LuaTEX 0.82 OpenType math enhancements

Hans Hagen

Abstract

LuaTEX 0.82 (and later) have had improvements in OpenType math typesetting.

1 Introduction

When TEX typesets mathematics it makes some assumptions about the properties of fonts and dimensions of glyphs. Due to practical limitations in the traditional eight-bit fonts, such as the number of available characters in a font and a limited number of heights and depths, some juggling takes place. For instance, TEX sometimes uses dimensions as a signal to treat some characters as special. This is not a problem as long as one knows how to make a font and in practice that was done by looking at the properties of Computer Modern to implement similar shapes. After all, there are not that many math fonts around and basically there is only one engine that can deal with them properly.

However, when Microsoft set the standard for OpenType math fonts it also steered the direction of their use in rendering mathematics. This means that the LuaTEX engine, which handles OpenType fonts, has to implement some alternative code paths. At the start, this involved a bit of gambling because there was no real specification; since then we now have a better picture. One of the more complex changes that took place is in the way italic correction is applied. A dirty way out of this dilemma would be to turn the math fonts into virtual ones that match traditional TEX properties, but this would not be a nice solution.

It must be noted that in the process of implementing support for the new fonts, Taco (Hoekwater) turned some noad types (see below) into a generic noad with a subtype. This simplified the transition. At the same time, a lot of detailed control was added in the way successive characters are spaced.

In LuaTEX before 0.85, the italic correction was always added when a character got boxed (a frequently used preparation in the math builder). Now this is only done for the traditional fonts because, concerning italic correction, the OpenType standard states:[1]

1. When a run of slanted characters is followed by a straight character (such as an operator or a delimiter), the italic correction of the last glyph is added to its advance width.
2. When positioning limits on an N-ary operator (e.g., integral sign), the horizontal position of the upper limit is moved to the right by half of the italic correction, while the position of the lower limit is moved to the left by the same distance.
3. When positioning superscripts and subscripts, their default horizontal positions are also different by the amount of the italic correction of the preceding glyph.

And, with respect to kerning:

4. Set the default horizontal position for the superscript as shifted relative to the position of the subscript by the italic correction of the base glyph.

I must admit that when the first implementation showed up, my natural reaction to unexpected behaviour was just to compensate for it. One such solution was simply not to pass the italic correction to the engine and deal with it in Lua. In practice, that didn't work well for all cases; one reason was that the engine saw the combination of old fonts as a new one and followed a mixed code path.[2] Another approach I tried was a mix of manipulated italic values and Lua, but finally, as specifications settled I decided to leave it to the engine completely, if only because successive versions of LuaTEX behaved much better.

So, as we were closing in on the first stable release of LuaTEX (1.0.0 was released on September 27, 2016; this note was mostly written in the early part of 2016), I decided to fix the pending issues and sat down to look at the math-related code. I must admit that I had never looked in depth into that part of the machinery. In the next sections I will discuss some of the outcomes of this exercise.

I will also discuss some extensions that have been on the agenda for years. They are rather generic and handy, but I must also admit that the MkIV code related to math has so many options to control rendering that I'm not sure if they will ever be used in ConTEXt. Nevertheless, these generic extensions fit well into the set of basic features of LuaTEX.

2 Italic correction

As stated above, the normal code path included italic correction in all the math boxes made. This meant that, in some places, the correction had to be removed and/or moved to another place in the chain. This is a natural side effect of the fact that TEX runs over the intermediate list of math nodes

[1] Recently version 1.8 has been published on the Microsoft website.

[2] ConTEXt employed Unicode math right from the start of LuaTEX.

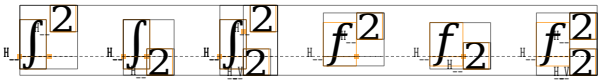

Figure 1: Italic correction examples (1): superscripts shifted right and subscripts left.

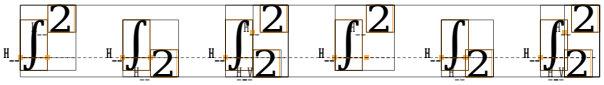

Figure 2: Italic correction examples (2): plain integral vs. integral with limits

(noads) and turns them into regular nodes, mostly glyphs, kerns, glue and boxes.

The complication is not so much the italic corrections themselves, because we could just continue to do the same, but the fact that these corrections are to be interpreted differently in case of integrals. There, the problem is that we have to (kind of) look backward at what is done in order to determine what italic corrections are to be applied.

The original solution was to keep track of the applied correction via variables but that still made some analysis necessary. In the new implementation, more information is stored in the processed noads. This is a logical choice given that we have already added other information. It also makes it possible to fix cases that will (for sure) show up in the future.

In figure 1 we show two examples of inline italic correction. The superscripts are shifted to the right and the subscripts to the left. In the case of an integral sign, we need to move half the correction. This is triggered by the `\nolimits` primitive. In figure 2 we show the difference between just an integral character and one tagged as having limits.[3]

The amount of correction, if present at all, depends on the font, and in this document we use DejaVu math. Figure 3 shows a few variants. As you can see, the amount of correction is highly font dependent.

3 Vertical delimiters

When we go into display math, there is a good chance that an integral has to be enlarged. The integral sign in Unicode has slot `0x222B`, so we can define a bigger one as follows:

```
\def\standardint{\Umathchar "1 "0 "222B }
\def\wrappedint{\mathop{\Umathchar "1 "0 "222B}}
\def\biggerint{\mathop{
    \Uleft height3ex depth3ex axis
```

[3] We show some boxes so that you can get an idea what TEX is doing. Essentially, TEX puts superscripts and subscripts on top of each other with some kern in between and then corrects the dimensions.

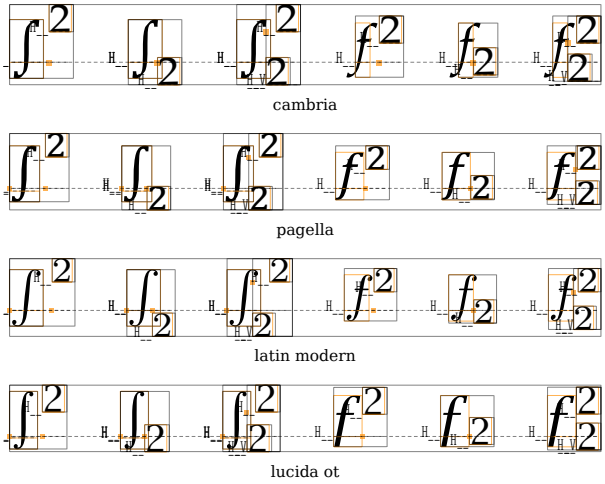

Figure 3: Italic correction examples (3): correction amounts are font-dependent.

```
    \Udelimiter "0 "0 "222B \Uright .}}
\def\evenbiggerint{\mathop{
    \Uleft height 6ex depth 6ex axis
    \Udelimiter "0 "0 "222B \Uright .}}
```

The `axis` keyword will apply a shift up over the size of the current styles math axis. We use this in some examples as:

```
$
\displaystyle\standardint  ^a_b\enspace
\displaystyle\wrappedint   ^a_b\enspace
\displaystyle\biggerint    ^a_b\enspace
\displaystyle\evenbiggerint^a_b\enspace
$
```

In figure 4 you can see some subtle differences. The wrapped version doesn't shift the superscript and subscript. The reason is that the operator is hidden in its own wrapper and the scripts attach at an outer level. So, unless we start analyzing the innermost noad and apply that to the outer, we cannot know the shift. Such analyzing is asking for problems: where do we stop and what slight variations do we take into account? It's better to be predictable.

Another observation is that Latin Modern does not provide (at least not yet) large integrals at all.

The following four cases are equivalent:

```
\Uleft height 3ex depth 3ex axis
\Udelimiter "0 "0 "222B
\Uright .

\Uleft .
\Uright height 3ex depth 3ex axis
\Udelimiter "0 "0 "222B

\Uleft   .
\Umiddle height 3ex depth 3ex axis
```

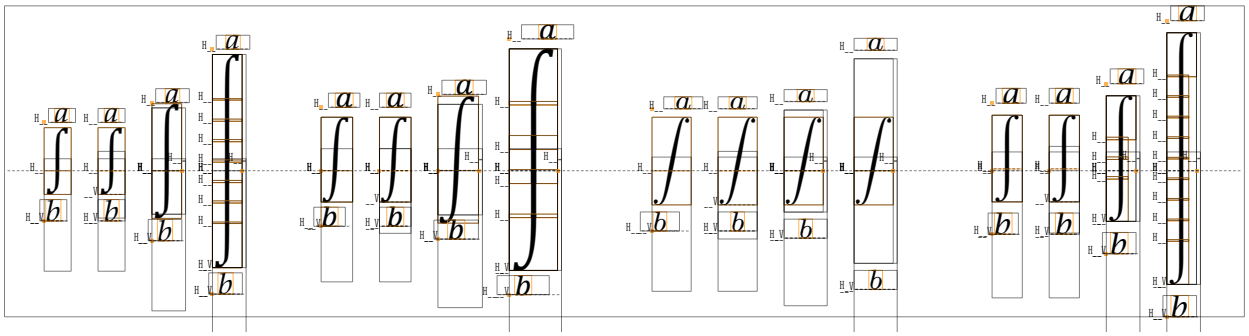

Figure 4: Comparison of integral variants (standard, wrapped, bigger, even bigger) among fonts: TEX Gyre Pagella, Cambria, Latin Modern, and Lucida OT.

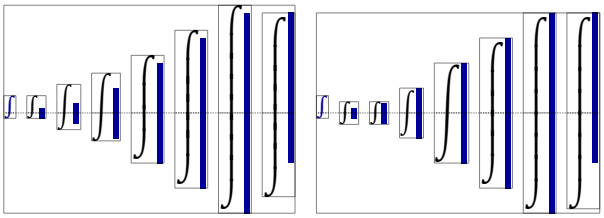

Figure 5: Cambria integrals, adaptive; `axis` left, `noaxis` right.

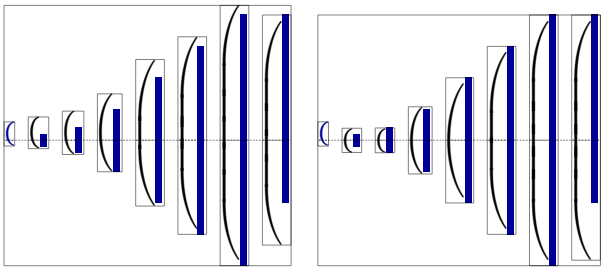

Figure 6: Cambria left parenthesis, adaptive; `axis` left, `noaxis` right.

```
\Udelimiter "0 "0 "222B
\Uright   .

\Uleft    .
\Umiddle height 3ex depth 3ex axis
\Udelimiter "0 "0 "222B
\Uright   .
```

However, because this all looks a bit clumsy, we now provide a new primitive:

```
\Uvextensible
    height ⟨dimension⟩
    depth ⟨dimension⟩
    [no]axis
    exact
    ⟨delimiter⟩
```

The symbol to be constructed will have size `height` plus `depth`. When an `axis` is specified, the symbol will be shifted up, which is normally the case for such symbols. The keyword `exact` will correct the dimensions when no exact match is made, and this can be the case as long as we use the stepwise larger glyphs and before we end up using the composed shapes. When no dimensions are specified, the normal construction takes place and the only keyword that can be used then is `noaxis` which keeps the axis out of the calculations. After about a week of experimenting and exploring options, this combination made most sense, read: no fuzzy heuristics but predictable behaviour. After all, one might need different solutions for different fonts or circumstances and the applied logic (and expectations) can (and will, for sure) differ per macro package. Figures 5–8 show some examples.

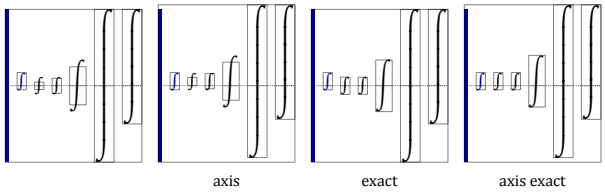

Figure 7: Cambria integrals, with dimensions.

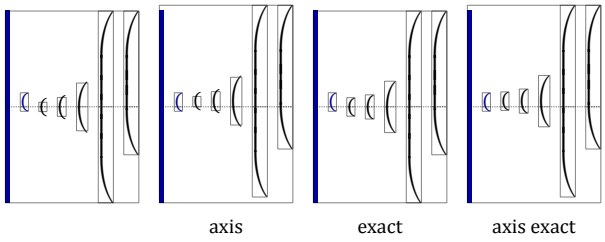

Figure 8: Cambria left parenthesis, with dimensions.

4 Horizontal delimiters

Horizontal extenders also have some new options. Although one can achieve similar results with macros, the following might look a bit more natural. Also,

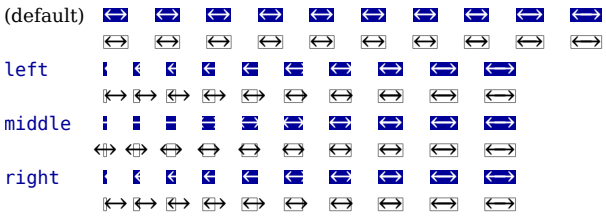

Figure 9: Stepwise wider `\Uhextensible` with options (Cambria).

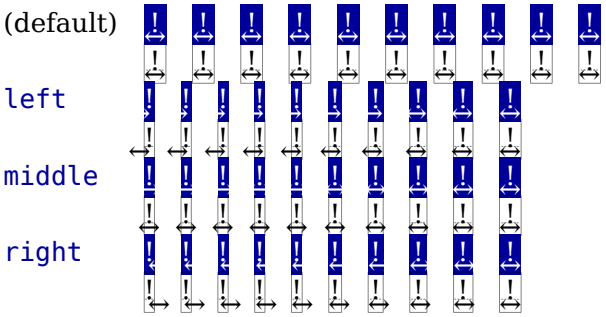

Figure 10: Stepwise wider `\Udelimiterunder` with options (Cambria).

some properties are lost once the delimiter is constructed, so macros can become complex when trying to determine the original dimensions involved.

We start with the new `\Uhextensible` primitive that accepts a dimension. It's just a variant of the over and under delimiters with no content part.

```
\Uvextensible
    height ⟨dimension⟩
    depth ⟨dimension⟩
    left | middle | right
    ⟨family⟩
    ⟨slot⟩
```

So for example you can say:

```
$\Uhextensible width 30pt 0 "2194$
```

The `left`, `middle` and `right` keywords are only interpreted when the requested size can't be met due to stepwise larger glyph selection (i.e., before we start using arbitrary sizes made of snippets). Figure 9 shows what we get when we step from 2–20 points by increments of 2 points in Cambria.

The dimensions and options can also be given to the four primitives:
`\Uoverdelimiter` `\Uunderdelimiter`
`\Udelimiterover` `\Udelimiterunder`
Figure 10 shows what happens when the delimiter is smaller than requested. The source for the samples looks like this:

```
$\Udelimiterunder width 1pt 0 "2194
  {\hbox{\strut !}}
```

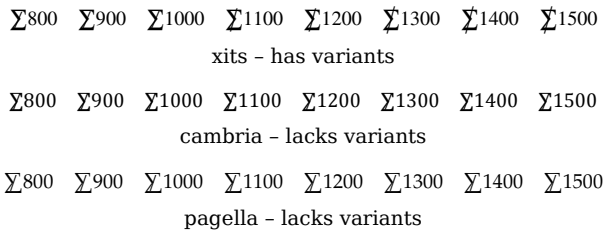

Figure 11: Using `overlay` in `\Umathaccent`.

When no dimension is given the keywords are ignored as it makes no sense to mess with the extensible in that case.

5 Accents

Many years ago, I observed that overlaying characters (which happens when we negate an operator which has no composed negation glyph) didn't always give nice results and, therefore, a tracker item was created. When going over the todo list, I ran across a suggested patch by Khaled Hosny that added an overlay accent type. As the suggested solution fits in with the other extensions, a variant has been implemented.

The results definitely depend on the quality and completeness of the font, so here we will use XITS. The placement of an `overlay` also depends on the top accent shift as specified in the font for the used glyph. Instead of a fixed criterion for trying to find the best match, an additional `fraction` (numerator) parameter can be specified. A value of 800 means that the target width is 800/1000.

The `\Umathaccent` command now has the following syntax:

```
\Umathaccent
    [top | bottom | overlay]
    [fixed]
    [fraction ⟨number⟩]
    ⟨delimiter⟩
    {⟨content⟩}
```

When we have an overlay, the fraction concerns the height; otherwise it concerns the width of the nucleus. In both cases, it is only applied when searching for stepwise larger glyphs, as extensibles are not influenced. An example of a specification is:

```
\Umathaccent
    overlay "0 "0 "0338
    fraction 950
    {\Umathchar"1"0"2211}
```

Figure 11 shows what we get when we use different fractions (from 800 up to 1500 with a step of 100). We see that `\overlay` is not always useful.

Normally you can forget about the factor because overlays make most sense for inline math, which

	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$
`exact`	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$
`noaxis`	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$
`exact noaxis`	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$

Figure 12: Skewed fraction results in Latin Modern.

uses relatively small glyphs, so we can get x̸ x̸ xx̸x with the following code:

```
$\Umathaccent overlay "0 "0 "0338 {x}$
$\Umathaccent overlay "0 "0 "0338 {\tf x}$
$\Umathaccent overlay "0 "0 "0338 {\tf xxx}$
```

A normal accent can also be influenced by `fraction`:

$\overbrace{a \times b}$ $\overbrace{a \times b}$ $\overbrace{a \times b}$ $\overbrace{a \times b}$ $\overbrace{a \times b}$

6 Fractions

A normal fraction has a reasonable thick rule but as soon as you make it bigger you will notice a peculiar effect:

$x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$

1pt 2pt 3pt 4pt 5pt

Such a fraction is specified as:

```
x + { {a} \abovewithdelims () 5pt {b} }
```

A new keyword `exact` avoids the excessive spacing:

```
x + { {a} \abovewithdelims () exact 5pt {b} }
```

Now we get:

$x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$

1pt 2pt 3pt 4pt 5pt

One way to get consistent spacing in such fractions is to use struts:

```
x + { {\strut a} \abovewithdelims () exact 5pt
    {\strut b} }
```

Now we get:

$x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$

1pt 2pt 3pt 4pt 5pt

Yet another way to increase the distance between the rule and text a bit is:

```
\Umathfractionnumvgap  \displaystyle4pt
\Umathfractiondenomvgap\displaystyle4pt
```

This looks quite consistent:

$x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$

1pt 2pt 3pt 4pt 5pt

Here we use code like:

```
$\displaystyle x +
 {{a} \abovewithdelims() exact 2pt {b}}$
```

Using struts, it is best to zero the gap:

$x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$ $x + \left(\frac{a}{b}\right)$

1pt 2pt 3pt 4pt 5pt

Here we use code like:

```
$\displaystyle x + {{\strut a} \abovewithdelims()
                   exact 2pt {\strut b}}$
```

7 Skewed fractions

The math parameter table contains values specifying horizontal and vertical gaps for skewed fractions. Some guessing is needed in order to implement something that uses them, so we now provide a primitive similar to the other fraction related ones but with a few options that one can use to influence the rendering. Of course, a user can mess around directly with the parameters `\Umathskewedfractionhgap` and `\Umathskewedfractionvgap`.

The syntax used here is:

```
{ {1} \Uskewed / ⟨options⟩ {2} }
{ {1} \Uskewedwithdelims / () ⟨options⟩ {2} }
```

The options can be `noaxis` and `exact`, a combination of them or just nothing. By default we add half the axis to the shifts and also by default we zero the width of the middle character. For Latin Modern, the results are shown in figure 12.

8 Side effects

Not all bugs reported as such are really bugs. Here is one that came from a misunderstanding: In Eijkhout's *TEX by Topic*, the rules for handling styles in scripts are described as follows:

- In any style superscripts and subscripts are taken from the next smaller style. Exception: in display style they are taken in script style.

- Subscripts are always in the cramped variant of the style; superscripts are only cramped if the original style was cramped.
- In an `..\over..` formula in any style the numerator and denominator are taken from the next smaller style.
- The denominator is always in cramped style; the numerator is only in cramped style if the original style was cramped.
- Formulas under a `\sqrt` or `\overline` are in cramped style.

In LuaTEX, one can set the styles in more detail, which means that you sometimes have to set both normal and cramped styles to get the effect you want. If we force styles in the script using `\scriptstyle` and `\crampedscriptstyle` we get the following (all render the same):

default $b_{x=xx}^{x=xx}$
script $b_{x=xx}^{x=xx}$
crampedscript $b_{x=xx}^{x=xx}$

This is coded as follows:

```
$b_{x=xx}^{x=xx}$
$b_{\scriptstyle x=xx}^{\scriptstyle x=xx}$
$b_{\crampedscriptstyle x=xx}
  ^{\crampedscriptstyle x=xx}$
```

Now we set the following parameters:

```
\Umathordrelspacing\scriptstyle=30mu
\Umathordordspacing\scriptstyle=30mu
```

This gives:

default $b_{x=xx}^{x=x\,x}$
script $b_{x\;=x\;x}^{x\;=x\;x}$
crampedscript $b_{x=xx}^{x=xx}$

Since the result is not what is expected (visually), we should say:

```
\Umathordrelspacing\scriptstyle=30mu
\Umathordordspacing\scriptstyle=30mu
\Umathordrelspacing\crampedscriptstyle=30mu
\Umathordordspacing\crampedscriptstyle=30mu
```

Now we get:

default $b_{x\;=x\;x}^{x\;=x\;x}$
script $b_{x\;=x\;x}^{x\;=x\;x}$
crampedscript $b_{x\;=x\;x}^{x\;=x\;x}$

mode	down	up	
0	dynamic	dynamic	$\mathrm{CH_2 + CH_2^+ + CH_2^2}$
1	d	u	$\mathrm{CH_2 + CH_2^+ + CH_2^2}$
2	s	u	$\mathrm{CH_2 + CH_2^+ + CH_2^2}$
3	s	$u+s-d$	$\mathrm{CH_2 + CH_2^+ + CH_2^2}$
4	$d+(s-d)/2$	$u+(s-d)/2$	$\mathrm{CH_2 + CH_2^+ + CH_2^2}$
5	d	$u+s-d$	$\mathrm{CH_2 + CH_2^+ + CH_2^2}$

$\mathrm{CH_2 + CH_2^+ + CH_2^2}$ 0 $\mathrm{CH_2 + CH_2^+ + CH_2^2}$ 1 $\mathrm{CH_2 + CH_2^+ + CH_2^2}$ 2

$\mathrm{CH_2 + CH_2^+ + CH_2^2}$ 3 $\mathrm{CH_2 + CH_2^+ + CH_2^2}$ 4 $\mathrm{CH_2 + CH_2^+ + CH_2^2}$ 5

Figure 13: The effect of setting `\mathscriptsmode`.

9 Fixed scripts

We have three parameters that are used for anchoring superscripts and subscripts, alone or in combinations.

d `\Umathsubshiftdown`
u `\Umathsupshiftup`
s `\Umathsubsupshiftdown`

When we set `\mathscriptsmode` to a value other than zero, these are used for calculating fixed positions. This is something that is needed in, for instance, chemical equations. You can manipulate the mentioned variables to achieve different effects, and the specifications are shown in figure 13, with enlarged examples below the table.

10 Remark

The changes that we have made are hopefully not too intrusive. Instead of extending existing commands, new ones were introduced so that compatibility should not be a significant problem. To some extent, these extensions violate the principle that extensions should be done in Lua, but TEX being a math renderer and OpenType replacing old font technology, we felt that we should make an exception here. Hopefully, not too many bugs were introduced.

◇ Hans Hagen
Pragma ADE
http://pragma-ade.com