Rendering Mathematics for the Web using Madoko

Daan Leijen
Microsoft Research
daan@microsoft.com

1. INTRODUCTION

Madoko [6–8] is a novel authoring system for writing complex documents. It is especially well suited for complex academic or industrial documents, like scientific articles, reference manuals, or math-heavy presentations. One particular important aspect of Madoko is to write a document in high-level Markdown [5] with a focus on semantic content. From this document specification we can generate both high-quality PDF output (via LaTeX) but also generate high-quality HTML that can re-scale and re-flow dynamically. Styling is done through standard CSS attributes and can be done orthogonal to the content.

Madoko provides extensive support for mathematics rendering. All math is rendered using LaTeX with full compatability with any LaTeX packages and commands. Rendering to PDF comes this way for free but a high quality rendering of the math in the resulting HTML is more involved. This application note article describes in detail how Madoko deals with the various technical challenges. Moreover we show how other mechanisms, like replacement rules, help with creating mini domain-specific extensions to cleanly express complex math.

Since this article is about the rendering of math to HTML, it is highly recommended to read this article as an HTML page instead of PDF! It can be found at `http://tinyurl.com/madokomath`.

2. AN OVERVIEW OF MADOKO

Madoko is based on *Markdown* [5] as its input format. The main design goal is to enable light-weight creation of high-quality scholarly and technical documents for the web and print, while maintaining John Gruber's Markdown philosophy of simplicity and focus on plain text readability. Since the Markdown input format is well-structured, this allows Madoko to generate both high quality HTML *and* PDF (through LaTeX and BibTeX). There has been a lot of effort in Madoko to make the LaTeX generation robust and customizable while integrating well with the various academic document- and bibliography styles.

DocEng '16, September 12-16, 2016, Vienna, Austria
 ISBN 978-1-4503-4438-8/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2960811.2967168

On modern devices like tablets and phones it is generally much more pleasant to read a paper or technical document as HTML instead of PDF since HTML can scale and reflow dynamically. A study by Franze et al. [2] showed that the most desired features when reading papers is being able to change the font size, alter margins, or have a single column layout; all of these are trivial in a web browser. Of course, this article itself was written in Madoko, and the HTML version can be viewed at `http://tinyurl.com/madokomath`. Others have tried to create re-scalable and re-flowable content from paginated PDF [11], or the other way around, paginating dynamic content [3], but we believe starting from a more high-level structured input format is a better way of approaching this problem.

The move to Markdown makes the the documents *structured*, *readable*, and output *independent*. The final ingredients that Madoko adds are to make the documents *styleable* through standard CSS rules, and *programmable* through transformation rules. These additions also makes it easy to add custom domain specific document elements, like *exercise* or *answer*, that can be transformed, numbered, and styled in a declarative manner.

Finally, the online version at madoko.net integrates seamlessly with Dropbox, GitHub, and OneDrive, making documents available anywhere on any device. Madoko synchronizes automatically and multiple authors can work concurrently on the same document using robust three-way merges on concurrent updates. This means that updates by others are not quite real-time as in other collaborative environments (although they are performed frequently), but anyone can now work off-line and still reliably merge when connecting again. Madoko.net is itself a HTML5 web application and the editor continues to work in the browser even when offline. Of course, you can always use the plain command line version of Madoko locally (`npm install -g madoko`).

3. SCALABLE MATH ON THE WEB

Madoko uses regular LaTeX for describing math formulas since TeX is still the gold standard for rendering and describing mathematics. Any formula can be directly embedded in a Madoko document. For example:

```
A famous formula is $e^{i\pi} + 1 = 0$, but the
following one is also well-known:
~ Equation { #eq-gaussian }
\int_{-\infty}^\infty e^{-a x^2} d x
```

```
  = \sqrt{\frac{\pi}{a}}
~
⇝
```

A famous formula is $e^{i\pi} + 1 = 0$, but the following one is also well-known:

$$\int_{-\infty}^{\infty} e^{-ax^2} dx = \sqrt{\frac{\pi}{a}} \tag{1}$$

Here we use `$` to start inline math as in LaTeX. For the equation we used a so-called *custom block* of Madoko. The standard prelude of Madoko defines `~ equation` for numbered equations, `~math` for plain display math, and `~mathpre` for pre-formatted math discussed in Section 4. In the example, we also give the equation a name so we can refer to it using links in Markdown where `[#eq-gaussian]` expands to the equation number, e.g. Equation (1).

When creating PDF output, Madoko can simply include the literal formula in the generated LaTeX with full compatibility with any LaTeX package. Unfortunately, for HTML output the process is more involved as we need to render math seperate from the rest of the HTML. There exist various tools that use JavaScript to interpret LaTeX math commands directly and generate a rendering on the client. One of the most well-known libraries to do that is MathJax [1].

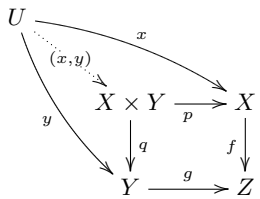
This works well for simple mathematics but one of the great advantages of using LaTeX for math is that it comes with many many packages to render advanced mathematics, or packages that render math in the style required by a journal. Such packages are generally not supported by tools like MathJax. Even though Madoko has an option to use MathJax for dynamic math rendering, the default mode is to invoke LaTeX and render all math at compile time.

In Madoko any LaTeX package can be used through a metadata key at the start of the document, for example:

```
Package: [curve]xypic
```

after which we can use the `\xymatrix` command to render category theory diagrams:

```
~ Math
\xymatrix @-0.5em{
U \ar@/_/[ddr]_y \ar@/^/[drr]^x \ar@{.>}[dr]|-{(x,y)} \\
& X \times Y \ar[d]^q \ar[r]_p & X \ar[d]_f \\
& Y \ar[r]^g & Z }
~
⇝
```



Generating good looking HTML from LaTeX rendered formulas is a challeging problem though and we describe here various solutions adopted by Madoko.

Hashing of formulas

When rendering a document, Madoko first collects all math formulas and assigns a unique MD5 hash to each individual formula. This ensures that each formula is only rendered once which is important since many short formulas are usually often repeated. Madoko generates a special LaTeX math file that contains 'snippet' entries for each formula. For example, for this document, one of the entries is:

```
%mdk-data-line={138}
\begin{mdInlineSnippet}[f2d2e607c3e99d5c34bc0aad01893a0d]
$e^{i\pi} + 1 = 0$%
\end{mdInlineSnippet}
```

The initial comment is how Madoko maps back LaTeX error messages to the correct line in the original Madoko file – this is very important in practice to quickly solve LaTeX problems. Next, the `mdInlineSnippet` command ensures that each formula gets rendered on its own page in the resulting DVI file. That DVI file is now passed to another tool to extract the rendering.

Madoko uses the excellent `dvisvgm` converter by Martin Gieseking [4] to convert LaTeX generated DVI files to *scalable vector graphics* (SVG) files. The `dvisvgm` converter automatically extracts an SVG file for each page in the DVI file, numbering them sequentially. Since Madoko maintains a mapping between the MD5 hashes and the page numbers, it can then automatically include the correct SVG images in the generated HTML for each formula.

Many tools extract math formulas as PNG images from a rendered PDF or Postscript file. Unfortunately, this is a non-scalable image and looks generally quite fuzzy on a screen especially for inline formulas surround by text. Figure 1 compares the rendering of Euler's identity in Wikipedia, which uses PNG images, versus the rendering in Madoko which uses SVG. The difference is quite stark and the quality of SVG rendering is excellent even compared to PDF – when demoing Madoko, often people are under the impression of viewing PDF while they are actually seeing the HTML rendering of a Madoko document in the browser.

it follows that

$$e^{i\pi} = -1 + 0i,$$

which yields Euler's identity:

$$e^{i\pi} + 1 = 0.$$

it follows that

$$e^{i\pi} = -1 + 0i,$$

which yields Euler's identity:

$$e^{i\pi} + 1 = 0.$$

Figure 1. Screenshots of different math renderings in the browser: the left image is rendered by Madoko using SVG graphics, while the right image is a rendering by Wikipedia using PNG images (https://en.wikipedia.org/wiki/Euler%27s_identity)

Baseline alignment

There are still various technical hurdles to overcome though. The most tricky one is proper baseline alignment. In particular, an inline formula should align as $\sum_{i=0}^{\infty} e^i$ with the e aligned with the text baseline. Contrast this with $\sum_{i=0}^{\infty} e^i$ for example where the bottom of the extracted image aligns with the baseline. There are often a lot of small inline formulas and not aligned well with the baseline looks very irregular to the eyes.

To achieve proper baseline alignment, we need to have an exact measurement of the *depth* of the formula, i.e. the bottom vertical distance to the baseline. If we know the depth, we can adjust the vertical alignment of the extracted image by lowering it by its depth.

The `mdInlineSnippet` environment does this by first rendering the formula in a TEX *box*. This box can be queried for its rendered height, width, and depth. After figuring out the dimensions the box is rendered to the page. For each formula, we write out the measured dimensions together with its hash (which is an argument to `mdInlineSnippet`) to a separate text file. After the LATEX run, Madoko reads this dimension file to determine the precise baseline alignment for each formula in the HTML.

The final height of the math image should be determined by the relative font size of the surrounding text used in the HTML. This means that the height and baseline adjustment must be made in font-relative `em` units instead of absolute units. Madoko renders mathematics in a `10pt` font size when taking measurements. For output to HTML we read the measurements from the dimension file (in `pt`) and divide by 10 to get the relative `em` units. We use the CSS `vertical-align` attribute to lower the math image by its measured depth. In practice we also scale the math image by 105% in order to look more natural with most web fonts. For example, our initial example, $\sum_{i=0}^{\infty} e^i$ is positioned in the HTML output as:

```
<svg style="vertical-align:-0.3502em;height:1.2355em"
     viewBox="88.467 53.397 33.929 11.767"
     class="math-inline math-render-svg math">
 <desc>$\sum_{i=0}^{\infty}e^i$</desc>
 <g id="page26">
  <use x="88.667" y="54.364" xlink:href="#g14-80"
    xmlns:xlink="http://www.w3.org/1999/xlink"></use>
  ...
 </g></svg>
```

The `<use>` element puts the glyph `#g14-80` (the Σ) at a specific position. That glyph is defined separately to enable sharing of graphical elements between different formulas.

Sharing glyph paths

Math heavy documents can easily contain thousands of formulas. Madoko already shares representations for equal formulas through hashing but more is needed. For example, in one example math-heavy article [10] the math formulas generate 2242kb of SVG images. It turns out though that many formulas contain similar glyphs, like e, or x. Each of these glyphs is (usually) rendered as a *path* in the SVG image. For example, the formula x is described in SVG as:

```
<defs>
  <path d="M3.328 -3.009C3.387 -3.268 3.616 -4.184 ...
    -0.986 2.879 -1.205 2.989 -1.644L3.328 -3.009Z"
    id="g6-120"></path></defs>
<g><use x="88.667" y="61.836" xlink:href="#g6-120"
     xmlns:xlink="http://www.w3.org/1999/xlink">
   </use></g>
```

Here we see how the image places the path element `#g6-120` at a specific position using an `xlink`. The path element though just traces a specific glyph, in this case the x, independently of its position. As the shapes are independent of the position, we can share all the glyph paths between different formulas. Madoko will collect all equivalent paths in a separate definition block and all formulas reference these shared paths. This can lead to significant space savings in practice – in the example article the space usage went down 79% from 2242kb to 467kb.

More significant space savings can be made by not describing glyphs with paths at all but using direct font entries and text elements in the SVG description. Unfortunately, font support in SVG is very spotty across browsers and most formulas do not render faithfully when using fonts directly[1]. Figure 2 shows two browser screenshots where one formula is rendered using traces while the other uses font elements.

$\int_{-\infty}^{\infty} e^{-ax^2}\,dx = \sqrt{\frac{\pi}{a}}$ vs. $\int_{-\infty}^{\infty} e^{-ax^2}\,dx = \sqrt{\frac{\pi}{a}}$

Figure 2. A browser screenshot of two math SVG images generated by Madoko. The left image used SVG path elements to trace glyphs, while the right image uses direct font elements.

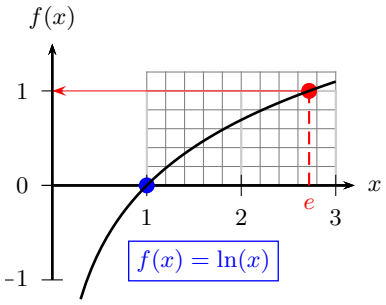


Figure 3. SVG image of a log graph rendered using the `pstricks` and `pst-plot` packages which emit Postscript specials.

Rendering of DVI specials

Some LATEX commands depend on specific output drivers. For example, the advanced TikZ package draws vector graphics using specific PDF primitives which are not directly supported in DVI files. In many cases, we can still extract correct SVG images from the DVI since `dvisvgm` supports many extensions. Ultimately, if that fails Madoko can also generate PNG files from a PDF or Postscript rendering although such image will no longer be scalable. However, currently even large packages like `pstricks` and `TikZ` work with DVI output so in practice this is almost never necessary. Figure 3 shows the SVG output of a log graph using the `pstricks` package which issues Postscript specials.

4. PRE-FORMATTED MATHEMATICS

Mathematics mode in TEX can be surprising in its handling of whitespace and identifiers. In general, whitespace in the text is not relevant and a sequence of letters is *not* seen as a single identifier. Look for example at the following formula:

```
$function sqr( x : int)$
```

$\rightsquigarrow$

$functionsqr(x : int)$

We can see that there is no whitespace between `function` and `sqr`, and how they are rendered as a sequence of letters instead of two identifiers; note in particular the whitespace between the `f` and `u` for example.

This behavior may be good for general mathematics, but in many fields, like computer science, this is often cumber-

[1]Madoko supports this option though through a metadata flag.

some to program with. To make programming with formulas more direct, Madoko offers a 'pre-formatted math' mode where:

- Whitespace is relevant where every space becomes a small math space (`\;`),
- An identifier is enclosed in a `\mathid` command so it gets rendered as *function* (instead of $function$).
- An identifier starting with `@` is enclosed in a `\mathkw` command and rendered as a keyword.
- Digits after an identifier are automatically subscripted where `x1` becomes x_1.
- Text that is an argument to `\begin`, `\end`, `\text`xx, or `\math`xx commands is kept unchanged.

Using these rules it becomes much easier to use more descriptive names and simple alignment. For example:

```
~ MathPre
@function sqr_\pi( x : int) \{
  @return (x\times x\times\pi);
\}
~
```

$\rightsquigarrow$

function sqr_π(x : int) {
  return $(x \times x \times \pi)$;
}

Domain specific math

As a final example we look combining existing Madoko features to math. In particular, Madoko extends CSS with a `replace` attribute where one can specify regular expression replacements over the content of a custom block. As an example, we will define a mini domain specific math language for defining natural deduction rules.

First, let's start with some simple replacement rules to make it easier to write type rules. For example, we would like to replace a capital `G` with Γ, or a plain `t` with τ. In Madoko we can simply use a CSS style rule with the special `replace` attribute:

```
.mathpre {
  replace: "/\bt\b/\tau{}/g";
  replace: "/\bG\b/\Gamma{}/g";
}
```

Here the `\b` specifies a word-boundary in the regular expression. The format of each replacer is */regex/replacement/*`g`, where the `g` flag specifies that the replacer should apply at every match in the content. Another flag is `i` which matches case-insensitive for example. See the reference manual for more information [9].

For natural deductions, we need a more complex regular expression. Let's assume we call our new custom block `~infer`, then we can define:

```
infer {
  replace:"/([\s\S]*?)\n *----+ *\[([^\]]*)\] *\n([\s\S]*)/\
           \\infer{\1}{\3}{\textsc{\2}}/m";
  replace:"~Begin MathPre {.infer}&nl;&nl;&source;&nl;\
           ~End MathPre";
}
```

The rule looks somewhat complicated but it is a straightforward regular expression where we first lazily match on any input `([\s\S]*?)` until we hit a horizontal line. Once that is matched, we pass the top and bottom part, together with an optional rule name as arguments to a new LaTeX command that we define ourselves to neatly typeset the inference rule. Combining these replacements (and adding some more), we can now write inference rules in a very natural style in pre-formatted math:

```
~ infer
G      & |- e1 : s | <>
G, x:s & |- e2 : t | eff
-----------------------------------[Let]
G |- @let x = e1 @in e2 : t | eff
~
```

$\rightsquigarrow$

$$\frac{\begin{array}{ll}\Gamma & \vdash\ e_1\ :\ \sigma \mid \langle\rangle \\ \Gamma,\ x:\sigma & \vdash\ e_2\ :\ \tau \mid \epsilon\end{array}}{\Gamma \vdash \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2\ :\ \tau \mid \epsilon}\ \ [\textsc{Let}]$$

Note how close the specification is to how one usually writes such rules in an email or while designing the rules – we can now concisely describe the rules while immediately rendering them nicely.

5. CONCLUSION

Try Madoko at madoko.net. Madoko is still a young project and any feedback is much appreciated. The author would like to thank Martin Gieseking for his help in making `dvisvgm` work well with Madoko.

REFERENCES

[1] Davide Cervone, Volker Sorge, Christian Perfect, and Peter Krautzberger. "MathJax: A Javascript Library for Rendering Mathematics." 2009. https://mathjax.org.

[2] Juliane Franze, Kim Marriott, and Michael Wybrow. "What Academics Want When Reading Digitally." In *DocEng '14*, 199–202. Fort Collins, CO. 2014.

[3] Fabio Giannetti. "Paginate Dynamic and Web Content." In *DocEng '11*, 143–152. Mountain View, California, USA. 2011.

[4] Martin Gieseking. "Dvisvgm: Converting DVI to SVG." 2005. http://dvisvgm.bplaced.net.

[5] John Gruber. "Markdown." 2004. http://daringfireball.net/projects/markdown.

[6] Daan Leijen. "Madoko: A Scholarly Markdown Processor." 2014. http://madoko.codeplex.com.

[7] Daan Leijen. "Madoko: Scholarly Markdown in the Cloud." In *SNAPL'15: The Inaugural Summit on Advances in Programming Languages*. May 2015. http://tinyurl.com/n6k3kht.

[8] Daan Leijen. "Madoko: Scholarly Documents for the Web." In *DocEng 2015, Lausanne, Switzerland*, 129–132. Sep. 2015. doi:10.1145/2682571.2797097. HTML available at: http://tinyurl.com/p4bm62o.

[9] Daan Leijen. "Madoko Reference Manual." http://research.microsoft.com/en-us/um/people/daan/madoko/doc/reference.html.

[10] "Madoko Sample Academic Article." 2015. http://tinyurl.com/madoko-effects.

[11] Simone Marinai. "Reflowing and Annotating Scientific Papers on eBook Readers." In *DocEng '13*, 241–244. Florence, Italy. 2013. doi:10.1145/2494266.2494311.