Writing Literate Proofs with XML Tools

Ruben Gamboa
Computer Science Department
University of Wyoming
ruben@cs.uwyo.edu

Abstract

Both mechanical proof scripts and programs benefit from documentation and modularity. Designed to be an industrial-strength theorem prover, ACL2 provides powerful support for both of these. The standard distribution includes hundreds of pages of documentation that are automatically generated from function doc-strings in the source code. However, it is still difficult to write easily understandable proof scripts. We need a way to communicate proof scripts to a future human reader, not just to a mechanical theorem prover. Literate programming offers a solution to the analogous problem for computer programs. We describe how the same techniques can be applied to proof scripts.

1 Introduction

From our experience, few users find the process of reading a proof script simple. A large part of this difficulty is probably due to the essence of proofs: proofs can be complex even without the help of computers. But some of the difficulty is accidental, due not to an inherent feature of the proof, but to a limitation of our current processes and tools.

Rather than presenting ideas in a way to make them easy to understand, a proof script introduces ideas in a way that makes it easy for a computer to verify. For instance, all functions must be defined before their use, even when their only purpose is to construct an object used inside a proof. While the proof itself may be hard to understand in essence, its presentation in a proof script, intended as it is for a mechanical as opposed to a human mind, adds considerable accidental complexity.

A similar situation faced early Pascal programmers, prompting Knuth to invent literate programming [5]. His idea is that programmers should work on making the program understandable to other programmers, not computers. A literate program is a mixture ("web" in Knuth's pre-www terminology) of documentation and source code, and literate programming tools extract these automatically from the mixture.

Literate programming has many adherents, but it failed to become mainstream programming technology. Partly, we believe, this is due to many programmers' aversion to writing documentation in any form. But we in the mechanical theorem proving community do not have the luxury of avoiding documentation. For one thing, proofs are often much more complex than programs; many of us prove theorems *about* programs. For another thing, we have reached the point where we have many significant results available to us, so it is no longer feasible for groups to work in isolation. It is important for us to see and understand each other's work, not only at the high level as described in a conference paper, but also at the detailed level of the proof itself. And finally, we need to train more students in the use of mechanical theorem provers, and having a large number of understandable proofs will help to do so, especially if they are accessible in a variety of formats, such as technical reports, source code, and web pages. We feel that the time is right to write literate proofs.

This is not a new idea. ACL2 ships with voluminous documentation in both TEXinfo and HTML formats [4]. In the spirit of literate programming, this documentation is automatically extracted from the source code, where it appears as function doc-strings and in specially marked documentation nodes. In [3], Kaufmann suggests a way to organize ACL2 proof scripts in a way that makes them accessible to readers. His idea is to use ACL2's modular mechanisms to make the hierarchical structure of the proof clear. Of particular note is the fact that Kaufmann developed software tools to traverse this hierarchical structure.

In early experiments, we used standard literate programming tools to organize ACL2 proof scripts [6, 2]. In this paper, we depart from that approach and urge the use of XML as a literate programming platform. Partly this is due to a perceived lack of vitality in the world of literate programming tools, a lack that stands in contrast to the vibrant world of XML tools. Others in the literate programming and XML community have reached the same conclusion, possibly for other reasons [7, 1].

The remainder of this paper is organized as follows. In section 2, we introduce some basic XML tools and concepts. In section 3, we report on our experience writing literate proofs in XML. Finally, in section 4 we suggest ways to improve the process of writing literate proofs, and we explore other ways that XML technology can be used in working with mechanical theorem provers.

2 A Few Paragraphs About XML

XML is a format for writing structured documents in a machine-independent format, making it easier for programs to communicate. To make this practical, it is important that XML documents can be transformed from one format (or vocabulary) to another. Take for example, the following XML fragment:

```
<bibitem>
  <article id="KM97">
    <author>M. Kaufmann</author>
    <author>J. S. Moore</author>
```

```
    <title>An Industrial Strength Theorem Prover for a Logic
      Based on Common Lisp</title>
    <journal>IEEE Trans on Software Engineering</journal>
    <year>1997</year>
  </article>
</bibitem>
```

This XML fragment may be used as part of the bibliography of a research paper or to list the collective output of a research group on a web page.

In XML terminology, the file is processed by a stylesheet which specifies how to transform the document into another format. A stylesheet can convert an XML document into a simple ASCII file, e.g., a BibTeX or Lisp file. A file can also be transformed into another XML file, possibly using a different XML "vocabulary" (i.e., different tag or attribute names) such as HTML for web publishing or XSL-FO for printing. The XSLT standard defines XML stylesheets. In the interest of brevity, we will not discuss stylesheets in more detail. It is only important to note that stylesheets can specify which parts of the document to process and in what order.

The structure of XML documents is defined by a schema, which describes the valid tag and attribute names, as well as the way in which elements may be nested inside each other. An XML document can contain elements from multiple schemas, using XML namespaces to separate the vocabularies. For example, DocBook is a popular XML language for writing articles and books, and SVG is a standard for describing vector-based graphics images. Using namespaces, it is possible for a DocBook article to contain a graph described in SVG.

3 Writing Literate Proofs

Traditional literate programming tools define a language that combines both documentation and program in a single file, called a web file. This file can be translated into a format suitable for documentation, historically TeX. Separately, the program can be extracted from the web file.

In [7] Walsh describes an XML literate programming tool that takes advantage of XML namespaces. The idea is to add two primitives to an existing XML vocabulary. These primitives, `<src:fragment>` and `<src:fragref>`, allow code fragments and references to fragments to be included in an XML document. The containing document can use *any* XML vocabulary, including DocBook. An XML stylesheet converts the input file into the vocabulary of the enclosing document, essentially by specifying how `<src:fragment>` and `<src:fragref>` should be converted and leaving all other elements the same. A second XML stylesheet extracts the `<src:fragment>` elements to produce the program source code.

This is precisely the approach we chose to write literate proofs. By choosing DocBook as our documentation language we can easily present literate proofs as PDF files and as HTML web pages. Some examples can be found in [2].

The structure of the proof is described in a `<src:fragment>` called "top":

```
<src:fragment id="top">
  (in-package "ACL2")
  (include-book "../data-structures/list-theory")
  (include-book "../data-structures/alist-theory")
  <src:fragref linkend="Definitions of satisfaction">
  <src:fragref linkend="Supporting definitions">
  <src:fragref linkend="Definition of gensat">
  <src:fragref linkend="Soundness lemma">
</src:fragment>
```

This can appear *anywhere* in the document, but we usually include this in the introduction, where the structure of the report is described.

The remainder of the report contains the fragments referred to above, as well as any other fragments. These can appear in *any* order. The order we choose reflects the way in which we want to *explain* the proof to a fellow human being. We continued the theory of gensat, an approximation algorithm to SAT, by introducing the top-level structure of the algorithm up-front[1]:

```
<src:fragment id="Definition of gensat">
  (defun gensat (clauses max-tries max-flips)
    ...)
</src:fragment>
```

In contrast, the equivalent ACL2 proof script starts with the definition of `truth-value`, which looks up the boolean value of a proposition in a truth assignment. Obviously such a function is necessary, but its details are largely irrelevant when trying to understand the correctness of gensat, which would be the focus of a human reader of this proof.

It should also be noted that not all ACL2 events need appear in the report. Many auxiliary functions or lemmas could be relegated to an appendix, or simply omitted from the report.

4 Conclusion

We believe there is a need for a convenient way to present proof scripts, both as printed reports and on the web. We have shown that it is possible to do this by using tools developed for literate programming, and we have argued that tools based on XML work well in this context.

Some may worry that writing literate proofs is much more difficult than writing regular proof scripts. In our experience, this is not the case. We still follow the emacs-based cut/paste approach to proof construction familiar to many ACL2 users. The only times that writing literate proofs has proved more difficult than writing traditional proof scripts have been when we are engaged in (rarely successful) fishing expeditions with the theorem prover.

[1] We omit the definition of gensat here, since it is of limited relevance to this paper. Interested readers can consult [2].

As the example in section 3 shows, we have chosen to write proof scripts in ACL2's native syntax. We believe this is a natural approach, since ACL2's syntax will be familiar to both writers and readers of proof scripts. However, an alternative would be to use a pure XML syntax for ACL2 expressions, or even a mixture of ACL2 expressions with XML ones. This offers some intriguing possibilities. Consider, for example, a `<defun>` XML element. A stylesheet could be used to transform this into ACL2's syntax for the proof script. But a different style sheet could render the definition in a completely different format for the report. For example, it could use Nqthm-style definitions instead of the `defun` form. Or it could use an infix notation for expressions. The same applies to fragments. The first ACL2 workshop established the convention that hints to the theorem prover need not be included in the report, even though it is important the reader be advised that some hint was required. This can be handled automatically by an XML stylesheet.

We have not explored other ways to use XML tools with ACL2. For example, ACL2 renders the tag tree of a proof as a list of theorems used in the proof. If the tag tree were to be rendered in XML instead (or as well), it could be used as input to a tool that graphs the dependency between theorems. While similar tools have been written by ACL2 users in the past, leveraging the existing and growing XML technology allows us to focus on the business of writing proofs, for humans as well as for the theorem prover.

References

[1] R. Cover. The XML cover pages: Literate programming with SGML and XML. `http://xml.coverpages.org/xmlLitProg.html`.

[2] R. Gamboa. Literate theorem proving. `http://www.cs.uwyo.edu/~ruben/projects/litproofs`.

[3] M. Kaufmann. Modular proof: The fundamental theorem of calculus. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 6. Kluwer Academic Press, 2000.

[4] M. Kaufmann and J S. Moore. ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual. Available on the world-wide web at `http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html`.

[5] D. E. Knuth. Literate programming. *The Computer Journal*, May 1984.

[6] N. Ramsey. Noweb — a simple, extensible tool for literate programming. `http://www.eecs.harvard.edu/~nr/noweb`.

[7] N. Walsh. Literate programming in XML. In *Proceedings of XML 2002*, December 2002.