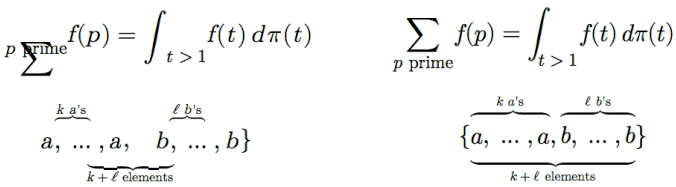
MathML Refactoring in WebKit

Jul 9, 2016

Introduction

If you follow WebKit developments, you are certainly aware that Igalia has been working on WebKit's MathML implementation for some time. More recently, effort has been made to write a clean implementation addressing issues reported by WebKit reviewers in the past. After joining Igalia in March, I have been in charge of getting this work reviewed and merged into WebKit's development branch. In the past four months, we have been successful in upstreaming the first phase of the refactoring and the work accomplished is described in this blog post.



MathML torture tests with the Latin Modern Math font

Left: Safari 9.1.1. Right: Current MathML branch.

Note that the focus was on code refactoring so the improvement may not be obvious for non-developers. Nevertheless many issues have already been fixed as a consequence of that work: math italic, position of scripts, stretchy and large operators, rendering update and more. More importantly, this preliminary step opens the way for beautiful math rendering based on TeX rules and the OpenType MATH table. Rendering improvements and implementation of new features have already started in the next phase of the refactoring, so stay tuned…

Design Issues

As explained in a previous report, the main design issues in the flexbox-based implementation released in 2013 can essentially be summarized in three points:

1. WebKit's code to stretch operator was not efficient at all and was limited to some basic fences buildable via Unicode characters.
2. WebKit's MathML code violated many layout invariants, making the code unreliable.
3. WebKit's MathML code relied heavily on the C++ renderer classes for flexboxes and had to manage too many anonymous renderers.

For the first point, the performance issue had been fixed by Igalia developers right after the initial feedback from WebKit developers and we improved that again during our refactoring. Partial support for the OpenType MATH table was added during my crowdfunding project and allowed to stretch more operators with the help of math fonts.

For the second point, the main issue was also fixed right after the initial feedback. However one could still have some doubts regarding the layout steps, given the complexity implied by the third point. That last issue was still problematic so far and addressing it was the main achievement of our refactoring.

Technically, the dependence on flexbox is unnecessary and the implementation actually only used a limited set of flexbox features. Thus executing the whole flexbox code was overkill. It can also be a burden for people working on other places of the layout code. For example Javi Fernández has worked on improving the box alignments in the past and he had hard time fixing the MathML code impacted by his changes. This is probably the cause of the bad position of the summation symbol that can be seen in the screenshot above.

From the layout perspective, most of the rendering logic was implemented in the flexbox classes and the MathML “renderer” classes were really just managing the creation and update of anonymous nodes and style. Although this sounds good code reuse it actually made impossible to understand how and when layout steps happen or to add more advanced features. The new implementation replaced this manipulation of the render tree with simple arithmetic calculations on box metrics which is safer and more reliable. Also, complex renderers such as `RenderMathMLScripts` or `RenderMathMLRoot` actually achieve better rendering quality with less code!

As an example of the complexity, `RenderMathMLUnderOver` can behave as a `RenderMathMLScripts` in some situation so we really want the former class to reuse the latter class. As we will see below the old implementation of the two renderers were quite different: `RenderMathMLUnderOver` only relied on setting column direction in the user agent stylesheet while `RenderMathMLScripts` created a complex render tree structures with anonymous style. Hence it seemed difficult to share the two cases or to handle DOM changes causing to move from one case to the other one. With our new implementation, this is simply reduced to simple C++ inheritance.

When I started to work on WebKit some years ago, I made the mistake of continuing with the existing approach. The implementation of multiscripts or automatic italic mathvariant added more anonymous objects and made the situation even worse. After the end of my crowdfunding project, Alex Castro did more cleanup and tried to implement important features such as displaystyle but he also soon realized that it was too hard to work with the current code base…

Layout Refactoring

In order to solve the issues discussed in the previous section, Javi and Alex worked on a new MathML branch where the first step was to remove the inheritance on the flexbox layout classes. During the Web Engines Hackfest 2015, I collaborated with the Igalia’s web platform team team to continue the work on this branch. In a second step, we rewrote many MathML renderer functions so that they stop creating anonymous nodes or style. We obtained very encouraging results: The implementation looked much simpler and much more understandable!

Alex announced the initial plan on the webkit-dev mailing list. He started opening bugs

and attaching patches to merge the first step. However, that step still required many of the flexbox logic and so made code hard to understand for reviewers. Hence when I joined Igalia four months ago Alex asked me to try and see how to reorganize patches so that the two initial steps can be submitted in one go. This corresponds to the first phase mentioned in the introduction. As indicated on the wiki page, the layout refactoring consisted in rewriting the following member functions of each renderer class:

- computePreferredLogicalWidths: calculate preferred widths, based on the preferred widths of child renderers.
- layoutBlock: set final position and size of child renderers.
- firstLineBaseLine: calculate the ascent of the renderer.
- paint (optional): perform special painting such as fraction bars.

Refactored renderers no longer rely on any flexbox code nor anonymous renderers and the functions mentioned above essentially perform arithmetic computations. By reading the code, one can be sure that we follow standard layout rules and that we do not perform unnecessary reflow. Moreover, the rules specific to math rendering are only located in the MathML renderers and can be better understood. Details for each class are provided in the next subsections. After all the layout functions were rewritten and the code managing the render tree structure removed, we were able to make the `RenderMathMLBlock` class inherit from `RenderBlock` instead of `RenderFlexibleBox`. Many of the bugs could then be immediately closed or otherwise fixed with small follow-up patches.

Spacing

`RenderMathMLSpace` is a simple class inserting blank boxes for adjusting spacing of math formulas. Obviously, we do not need any of the complexity of flexbox so it was straightforward to write the layout functions.

$$3 \quad x$$

Large space between 3 and x.

Grouping

`RenderMathMLRow` performs rendering of a row of math items. Since WebKit does not support linebreaking in MathML at the moment, this is just putting child boxes on a same baseline. One specificity is that some operators can be stretched vertically and so their width may depend on their height.

$$\left\{\frac{2}{x}\, x^3\right.$$

Row containing a stretched brace, a fraction and a scripted element.

Again, flexbox features are useless here. With the old code, it was not clear whether we were violating the CSS invariant with preferred and logical widths and which kind of relayout or render tree changes would happen when doing the stretch call. By properly implementing the layout functions previously mentioned all of this became much more trustable.

Fractions

`RenderMathMLFraction` draws a fraction with numerator and denominator.

Simple fraction.

This used to be implemented using a column direction for the fraction element. Numerator and denominator were wrapped into anonymous nodes with additional style to leave space for the fraction bar and to adjust the horizontal alignments.

```
RenderMathMLFraction (flexbox with column direction)
  RenderMathMLBlock (anonymous flexbox)
    RenderMathMLRow (numerator)
      ...
  RenderMathMLBlock (anonymous flexbox)
    RenderMathMLRow (denominator)
      ...
```

It was relatively easy to implement this without any anonymous nodes and again the use of flexbox did not sound justified. For example, to calculate the preferred width we just take the maximum of the preferred widths of the numerator and denominator. For the layout, the calculation of the logical width is similar and we calculate the horizontal coordinates of numerator and denominator so that their centers are aligned. Vertical metrics are similarly calculated from the vertical metrics of the numerator and denominator. During that step, we also fixed some bugs with the `linethickness` attribute and added support for some OpenType MATH table constants.

Scripts above and below

`RenderMathMLUnderOver` is used to attach some scripts above and below a base. Each child can itself be a horizontal stretchy operator.

Base with stretchy arrow over it.

This was implemented in the user agent stylesheet by using flexboxes with column direction for the corresponding MathML elements and the C++ class had additional rules to fire the stretching. So the problems and solutions for this class were essentially a mixed of the cases of `RenderMathMLFraction` and `RenderMathMLRow` we just discussed.

Subscripts and Superscripts

`RenderMathMLScripts` is used for a base with some arbitrary number of scripts. All the scripts can have different positions (pre, post, sub, super) and metrics (width, ascent and descent). We must avoid collisions and take care of horizontal and vertical alignements.

Base with pre and post scripts.

The old code used a complex render tree with additional style to achieve the best possible result. However, the quality was still bad as you can see for the script attached to the integral in the screenshot above. Managing the render tree was a nightmare: Just to give the idea, additional anonymous node and style were used to allow horizontal and vertical

adjustments (similar to `RenderMathMLFraction` above) and prescripts had negative order property so that they were positioned before the base.

```
RenderMathMLScripts
    Base Wrapper (anonymous flexbox)
      RenderMathMLRow (base)
        ...
    SubSupPair Wrapper (anonymous flexbox with column direction)
      RenderMathMLRow (post-subscript)
        ...
      RenderMathMLRow (subperscript)
        ...
    SubSupPair Wrapper (anonymous flexbox with column direction)
      RenderMathMLRow (post-subscript)
        ...
      RenderMathMLRow (post-superscript)
        ...
    ... (more postscripts)
    RenderMathMLBlock (prescripts separator)
    SubSupPair Wrapper (anonymous flexbox with column direction and order -1)
      RenderMathMLRow (pre-subscript)
        ...
      RenderMathMLRow (pre-subperscript)
        ...
    SubSupPair Wrapper (anonymous flexbox with column direction and order -1)
      RenderMathMLRow (pre-subscript)
        ...
      RenderMathMLRow (pre-superscript)
        ...
    ... (more prescripts)
```

Rules from TeX and the OpenType MATH table are relatively complex and we decided to implement them directly in the new refactoring as otherwise it was impossible to get decent quality. The code is still complex but we now have clear rules, we only perform simple calculations and the render tree structure matches the DOM tree.

“Enclosing” Notations

`RenderMathMLMenclose` is a row of math items with some additional notations. Gurpreet Kaur implemented this element two years ago but she followed the same approch, combining anonymous nodes and style for some simple notations and special painting for others.

circle and strike notations

During the refactoring, the code has been completely rewritten so that `RenderMathMLMenclose` is now essentially a derived class of `RenderMathMLRow` with the measuring and painting functions adjusted to take into account the additional notations. During that refactoring, we also removed support for unused radical notation, which was implemented using an anonymous `RenderMathMLSquareRoot` (see Radicals section below).

Helper Classes for Operators

The `RenderMathMLOperator` class is used for math operators. It was quite complex class

and we decided to extract from it two features that are unrelated to layout:

- The MathML operator dictionary and corresponding search functions have been moved into a `MathOperatorDictionary` class.
- Selection, measuring and drawing of stretchy operators have been moved into a `MathOperator` class. This is essentially the low-level text shaping being implemented in HarfBuzz.

The remaining code was indeed the real layout part but the mess with anonymous node and style was only removed later (see Text Classes below). Although it seems we just needed to move the code out of `RenderMathMLOperator` into those new classes, the case of `MathOperator` was particularly difficult. We had to split the effort into several small steps to make review possible and also fixed many issues due to the entanglement and confusion of these three different features of the `RenderMathMLOperator` class… The work done for `MathOperator` actually improved the rendering of stretchy operators as you can see for the horizontal braces in the screenshot above.

Radicals

`RenderMathMLRoot` is used for square root or arbitrary N-th root. Many of the TeX and OpenType MATH table rules were already used by the old implementation with anonymous nodes and style. However, there were bugs difficult to fix related to zooming, child removal or style change due to the management of the anonymous `RenderMathMLOperator` to draw the radical sign.

$$\sqrt{x+1}+\sqrt[3]{x+1}$$

square and cube roots

The old implementation actually had two classes for the square and general cases (`RenderMathMLSquareRoot` and `RenderMathMLRoot`). The usual technique with various anonymous wrappers and style was used. After the refactoring, we were able to merge everything in a single `RenderMathMLRoot` class. Because the square root behaves as an `mrow` , we also made that class derive from `RenderMathMLRow` to reuse as much code as possible. Here is are how the render trees used to look like:

```
RenderMathMLSquareRoot
  RenderMathMLBlock (anonymous used for metric adjustements)
    RenderMathMLRadicalOperator (anonymous used for the radical symbol)
      ...
  RenderMathMLRootWrapper (anonymous used for the children)
    RenderMathMLRow (child 1)
      ...
    RenderMathMLRow (child 2)
      ...
    ...
    RenderMathMLRow (child N)
      ...

RenderMathMLRoot
  RenderMathMLRootWrapper (anonymous for the index)
    ...
  RenderMathMLBlock (anonymous used for metric adjustements)
```

```
      RenderMathMLRadicalOperator (anonymous used for the radical symbol)
        ...
    RenderMathMLRootWrapper (anonymous for the base)
      ...
```

Again, we rewrote the implementation using only simple box positioning. The difficult part was to get rid of the anonymous `RenderMathMLRadicalOperator` to draw the radical symbol. This class was derived from `RenderMathMLOperator` and extended it with some fallback drawing when math fonts were not available. After having extracted stretchy operator shaping from `RenderMathMLOperator` it became possible to use the `MathOperator` helper class to draw the radical symbol. We implemented the fallback for missing math fonts the same as Gecko: Use a scale transform to stretch the base glyph for U+221A SQUARE ROOT. As a bonus, we used such transform to implement glyph mirroring, as required to draw right-to-left radicals in some Arabic mathematical notations.

Text Classes

These classes are containers for math text such as variables or operators. There is a generic `RenderMathMLToken` class and a derived class `RenderMathMLOperator` adding features specific to operators such as spacing, dictionary property, stretching… Anonymous wrappers and style were used to implement automatic italic mathvariant or operator spacing. The `RenderText` child of `RenderMathMLOperator` was (re)built as an anonymous text node so that is was possible to convert U+002D HYPHEN-MINUS into U+2212 MINUS SIGN or to provide some text for anonymous operators created by `RenderMathMLFenced` (see Unchanged Classes section).

```
RenderMathMLToken (e.g. mi element)
  RenderMathMLBlock (anonymous flexbox used to apply CSS italic)
    RenderBlock (anonymous created elsewhere to honor CSS rules)
      RenderText
        text run "x"

RenderMathMLOperator (mo element)
   RenderMathMLBlock (anonymous flexbox used for spacing)
    RenderBlock (anonymous created elsewhere to honor CSS rules)
      RenderText (anonymous destroyed and built again)
         text run "−"
```

We did a big refactoring to remove all the anonymous nodes created by the MathML renderer classes. Just like for `MathOperator` , we had to be careful and submit various small pieces as the text rendering was quite sensible to code change.

The simplified operator spacing that was supported by WebKit was easy to implement with the new approach. To do automatic italic mathvariant, we modified the paint function to use Mathematical Alphanumeric Symbols instead of CSS italic as you can notice for the variables displayed in the screenshot above. Hence we could remove the `RenderMathMLBlock` anonymous wrapper.

The use of an anonymous node for the text prevented it to appear in the dumped render tree of layout tests and also required some hacks in the accessibility code to expose that text. In order to address the cases of the minus sign and of mfenced operators, we

decided to use our new `MathOperator` class again. Indeed `MathOperator` is actually also able to draw unstretched operators made of a single character and this works for the minus sign and for mfenced operators used in practice.

Unchanged Classes

Two classes have not been modified but such modifications were not needed to remove the dependency on `RenderFlexibleBox` :

- `RenderMathMLFenced` is used for an mrow-like element that is defined in the MathML specification as strictly equivalent to constructions with rows and operators. It is implemented as a derived class of `RenderMathMLRow` and creates anonymous `RenderMathMLOperators` . This is the only remaining class that modifies the render tree structure. Note that prominent MathML websites and generators do not use the mfenced element, so it is not a big concern.

- `RenderMathMLTable` is used for table layout. It is just derived from `RenderTable` , not `RenderFlexibleBox` . We did not change anything for now but we considered creating our own implementation in order to make our code independent from HTML table, to support MathML-specific table features and to make it better integrated with the rest of the MathML code.

Accessibility

Even if our main focus was on rendering, the changes we made also had impact on the MathML accessibility code. Indeed, the accessibility tree is generated from the MathML renderer classes: Since we changed the latter during the refactoring, we also had to adjust the accessibility code. Fortunately, we are lucky to have Joanmarie Diggs in our team and she was able to provide some help here.

First, the accessibility code exposes the `linethickness` of fractions to implement Apple's `AXMathLineThickness` attribute. In practice, this is really necessary to know whether the `linethickness` is null or not (e.g. binomial coefficient VS the Legendre symbol). Apple's unit test seemed to expose the ratio between the actual thickness and the default thickness but the accessibility code really just reads the actual thickness calculated by `RenderMathMLFraction` . Our fix and improvement for `linethickness` made the Apple's unit test fail so we had to adjust `RenderMathMLFraction` to expose the value expected by that test.

In general, the accessibility code does not care about anonymous nodes created for layout purpose and there was some code to avoid exposing them in the accessibility tree. So removing all the anonymous during the layout refactoring was actually a good and safe thing to do. There were some helper functions to implement Apple's `AXMathRootRadicand` and `AXMathRootIndex` attributes that had to be adjusted, though. These functions used to do some work to skip the anonymous wrappers and we were actually able to simplify them.

There was also some specific code for the `RenderMathMLOperators` and their anonymous `RenderText` that were necessary to expose the text content. Actually, there was an old bug in the accessibility code and the anonymous `RenderMathMLOperators`

created by mfenced were not correctly exposed. The unit test we had for mfenced operators was only checking the text content but it was still passing and so the regression had never been detected before. After the layout refactoring we removed the anonymous `RenderText` of mfenced operators and so broke that test… We thus spent some time to fix the `RenderMathMLOperator` code. Essentially, we removed all the old hacks and only left a specific handling for mfenced operators. We also used this opportunity to improve and extend our MathML accessibility tests.

Finally, the MathML accessibility code was directly implemented into a generic `AccessibilityRenderObject` class. There was some functions to access math nodes and properties but also specific cases scattered all over the code (anonymous boxes, mfenced operators, math roles etc). In order to facilitate future work and maintenance we decided to move all the MathML code into a new `AccessibilityMathMLElement` class. Hence the work implied by the layout refactoring actually encouraged us to improve the organization and testing of our accessibility code!

Conclusion

In the past four months, Igalia's web platform team has successfully upstreamed the refactoring of WebKit's MathML renderer classes and we are now very confident about the quality of the layout code. In addition to the people mentioned above I would personally like to thank everybody who helped with this work. More specifically, I am very grateful to other people from Igalia (Martin Robinson, Sergio Villar and Manuel Rego) or Apple (Brent Fulgham and Darin Adler) who have spent some time to review patches. As a nice side effect of this work, mathematical formulas look better and the accessibility code has been improved. More is happenning in the next two phases. We are looking forward to continuing implementation of Web standards and collaboration with browser vendors at the next Web Engines Hackfest!