Literate Sources for Content Dictionaries

Lars Hellström

Department of Mathematics and Mathematical Statistics, Umeå University, Umeå, Sweden;
lars.hellstrom@residenset.net

Abstract

It is suggested that a LaTeX document could be used as the Literate Programming source of an OpenMath content dictionary. Several approaches to literate programming are reviewed and a possible implementation is sketched.

1 Introduction and Motivation

Historically, one of the key features of OpenMath has been the use of *content dictionaries* for defining the symbols that may appear in formalised mathematical formulae. This allows OpenMath to be relevant for arbitrary mathematics (as opposed to just the K–14 segment of mathematics that is often the primary target of computer mathematics projects), but in order to *be* relevant it is necessary that interested mathematicians can find or produce content dictionaries that are appropriate for their work. It is not realistic that these could mostly be produced by the OpenMath society—there is simply too much mathematics out there, and in addition not very many people available working on OpenMath—so the remaining possibility is that many mathematicians who wish to employ OpenMath will have to write up some content dictionaries of their own. But can they do that? They will face a number of obstacles.

It's not just the `.ocd` file, since some of the information that may reasonably be thought of as a natural part of "defining a mathematical symbol" should instead be placed in separate files. Type definitions go into `.sts` files, and notation has to be supplied from a third (currently not standardised) source. This division is logical from a tool perspective—many tools don't care about the information in the additional files—but it is an extra complication for the author, especially when experimenting and being creative, since the separate files need to be kept in sync.

A content dictionary file is XML-with-namespaces, which is not something with which the typical mathematician is familiar. XML is often criticised for being too verbose (as a consequence of its SGML ancestry), but this need not be that much of a deterrent here, since the main experience of verbosity is likely to come already when writing mathematical formulae; if you're gotten as far as writing a content dictionary, then you've probably already accepted the `<OMA>...</OMA>` as part of the game. Moreover, it must have been obvious from the start that some manner of formal encoding was going to be employed, and among the formal languages that could have been chosen, XML has an unusually good chance of looking familiar since even mathematicians get exposed to HTML at times.

The problem lies instead with the namespaces, which can pretty much be ignored when writing OMOBJs, but not when writing CDs since the latter mixes tags from `http://www.openmath.org/OpenMath` and `http://www.openmath.org/OpenMathCD`. Writing a valid content dictionary requires getting the namespace markup right, but the rules for this are not obvious (unlike those for XML-sans-namespaces elements, which one can pretty much intuit from looking at some examples), so an author extrapolating from examples is bound to get

confused by differences in choice of namespace prefixes—both in that two code fragments can look the same, but aren't, and in that they can look different, but are in fact equivalent. Mere examples aren't enough to get around this hurdle.

Another issue with XML is that there is quite a technological complex constructed around the basic specification, and ordinary mathematicians coming to OpenMath cannot be expected to have seen much of this beyond the occasional XML document. They will need some form of quick orientation to this, perhaps particularly to XSLT and the process of validation, but that is beside the main point of the present paper.

There is more that can be said than what fits. In my own humble attempts at creating content dictionaries (several of which are unfinished), I have often found myself writing far more FMPs and CMPs per symbol than there are in the official content dictionaries. Upon reflection, I've realised that some of the things I wrote were perhaps not exactly about defining/identifying (the semantics of) the symbol in question, but more about recording some thought that played a role in crafting the definition details.

This last point eventually led me to think about Literate Programming [8] as a paradigm that might apply to writing content dictionaries, since one way of looking at literate programming is that it merely gives programmers the opportunity to *record*, in a reasonably organised manner, the nontrivial thoughts on how their program works that they should anyway form while creating it. If there was a literate layer on top of the XML document formally defining a content dictionary—if there was a literate source from which the `.ocd` files were generated—then I could use that literate layer to expound upon why things are defined the way they are and prove (if necessary) that the stated properties suffice for uniquely characterising the symbols being defined, allowing the XML document to focus on the essentials of the definitions. It is true that [2] recommends "Every symbol defined in a CD should have at least one example", "FMPs should be as comprehensive as reasonable", and "If an FMP is given, then the equivalent (English) CMP should also be given", but one can expect that the typical content dictionary reader is primarily concerned with how to *use* the symbols as defined, and would be less than well served by having to skip past material on how a symbol might alternatively have been defined. The `.ocd` file is perhaps best imagined as the reference card, whereas the literate source file would be the full report.

Less personally, one might observe that the creation of new content dictionaries is likely to become an end-user development effort. Quoting [3]:

> The argument for literate programming for end-user developers, especially in knowledge-intensive problem-solving domains, is that beyond simply solving the problem, the domain expert wants to share the solution with others. In effect, the expert is writing a description of the solution and to the extent that this is also an executable (perhaps even tailorable) representation of the solution, it becomes a compelling vehicle for sharing and reusing the artifact.

It should be remarked that the kind of 'end-user' imagined in [3] is somewhat different from the one (professional mathematicians) at hand here, so there is little reason to believe that the optimal form of the literate programming that is sought here need to be as imagined in that paper. But what forms are there?

2 Some Approaches to Literate Programming

It could be argued that the ancestor of literate programming is the printed book of a program/code library, where (mostly short) passages of code are interlaced with paragraphs of ordinary text explaining how they work; a cute example of this, that I've come across for other reasons, is [17]. A catch with this kind of book is however that they usually wouldn't be machine-readable source even if given as a searchable digital file; the expectation is at best that a diligent human reader could stitch together the given code fragments into a working program—and often even that various uninteresting but necessary (for example input and output) blanks should be filled in as appropriate—not that there would necessarily be a reliable *automated* procedure for extracting a working program from the book. These texts are then literate, but not fully programming.

At the other end, it should be mentioned that documentation generators that work with texts embedded into code—for example Javadoc, Doxygen, and ROBODoc—are usually not considered literate programming systems, on the grounds that any narrative there is in the documentation will be subordinate to the formal structures of the programming language. They do therefore not allow the author to create literature.

2.1 Web-style literate programming

Web [8] is Knuth's original literate programming system, created to facilitate the coding of TEX and MetaFont before the literate programming concept was coined. It inspired the creation of a number of similar systems (for example CWeb, FWeb, FunnelWeb, Noweb, Nuweb, and Spider), and the vast majority of the Computer Science literature on literate programming [1] is about Web or one of its followers, in form or in style.

Web consists of the two programs TANGLE and WEAVE. The former is a preprocessor with rather extensive, but specialised, code transformation features, most of which were motivated by limitations in the Pascal in which TEX and MetaFont are implemented. Of lasting interest is primarily the concept of *modules*, which are named code fragments that the programmer can introduce in an order quite independent of that in which they will be presented to the compiler; originally this was to overcome a restriction that for example all type definitions would have to be made before the first subroutine definition, but it turned out to be useful also to stub out parts of subroutines so that the overall structure would appear more clearly. WEAVE is a prettyprinter with some indexing features, providing powerful mathematical typography in documentation sections as the material there is mostly handed over as-is to TEX for the typesetting step.

Because the original Web actually parsed the code fragments, it would only be applicable to Pascal programs. One trend among the followers has been to port it to other languages, and another trend has been to go more language-independent. A later trend has been to move away from text file as source format, and instead employ opaque formats that require specialised editors. Success has, on the whole, been limited. For the case of literate content dictionaries, it is doubtful whether there is anything to salvage from Web; its two main competencies of prettyprinting and code fragment rearrangement are pretty much irrelevant for an application to content dictionaries.

2.2 The doc/docstrip style of literate programming

The by far most successful form of literate programming is instead the doc/docstrip system used for the LATEX 2_ε kernel and a vast number of ditto packages. doc [15] began as a LATEX package

for using LaTeX markup in the *comments* of a `.sty` file, by typesetting these and having code lines wrapped up in a `verbatim`-like environment. As the amount of commentary then grew, the docstrip utility was created as a means of stripping away the comment lines from production `.sty` files. The unstripped sources then began to carry the `.dtx` extension, and it was discovered that they could contain the "driver" code needed to direct typesetting, so that today the doc equivalent of "weaving" is to '`latex` *`something`*`.dtx`'.

Since docstrip is supposed to copy the code lines from the `.dtx` source to a generated file, it is possible to use it for arbitrary programming languages. The docstrip module mechanism also provides conditional code inclusion and the ability to generate multiple files from the same source; the latter is particularly useful for projects employing multiple programming languages. A weakness of doc is that it offers no facilities for marking up code that is not TeX, but xdoc2 [5] reimplements those parts of doc in such a way that additional markup commands and environments are easy to define, as demonstrated by tclldoc [4].

Since a `.dtx` file can be an arbitrary LaTeX document, the doc/docstrip style of literate programming effectively equips the 'book on a program' ideal with an automated extraction procedure. It fully supports having a literate layer on top of the base XML of a content dictionary. It also supports (through a judicious use of modules) keeping the corresponding parts of `.ocd`, `.sts`, and notation files together in the source, although doing that is not at the beginner difficulty level. It provides no assistance with the XML namespace issue.

2.3 The document as code

A third approach to literate programming is to unify code and narrative, by making both aspects of the same document; that one aspect becomes the code and another typeset documentation is a matter of making different interpretations, not a matter of syntax. At least one experimental such system [16] has been described in the CS literature, but here it should be more instructive to consider a mature system. As it happens, the infrastructure used to typeset the present paper actually exercises also this third approach to literate programming, in the form of Fontinst encoding and metric files, although the literate aspect of these are somewhat of an afterthought.

Fontinst [7] can be described as a (mis)use of TeX to serve the role of general-purpose scripting language, in the task of converting industry-standard font metrics to something suitable for TeX; the typical state of operation is that data are being read from an external file by `\input`ting (thus executing) it, which results in data being written to zero or more other files, and macros being redefined, but no typesetting. Among the files being read are the encoding files [6], which are primarily expressing a data structure, even if they are often treated more like imperative sequences of commands. There is however also the literate aspect that they may be typeset as ordinary LaTeX documents, in which case the encoding commands produce English phrasings of the expressed data and additional "comment" commands may contribute narrative to tie it all together.

Traditionally, Fontinst encoding files have been fairly rigid as texts, but could be made far less so if the condition that processing as code should typeset nothing was lifted.

2.4 Miscellanea

Upon review, it was pointed out that [13] is likely a another relevant reference point for this work. Even from a cursory glance at it I'd agree, but I have not had time to consider it in depth before the conference.

```
\begin{OpenMathCD}{set1}
  \OMCDlicense % To generate half a page of legalese.
  \CDDate{2004-03-30}
  \CDStatus{official}
  \CDVersion{3}{0} % version, revision
  \CDDescription{
    This CD defines the set functions and constructors for basic
    set theory. It is intended to be 'compatible' with the
    corresponding elements in MathML.
  }
  \begin{CDDefinition}{cartesian_product}
    \Description{
      This symbol represents an n-ary construction function for
      constructing the Cartesian product of sets. It takes n set
      arguments in order to construct their Cartesian product.
    }
    \CDRoleApplication
    \NotationNassocBinop{50}{\times}{\UnicodeChar{00D7}}
      % Priority, LaTeX command, character (for MathML)
    \begin{STSSignature}
      ⋮
    \end{STSSignature}
    \begin{CDExample}
      ⋮
    \end{CDExample}
  \end{CDDefinition}
  ⋮
\end{OpenMathCD}
```

Figure 1: A mock-up of source for a content dictionary

3 Sketch of an implementation

The most promising approach seems to be the third one: make the source defining a content dictionary a LaTeX document, and include in it certain commands and environments that have as side-effect to output appropriate material to the corresponding `.ocd`, `.sts`, and notation files. To the novice user, this would look like an ordinary LaTeX document that however uses some fairly structured markup for stating OpenMath symbol definitions. A sketch of what such a document could look like, minus all literate narrative, can be found in Figure 1. It is natural that block structures in the content dictionary—such as the CDDefinition as a whole, individual symbol definitions, FMPs, CMPs, etc.—should translate to LaTeX environments, whereas more simple items such as the CD date are well cared for already by simple LaTeX commands.

3.1 Reviewing the Basic Design Choice

There are no doubt those who instinctively feel that source files should be some manner of well-defined XML rather than a quirky jumble such as LaTeX. Realistically though, any approach that requires authors to compose what amounts to a minor math paper in XML is going to alienate a huge share of the target demographic of mathematicians in general. A response could be that humans are not expected to edit the XML themselves; instead they should use a WYSIWYG editor to create the wanted XML document. But if that was a satisfactory alternative, then why do mathematicians use LaTeX rather than Word for writing papers? WYSIWYG might not alienate quite as large a fraction of the demographic as raw XML would, but it would still alienate far too many.

A related, but separate, concern could be that the source format should provide for structured representation of mathematical theories, like OMDoc [11] does; indeed, handling content dictionaries has been presented as an application of OMDoc [10]. There are two reasons not to do this, at least not for the foreseeable future. One is that formalising the structure of the narrative goes against the spirit of Literate Programming, moving more towards embedded documentation. It is not given that it goes so far as to actually give up on being literate, but it is a risk one must consider. Second, OMDoc is about formalising mathematics, whereas the act of creating a content dictionary is arguably *metamathematics*, since it defines (a piece of) the language for one's mathematical theories. We may know fairly well how to formalise mathematics, but that is not necessarily the same as formalising metamathematics; at the very least, wild analogies (such as between multiplication of numbers and application of functions) can be perfectly good metamathematics and inspire very fruitful choices of notation even though there is (at least initially) no clear mathematical foundation for making that analogy. (Cf. the analysis in [18] of the role that notation has played in the development of 20th century physics.) Stressing formalisation could therefore be unnecessarily limiting. On the other hand, there is nothing in the basic design which would prevent authors from using packages such as sTeX [9] for formalising the literatary parts of the sources, should they prefer to do so. But an outright *requirement* to do so would again be likely to alienate many prospective users.

3.2 Steps towards Implementation

The first technical problem, which is perhaps also the largest, that one faces when asking TeX to output XML is to produce well-formed character data. LaTeX syntax follows different rules than XML syntax, and whatever is presented to the user must look reasonably consistent, so the user should not have to manually supply XML entity markup for characters that happen to be special to XML. A solution to this problem is however halfway implemented; basically it elaborates on the harmless character strings mechanism the author created for `xdoc2` [5], updated to cope with the full Unicode character set, and with XML among the supported target formats. On top of that, it is not too difficult to implement commands for outputting arbitrary (up to equivalence and whitespace normalisation) XML fragments.

With this in place, it becomes clear that a LaTeX document can be used as source for `.ocd`, `.sts`, and notation files, so what remains is to make this reasonably convenient by providing suitable higher-level commands. The typical way to output an `OMS` element might for example be to use a command

`\OMS[`⟨*cdbase*⟩`]{`⟨*cd*⟩`}{`⟨*name*⟩`}`

and an `FMP` environment could generate not only the `FMP` element but also the `OMOBJ` element it must contain, and any namespace declarations that are needed.

An interesting point is what the commands for OpenMath objects should typeset. A perfectly serviceable approach is to have them typeset the same XML code as is being output; in the examples and FMPs of a content dictionary, fine details in structure and encoding may well be of great interest to the reader. It *could* be feasible to alternatively present objects by typesetting their PopCorn encoding, but the implementation (remember that this would be done by TEX macros) might become nontrivial. Typesetting as normal mathematical formulae is likely to be unreasonably difficult, and probably not even desirable.

High-level commands for writing OpenMath objects could also bring about a dramatic simplification of certain common coding tasks. FMPs often have an outermost layer stating 'For all a, b, ... in set S, it holds that ...' Conceptually, this is one thing, but in OpenMath it has to be encoded as

```
<OMBIND>
  <OMS cd="quant1" name="forall"/>
  <OMBVAR> <OMV name="a"/> <OMV name="b"/> ... </OMBVAR>
  <OMA>
    <OMS cd="logic1" name="implies"/>
    <OMA>
      <OMS cd="logic1" name="and"/>
      <OMA> <OMS cd="set1" name="in"/> <OMV name="a"/> S </OMA>
      <OMA> <OMS cd="set1" name="in"/> <OMV name="b"/> S </OMA>
      ⋮
    </OMA>
    ⟨body of formula⟩
  </OMA>
</OMBIND>
```

This is an awkward amount of boilerplate code, and it would be much easier on authors if they in the source could instead simply say

```
\begin{forallin}{a,b,...}{
  ⟨the set S⟩
}
  ⟨body of formula⟩
\end{forallin}
```

Another nice thing about using high-level commands for generating the raw XML is that they can target several formats simultaneously. For notations, there is no established standard, with at least two `.ntn` formats [12, 14] having been proposed, but the presentation in the content dictionary collection at `www.openmath.org` rather relying on explicit XSLT. For the latter, LATEX would even serve as something of a compiler (reading high-level descriptions such as 'n-associate binop' and generating low-level XSLT to make it a reality), even though one should probably not expect it to be capable of handling unusual presentation forms; those who want to create unusual effects will have to supply the details themselves.

One question that remains is how to name things. The XML encoding of content dictionaries favours CamelCase, whereas the LATEX tradition is rather lower case in situations like this; some of the CamelCase in Figure 1 looks uncalled-for, although there is a merit in using the same names as in the XML encoding. There are probably other choices of a similar character still left to identify and make.

Acknowledgements

Thanks to Christoph Lange, Paul Libbrecht, and others on the OpenMath mailing list for help with references and explaining the situation of the varying notation systems.

References

[1] Nelson H. F. Beebe. *A Bibliography of Literate Programming* (version of 11 April 2012). `http://www.math.utah.edu/pub/tex/bib/litprog.html`

[2] James Davenport. *On Writing OpenMath Content Dictionaries*. 2002. `http://www.openmath.org/documents/writingCDs.pdf`

[3] Matthew Dinmore and Anthony F. Norcio. Literacy for the Masses: Integrating Software and Knowledge Reuse for End-User Developers Through Literate Programming. In *Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on* (pp. 455–460).

[4] Lars Hellström. *The tclldoc package and class*. LaTeX macro package, 2003. `http://ctan.org/pkg/tclldoc`

[5] Lars Hellström. *The xdoc package*. LaTeX macro package, 2003. `http://ctan.org/pkg/xdoc`

[6] Lars Hellström. Writing ETX format font encoding specifications. *TUGboat* **28**:2 (2007), 186–197.

[7] Alan Jeffrey, Sebastian Rahtz, Ulrik Vieth, and Lars Hellström. *The fontinst utility*. TeX macro package, 1993–2009. `http://ctan.org/pkg/fontinst`

[8] Donald E. Knuth. Literate Programming. *The Computer Journal*, vol. 27, no. 2 (May 1984), 97–111.

[9] Andrea Kohlhase, Michael Kohlhase, and Christoph Lange. sTeX – A System for Flexible Formalization of Linked Data. Article 4 in: *Proceedings of the 6th International Conference on Semantic Systems (I-Semantics) and the 5th International Conference on Pragmatic Web*, ACM, 2010. doi:10.1145/1839707.1839712

[10] Michael Kohlhase. OMDoc: An Infrastructure for OpenMath Content Dictionary Information. *SIGSAM Bulletin* **34** (2) (2000), 43–48.

[11] Michael Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Springer, 2006. `http://omdoc.org/pubs/omdoc1.2.pdf`

[12] Michael Kohlhase, Christine Müller, and Florian Rabe. Notations for Living Mathematical Documents. Pp. 504–519 in: *Intelligent Computer Mathematics*, Lecture Notes in Computer Science **5144**, Springer, 2008.

[13] Achim Mahnke and Bernd Krieg-Brückner. Literate Ontology Development. Pp. 753–757 in: *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, Lecture Notes in Computer Science **3292**, Springer, 2004.

[14] Shahid Manzoor, Paul Libbrecht, Carsten Ullrich, and Erica Melis. Authoring Presentation for OpenMath. Pp. 33–48 in: *Mathematical Knowledge Management, MKM'05*, Lecture Notes in Computer Science **3863**, Springer, 2006.

[15] Frank Mittelbach. The `doc`–option. *TUGboat* **10**:2 (1989), 186–197. An updated version of this paper is part of `doc.dtx` in the base LaTeX distribution.

[16] James Dean Palmer and Eddie Hillenbrand. Reimagining literate programming. In: *OOPSLA '09* (ISBN 978-1-60558-768-4), 1007–1014. doi:10.1145/1639950.1640072

[17] David E. Rydeheard and Rodney M. Burstall. *Computational category theory*. Prentice Hall, New York, 1988. ISBN 0-13-162736-8. Also available for download from the author's homepage.

[18] Mark Steiner. *The applicability of mathematics as a philosophical problem*. Harvard University Press, 1998. ISBN 0-674-00970-3.