note 2 $(A \neq \emptyset)$	‡ G
Note 3 $(a \equiv$ an $x(x \in A))$	‡ S
Note 4 $(a \in A)$	‡ .2,.3

This feature of the proof syntax depends on using a logic which allows descriptions, see [2].

3.6 Reasoning Chains

A note may consist of lines, all but the first of which are introduced by some transitive relation. In this case each consecutive pair of lines defines a step to be given its own proof. The note is then telescoped when used as a reference. For example:

Note 7 $(A \subset B$	‡ ...
$\subset C)$	‡ ...

Here the inclusions $(A \subset B)$ and $(B \subset C)$ are checked separately, but if note 7 is referred to later, just the inclusion $(A \subset C)$ will be invoked by this reference.

4 The Unifier

Each step to be checked is matched against rules of inference in a blind linear search. Each rule whose sequence of arguments and punctuators matches with numerical references and punctuators in the reference note is submitted to a unifier. If a unification is found the step is checked.

The unifier is based on standard first order unification. It does however go beyond standard first order unification in two ways. Although much less general than [4], it allows the terms of a conjunction to be re-ordered in order to accomplish a match. It also attempts to match the second order variables which occur in Morse's language.

It is written to succeed or fail quickly. It may fail to find a unifier even when one exists. For example if a conjunction with n conjuncts is matched against a conjunction '$(p \wedge q)$', where 'p' and 'q' are unmatched variables, this unification will not be attempted because of the $(2^n - 2)$ different possible matchings. A rule

of inference must avoid presenting such unifications to the checker or it will be ignored. The unifier does not aim at any ambitious sort of completeness.

5 Results and Prospects

The manuscript being checked contains over 1200 theorems, with proofs in various stages of completion. Roughly the first 120 of these have been checked. This number is increasing as the work progresses.

As the work proceeds, bugs are encountered in the checking program, as well as cases which should check but do not. The program is then revised, rules of inference and "obvious" theorems are stored to the reference files. There are now almost 600 rules of inference and over 400 "obvious" theorems in the zero-plus references file. The checking program now contains about 4000 lines of code. The manuscript also has appendices containing over 150 elementary results which are referenced in the proofs.

The program takes as input the number of a single theorem to be checked. Although Python is an interpreted language, the program executes in a few seconds on a machine of recent vintage.

The proof syntax at its present stage of development is and should be "low-level." Once avenues of checkable proof begin flowing it will be time for the appearance of higher levels of expression which will attenuate to some extent the labor of picking through all the details of a proof.

Because very elementary facts must be individually listed in a file, mathematics involving significant amounts of computation would not be suitable for this treatment. It is intended for abstract mathematics more or less without restriction. Abstract algebra, or any mathematics which uses sets "endowed" with structure, raises problems of formalization. These problems were encountered in [7] and one approach to coping with them is discussed in Appendix D of this work.

6 Observations

We close with a few observations.

1. Including the details necessary to get a proof to check requires roughly an order of magnitude more time than writing a conventional proof.

2. Proofs stated in checkable detail become longer by a factor of two or three.
3. Reading checkable proofs requires slightly more effort on the part of a knowledgeable reader than reading a conventional proof.
4. Checkable proofs can be read by mathematicians who are not specialists.

7 Conclusion

Despite its preliminary and incomplete nature the checking program as it stands now shows that it is practicable to write and check complete proofs, given a willingness to adopt a formal language and to submit to the discipline of itemizing all necessary references.

Readers interested in trying out these programs may download them from:

`http://cs.widener.edu/proofcheck`

References

[1] R.A. Alps. *A Translation Algorithm for Morse Systems*, PhD dissertation, Northwestern University, 1979.

[2] R.A. Alps and R.C. Neveln, A Predicate Logic Based on Indefinite Description and Two Notions of Identity. *Notre Dame Journal of Formal Logic* 22(3) 1981.

[3] W.W. Bledsoe and E.J. Gilbert. *Automatic Theorem Proof-Checking in Set Theory*, Sandia Laboratories Research Report SC-RR-67-525, July 1967.

[4] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67:203–260, 1989.

[5] A.P. Morse. *A Theory of Sets* Second Edition. Academic Press 1984.

[6] R.C. Neveln. *Basic Theory of Morse Languages*, PhD dissertation, Northwestern University, 1975.

[7] Bob Neveln and Bob Alps. *Foundations of the Topology of Manifolds*, (book in preparation).

[8] William Richter. TEX and Scripting Languages. *TUGboat*, 25(1), 2004.