A different approach to math spacing

Introduction

The TEX engine is famous for its rendering of math and even after decades there is no real contender. And so there also is no real pressure to see if we can do better. However, when Mikael Sundqvist ran into a Swedish math rendering specification and we started discussing a possible support for that in ConTEXt, it quickly became clear that the way TEX does spacing is a bit less flexible than one wishes for. We already have much of what is needed in place but it also has to work well with how TEX sees things:

1. Math is made from a sequence of atoms: a quantity with a nucleus, superscript subscript.[1] Atoms are spaced by `\thinmuskip`, `\medmuskip` and `\thickmuskip` or nothing, and that is sort of hard coded.
2. Atoms are organized by class and there are seven (or eight, depending on how you look at it) of them visible: binary symbols, relations, etc. The invisible ones, composites like fractions and fenced material (we call them molecules) are at some point mapped onto the core set. Molecules like fences have a different class left and right of the fenced material.
3. In addition the engine itself has all kind of spacing related parameters and these kick in automatically and sometimes have side effects. The same is true for penalties.

The normal approach to spacing other than imposed by the engine is to use correction space, like `\,` and I think that quite some TEX users think that this is how it is supposed to be. The standard way to enter math relates to scientific publishing and there the standards are often chiseled in stone so why should users tweak anyway. However, in ConTEXt we tend to start from the users and not the publishers end so there we can decide to follow different routes. Users can always work around something they don't like but we focus on reliable input giving predictable output. Also, when reading on, it is good to realize that it is all about the user experience here: it should look nice (which then of course makes one become aware of issues elsewhere) and we don't care much about specific demands of publishers in the scientific field: the fact that they often re-key content doesn't go well with users paying attention themselves, let alone the fact that nowadays they can demand word processor formats.

The three mentioned steps are fine for the average case but sometimes make no sense. It was definitely the best approach given time and resources but when LuaTEX went OpenType a lot of parameters were added and at that time we therefore added spacing by class pair. That not only decoupled the relation between the three (configurable) muskip parameters but also made it possible to use plenty of them. Now it must be said that for consistency having these three skips works great but given the tweaking expected from users consistency is not always what comes out.

This situation is very well comparable to the proclaimed qualities of the typesetting of text by TEX. Yes, it can do a great job, and often does, but users can mess up quite well. I remember that when we did tests with hz the outcomes were pretty

1. I suddenly realize why in the engine noads have a nucleus field: they are atoms . . . but what does that make super and subscripts.

unimpressive. When you give an audience a set of sample renderings, where each sample is slightly different and each user gets a randomized subset, the sudden lack of being able to compare (and agree) with another TEXie makes for interesting conclusions. They look for the opposites of what is claimed to be perfect. So, two lines with hyphens rate low, even if not doing it would look worse. The same for a few short words in the last line of a paragraph. Excessive spacing is also seen as bad. So, when asked why some paragraphs looked okay noticing (excessive and troublesome) expansion was not seen as a problem; instead it were hyphens that got the attraction.

The same is probably true for math: the input with lots of correction spaces or commands where characters would do can be horrible but it's just the way it is supposed to be. The therefore expected output can only be perfect, right, independent of how one actually messed up spacing. But personally I think that it is often spacing messed up by users that make a TEX document recognizable. It compares to word processor results that one can sometimes identify by multiple consecutive spaces in the typeset text instead of using a glue model like TEX. Reaching perfection is not always trivial, but fortunately we can also find plenty of nice looking documents done with TEX.

The TEXbook has an excellent and intriguing chapter on the fine points of math and it definitely shows why Don Knuth wrote TEX as a tool for his books. He pays a lot of attention to detail and that is also why it all works out so well. If you need to render from unseen sources (as happens in an xml workflow) coming from several authors and have time nor money to check everything, you're off worse. And I'm not even talking of input where invisible Unicode spacing characters are injected. It is the TEX book(s) that has drawn me to this program and believe it or not, in the first project I was involved in that demanded typeset (quantum mechanics) math the ibm typewriter with changing bulbs ruled the scenery. In fact, our involvement was quickly cut off when we dared to show a chapter done in TEX that looked better.

Apart from an occasional tweak, in ConTEXt we never really used this opened up math atom pair spacing mechanism available in LuaTEX extensively. So, when I was pondering how to proceed it stroke me that it would make sense to generalize this mechanism. It was already possible (via a mode parameter) to bypass the second step mentioned above, but we definitely needed more than the visible classes that the engine had. In ConTEXt we already had more classes but those were meant for assigning characters and commands to specific math constructs (think of fences, fractions and radicals) so in the end they were not really classes. Considering this option was made easier by the fact that Mikael would do the testing and help configuring the defaults, which all will result in a new math user manual.

There are extensions introduced in LuaTEX and later LuaMetaTEX that are not discussed here. In this expose we concentrate on the features that were explored, extended and introduced while we worked on updating math support in LMTX.

An example

Before we go into details, let's give an example of unnoticed spacing effects. We use three simple formulas all using fractions:

```
\ruledhbox{$\frac{x^2}{a+1}$}
```

and:

```
\ruledhbox{$x + \frac{x^2}{a+1} = 10$}
```

as well as:

```
\ruledhbox{$\frac{1}{2}\frac{1}{2}x$}
```

$$\frac{x^2}{a+1} \quad x+\frac{x^2}{a+1}=10 \quad \frac{1}{2}\,\frac{1}{2}x$$

If you look closely you see that the fraction has a little space at the left and right. Where does that come from? Because we normally don't put a tight frame around a fraction, we are not really aware of it. The spacing between what are called ordinary, operator, binary, relation and other classes of atoms is explained in the TEXbook (or "TEX by Topic" if you want a summary) and basically we have a class by class matrix that is built into TEX. The engine looks at successive items and spacing depends on their (perceived) class. Because the number of classes is limited, and because the spacing pairs are hard coded, the engine cheats a little. Depending on what came before or comes next the class of an atom is adapted to suit the spacing matrix. One can say that a "reading mathematician" is built in. And most of the decisions are okay. If needed one can always wrap something in e.g. `\mathrel` but of course that also can interfere with grouping. All this is true for TEX, pdfTEX, XETEX and LuaTEX, but a bit different in LuaMetaTEX as we will see.

The little spacing on both edges of the fraction is a side effect of the way they are built internally: fractions are actually a generalized form of "stuff put on top of other stuff" and they can have left and/or right delimiters: this is driven by primitives that have names like `\atop` and `\atopwithdelims`. The way the components are placed is (especially in the case of OpenType) driven by lots of parameters and I will leave that out of the discussion.

When there are no delimiters, a so called `\nulldelimiterspace` will be injected. That parameter is set to 1.2 points and I have to admit that in ConTEXt I never considered letting that one adapt to the body font size, which means that, as we default to a 12 point body font, the value there should have been 1.44 points: mea culpa. When we set this parameter to zero point, we get this:

$$\frac{x^2}{a+1} \quad x+\frac{x^2}{a+1}=10 \quad \frac{1}{2}\frac{1}{2}x$$

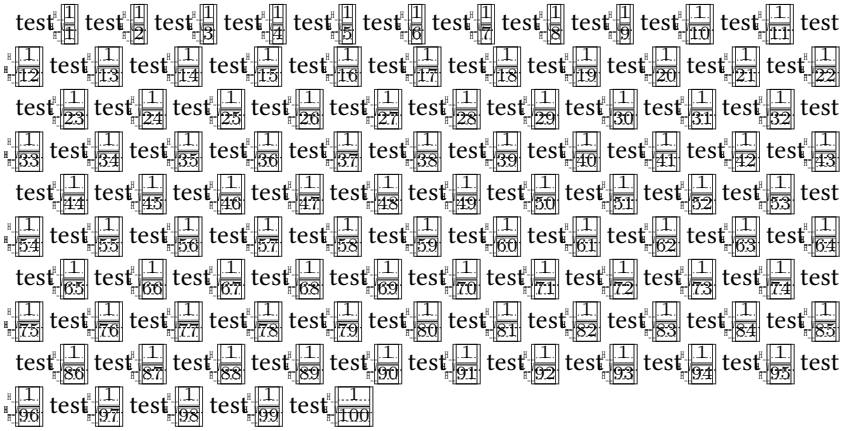
As intermezzo and moment of contemplation I show some examples of fractions mixed into text. When we have the delimiter space set we get this:

test $\frac{1}{1}$ test $\frac{1}{2}$ test $\frac{1}{3}$ test $\frac{1}{4}$ test $\frac{1}{5}$ test $\frac{1}{6}$ test $\frac{1}{7}$ test $\frac{1}{8}$ test $\frac{1}{9}$ test $\frac{1}{10}$ test $\frac{1}{11}$ test $\frac{1}{12}$ test $\frac{1}{13}$ test $\frac{1}{14}$ test $\frac{1}{15}$ test $\frac{1}{16}$ test $\frac{1}{17}$ test $\frac{1}{18}$ test $\frac{1}{19}$ test $\frac{1}{20}$ test $\frac{1}{21}$ test $\frac{1}{22}$ test $\frac{1}{23}$ test $\frac{1}{24}$ test $\frac{1}{25}$ test $\frac{1}{26}$ test $\frac{1}{27}$ test $\frac{1}{28}$ test $\frac{1}{29}$ test $\frac{1}{30}$ test $\frac{1}{31}$ test $\frac{1}{32}$ test $\frac{1}{33}$ test $\frac{1}{34}$ test $\frac{1}{35}$ test $\frac{1}{36}$ test $\frac{1}{37}$ test $\frac{1}{38}$ test $\frac{1}{39}$ test $\frac{1}{40}$ test $\frac{1}{41}$ test $\frac{1}{42}$ test $\frac{1}{43}$ test $\frac{1}{44}$ test $\frac{1}{45}$ test $\frac{1}{46}$ test $\frac{1}{47}$ test $\frac{1}{48}$ test $\frac{1}{49}$ test $\frac{1}{50}$ test $\frac{1}{51}$ test $\frac{1}{52}$ test $\frac{1}{53}$ test $\frac{1}{54}$ test $\frac{1}{55}$ test $\frac{1}{56}$ test $\frac{1}{57}$ test $\frac{1}{58}$ test $\frac{1}{59}$ test $\frac{1}{60}$ test $\frac{1}{61}$ test $\frac{1}{62}$ test $\frac{1}{63}$ test $\frac{1}{64}$ test $\frac{1}{65}$ test $\frac{1}{66}$ test $\frac{1}{67}$ test $\frac{1}{68}$ test $\frac{1}{69}$ test $\frac{1}{70}$ test $\frac{1}{71}$ test $\frac{1}{72}$ test $\frac{1}{73}$ test $\frac{1}{74}$ test $\frac{1}{75}$ test $\frac{1}{76}$ test $\frac{1}{77}$ test $\frac{1}{78}$ test $\frac{1}{79}$ test $\frac{1}{80}$ test $\frac{1}{81}$ test $\frac{1}{82}$ test $\frac{1}{83}$ test $\frac{1}{84}$ test $\frac{1}{85}$ test $\frac{1}{86}$ test $\frac{1}{87}$ test $\frac{1}{88}$ test $\frac{1}{89}$ test $\frac{1}{90}$ test $\frac{1}{91}$ test $\frac{1}{92}$ test $\frac{1}{93}$ test $\frac{1}{94}$ test $\frac{1}{95}$ test $\frac{1}{96}$ test $\frac{1}{97}$ test $\frac{1}{98}$ test $\frac{1}{99}$ test $\frac{1}{100}$

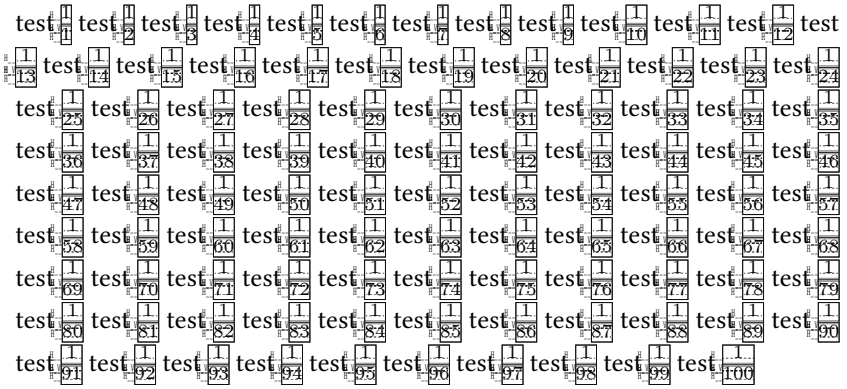
While with zero it looks like this, quite a different outcome:

test $\frac{1}{1}$ test $\frac{1}{2}$ test $\frac{1}{3}$ test $\frac{1}{4}$ test $\frac{1}{5}$ test $\frac{1}{6}$ test $\frac{1}{7}$ test $\frac{1}{8}$ test $\frac{1}{9}$ test $\frac{1}{10}$ test $\frac{1}{11}$ test $\frac{1}{12}$ test $\frac{1}{13}$ test $\frac{1}{14}$ test $\frac{1}{15}$ test $\frac{1}{16}$ test $\frac{1}{17}$ test $\frac{1}{18}$ test $\frac{1}{19}$ test $\frac{1}{20}$ test $\frac{1}{21}$ test $\frac{1}{22}$ test $\frac{1}{23}$ test $\frac{1}{24}$ test $\frac{1}{25}$ test $\frac{1}{26}$ test $\frac{1}{27}$ test $\frac{1}{28}$ test $\frac{1}{29}$ test $\frac{1}{30}$ test $\frac{1}{31}$ test $\frac{1}{32}$ test $\frac{1}{33}$ test $\frac{1}{34}$ test $\frac{1}{35}$ test $\frac{1}{36}$ test $\frac{1}{37}$ test $\frac{1}{38}$ test $\frac{1}{39}$ test $\frac{1}{40}$ test $\frac{1}{41}$ test $\frac{1}{42}$ test $\frac{1}{43}$ test $\frac{1}{44}$ test $\frac{1}{45}$ test $\frac{1}{46}$ test $\frac{1}{47}$ test $\frac{1}{48}$ test $\frac{1}{49}$ test $\frac{1}{50}$ test $\frac{1}{51}$ test $\frac{1}{52}$ test $\frac{1}{53}$ test $\frac{1}{54}$ test $\frac{1}{55}$ test $\frac{1}{56}$ test $\frac{1}{57}$ test $\frac{1}{58}$ test $\frac{1}{59}$ test $\frac{1}{60}$ test $\frac{1}{61}$ test $\frac{1}{62}$ test $\frac{1}{63}$ test $\frac{1}{64}$ test $\frac{1}{65}$ test $\frac{1}{66}$ test $\frac{1}{67}$ test $\frac{1}{68}$ test $\frac{1}{69}$ test $\frac{1}{70}$ test $\frac{1}{71}$ test $\frac{1}{72}$ test $\frac{1}{73}$ test $\frac{1}{74}$ test $\frac{1}{75}$ test $\frac{1}{76}$ test $\frac{1}{77}$ test $\frac{1}{78}$ test $\frac{1}{79}$ test $\frac{1}{80}$ test $\frac{1}{81}$ test $\frac{1}{82}$ test $\frac{1}{83}$ test $\frac{1}{84}$ test $\frac{1}{85}$ test $\frac{1}{86}$ test $\frac{1}{87}$ test $\frac{1}{88}$ test $\frac{1}{89}$ test $\frac{1}{90}$ test $\frac{1}{91}$ test $\frac{1}{92}$ test $\frac{1}{93}$ test $\frac{1}{94}$ test $\frac{1}{95}$ test $\frac{1}{96}$ test $\frac{1}{97}$ test $\frac{1}{98}$ test $\frac{1}{99}$ test $\frac{1}{100}$

A little tracing shows it more clearly:

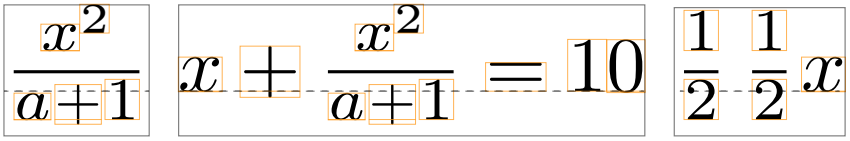


You can zoom in and see where it interferes with margin alignment.

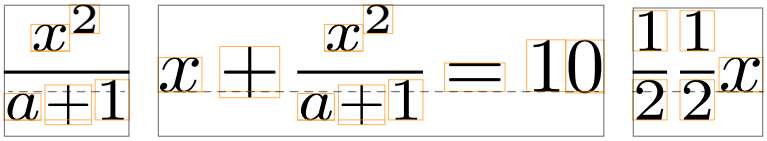


So, if you ever meet a user who claims perfection and superiority of typesetting, check out her/his work which might have inline fractions done the spacy way. It might make other visually typesetting claims less trustworthy. And yes, one can wonder if margin kerning could help here but as this content is wrapped in boxes it is unlikely to work out well (and not worth the effort).

In order to get a better picture of the spacing, two more renderings are shown. This time we show the bounding boxes of the characters too (you might need to zoom in to see it):

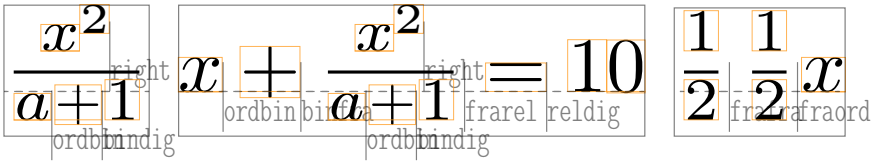


Again we also show the zero case

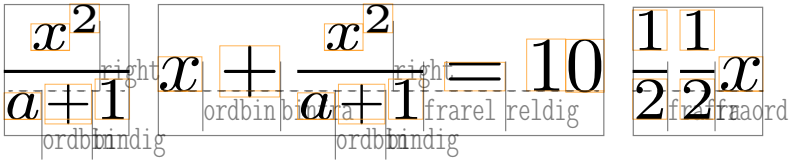


This makes clear why there actually is this extra space around a fraction: regular operators have side bearings and thereby have some added space. And when we put a fraction in front of a symbol we need that little extra space. Of course a proper class pair spacing value could do the job but there is no fraction class. The engine cheats by changing the class depending on what follows or came before and this is why on the average it looks okay. However, these examples demonstrate that there are some assumptions with regard to for instance fonts and this is one of the reasons why the more or less official expected OpenType behavior as dictated by the Cambria font doesn't always work out well for fonts that evolved from the ones used in the TEX community. Also imagine how this interferes with the fact that traditional TEX fonts and the machinery do magic with cheating about width combined with italic correction (all plausible and quite clever but somewhat tricky with respect to OpenType).

Because here we discuss the way LuaMetaTEX and ConTEXt deal with this, the following examples show a probably unexpected outcome. Again first the non-zero case:



And here the zero case:



I will not go into details about the way fractions are supported in the engine because some extensions are already around for quite a while. The main observation here is that in LuaMetaTEX we have alternative primitives that assume forward scanning, as if the numerator and denominator are arguments. The engine also supports skewed (vulgar) fractions natively where numerator and denominator are raised and lowered relative to the (often) slash. Many aspects of the rendering can be tuned in the so called font goodie files, which is also the place where we define the additional font parameters.

Atom spacing

If you are familiar with traditional TEX you know that there is some built in `ordbin` spacing. But there is no such pair for a fraction and a relation, simply because there is no fraction class. However, in LuaMetaTEX there is one, and we'd better set it up if we zero the margins of a fraction.

It is worth noticing that fractions are sort of special anyway. The official syntax is `n \over m` and numerator and denominator can be sub formulas. This is the one case where the parser sort of has to look back, which is tricky because the machinery is a forward looking one. Therefore, in order to get the expected styling (or avoid unexpected side effects) one will normally wrap all in braces as in: `{ {n} \over{m} }` which of course kind defeats the simple syntax which probably is supported for `1\over2` kind of usage, so a next challenge is to make `1/2` come out right. All this means that in practice we have wrappers like `\frac` which accidentally in LuaMetaTEX can be defined using forward looking primitives with plenty extra properties driven by keywords. It also means that fractions as expected by the engine due to wrapping actually can be a different kind of atom, which can have puzzling side effects with respect to spacing (because the remapping happens unseen).

Interesting is that adapting LuaMetaTEX to a more extensive model was quite doable, also because the code base had already been made more configurable. Of course it involved quite a bit of tedious editing and throwing out already nice and clean code that had taken some effort, but that's the way it is. Of course more classes also means that some storage properties had to be adapted within the available space but by sacrificing families that was possible. With 64 potential classes we now are back to 64 families compared to 7 classes and 256 families in LuaTEX and 7 classes and 16 families in traditional TEX.

Also interesting is that the new implementation is actually somewhat simpler and therefore the binary is a tad smaller too. But does all that mean that there were no pitfalls? Sure there were! It is worth noticing that doing all this reminded me of the early days of LuaTEX development, where Taco and I exchanged binaries and TEX code in a more or less constant way using Skype. For LuaMetaTEX we used good old mail for files and Mojca's build farm for binaries and Mikael and I spent many months exchanging information and testing out alternatives on a daily basis: it is in my opinion the only way to do this and it's fun too. It has been a lot of work but once we got going there was nothing that could stop us. A side effect was that there were no updates during this period, which was something users noticed.

In the spacing matrix there is `inner` and internally there's also some care to be taken of `vcenter`. The `inner` class is actually shared with the `variable` class which is not so much a real class but more a signal to the engine that when an alphabetic or numeric character is included it has to come from a specific family: upright family zero or math italic family one in traditional speak. But, what if we don't have that setup? Well, then one has to make sure that this special class number is not associated (which is no big deal). It does mean that when we extend the repertoire of classes we cannot use slot seven. Always keep in mind that classes (and thereby signals) get assigned to characters (some defaults by the engine, others by the macro package). It is why in ConTEXt we use abstract class numbers, just in case the engine gets adapted.

We also cannot use slot eight because that one is a signal too: for a possible active math character, a feature somewhat complicated by the fact that it should not interfere with passing around such active characters in arguments. In math mode where we have lots of macros passing around content, this special class works around these side effects. We don't need this feature in ConTEXt because contrary to other macro packages we don't handle primes, pseudo superscripts potentially followed by other super and subscripts by making the `'` an active character and thereby a macro in math mode. This trickery again closely relates to preferable input, font properties,

and limitations of memory and such at the time TEX showed up (much has to fit into 8, 16 or 32 bits, so there is not much room for e.g. more than 8 classes). Since we started with MkIV the way math is dealt with is a bit different than normally done in TEX anyway.

Atom rules

We can now control the spacing between every atom but unfortunately that is not good enough. Therefore, we arrive at yet another feature built into the engine: turning classes into other classes depending on neighbors. And this is precisely why we have certain classes. Let's quote "TEX by Topic": *The cases ⋆ (in the atom spacing matrix) cannot occur, because a* `bin` *object is converted to* `ord` *if it is the first in the list, preceded by* `bin`, `op`, `open`, `punct`, `rel`, *or followed by* `close`, `punct`, *or* `rel`*; also, a* `rel` *is converted to* `ord` *when it is followed by* `close` *or* `punct`.

We can of course keep these hard coded heuristics but can as well make that bit of code configurable, which we did. Below is demonstrated how one can set up the defaults at the TEX end. We use symbolic names for the classes.

```
\setmathatomrule  \mathbegincode         \mathbinarycode       % old
  \allmathstyles    \mathordinarycode      \mathordinarycode   % new

\setmathatomrule  \mathbinarycode        \mathbinarycode
  \allmathstyles    \mathbinarycode        \mathordinarycode
\setmathatomrule  \mathoperatorcode      \mathbinarycode
  \allmathstyles    \mathoperatorcode      \mathordinarycode
\setmathatomrule  \mathopencode          \mathbinarycode
  \allmathstyles    \mathopencode          \mathordinarycode
\setmathatomrule  \mathpunctuationcode   \mathbinarycode
  \allmathstyles    \mathpunctuationcode   \mathordinarycode
\setmathatomrule  \mathrelationcode      \mathbinarycode
  \allmathstyles    \mathrelationcode      \mathordinarycode

\setmathatomrule  \mathbinarycode        \mathclosecode
  \allmathstyles    \mathordinarycode      \mathclosecode
\setmathatomrule  \mathbinarycode        \mathpunctuationcode
  \allmathstyles    \mathordinarycode      \mathpunctuationcode
\setmathatomrule  \mathbinarycode        \mathrelationcode
  \allmathstyles    \mathordinarycode      \mathrelationcode

\setmathatomrule  \mathrelationcode      \mathclosecode
  \allmathstyles    \mathordinarycode      \mathclosecode
\setmathatomrule  \mathrelationcode      \mathpunctuationcode
  \allmathstyles    \mathordinarycode      \mathpunctuationcode
```

Watch the special class with `\mathbegincode`. This is actually class 62 so you don't need much fantasy to imagine that class 63 is `\mathendcode`, but that one is not yet used. In a similar fashion we can initialize the spacing itself:[2]

```
\setmathspacing\mathordcode  \mathopcode    \allmathstyles \thinmuskip
\setmathspacing\mathordcode  \mathbincode   \allsplitstyles\medmuskip
\setmathspacing\mathordcode  \mathrelcode   \allsplitstyles\thickmuskip
\setmathspacing\mathordcode  \mathinnercode \allsplitstyles\thinmuskip

\setmathspacing\mathopcode   \mathordcode   \allmathstyles \thinmuskip
\setmathspacing\mathopcode   \mathopcode    \allmathstyles \thinmuskip
\setmathspacing\mathopcode   \mathrelcode   \allsplitstyles\thickmuskip
```

2. Constant, engine specific, numbers like these are available in tables at the Lua end so we can change them and users can check that.

```
\setmathspacing\mathopcode   \mathinnercode \allsplitstyles\thinmuskip

\setmathspacing\mathbincode  \mathordcode   \allsplitstyles\medmuskip
\setmathspacing\mathbincode  \mathopcode    \allsplitstyles\medmuskip
\setmathspacing\mathbincode  \mathopencode  \allsplitstyles\medmuskip
\setmathspacing\mathbincode  \mathinnercode \allsplitstyles\medmuskip

\setmathspacing\mathrelcode  \mathordcode   \allsplitstyles\thickmuskip
\setmathspacing\mathrelcode  \mathopcode    \allsplitstyles\thickmuskip
\setmathspacing\mathrelcode  \mathopencode  \allsplitstyles\thickmuskip
\setmathspacing\mathrelcode  \mathinnercode \allsplitstyles\thickmuskip

\setmathspacing\mathclosecode\mathopcode    \allmathstyles \thinmuskip
\setmathspacing\mathclosecode\mathbincode   \allsplitstyles\medmuskip
\setmathspacing\mathclosecode\mathrelcode   \allsplitstyles\thickmuskip
\setmathspacing\mathclosecode\mathinnercode \allsplitstyles\thinmuskip

\setmathspacing\mathpunctcode\mathordcode   \allsplitstyles\thinmuskip
\setmathspacing\mathpunctcode\mathopcode    \allsplitstyles\thinmuskip
\setmathspacing\mathpunctcode\mathrelcode   \allsplitstyles\thinmuskip
\setmathspacing\mathpunctcode\mathopencode  \allsplitstyles\thinmuskip
\setmathspacing\mathpunctcode\mathclosecode \allsplitstyles\thinmuskip
\setmathspacing\mathpunctcode\mathpunctcode \allsplitstyles\thinmuskip
\setmathspacing\mathpunctcode\mathinnercode \allsplitstyles\thinmuskip

\setmathspacing\mathinnercode\mathordcode   \allsplitstyles\thinmuskip
\setmathspacing\mathinnercode\mathopcode    \allmathstyles \thinmuskip
\setmathspacing\mathinnercode\mathbincode   \allsplitstyles\medmuskip
\setmathspacing\mathinnercode\mathrelcode   \allsplitstyles\thickmuskip
\setmathspacing\mathinnercode\mathopencode  \allsplitstyles\thinmuskip
\setmathspacing\mathinnercode\mathpunctcode \allsplitstyles\thinmuskip
\setmathspacing\mathinnercode\mathinnercode \allsplitstyles\thinmuskip
```

And because we have a few more atom classes this also needs to happen:

```
\letmathspacing  \mathactivecode   \mathordinarycode
\letmathspacing  \mathvariablecode \mathordinarycode
\letmathspacing  \mathovercode     \mathordinarycode
\letmathspacing  \mathundercode    \mathordinarycode
\letmathspacing  \mathfractioncode \mathordinarycode
\letmathspacing  \mathradicalcode  \mathordinarycode
\letmathspacing  \mathmiddlecode   \mathopencode
\letmathspacing  \mathaccentcode   \mathordinarycode

\letmathatomrule \mathactivecode   \mathordinarycode
\letmathatomrule \mathvariablecode \mathordinarycode
\letmathatomrule \mathovercode     \mathordinarycode
\letmathatomrule \mathundercode    \mathordinarycode
\letmathatomrule \mathfractioncode \mathordinarycode
\letmathatomrule \mathradicalcode  \mathordinarycode
\letmathatomrule \mathmiddlecode   \mathopencode
\letmathatomrule \mathaccentcode   \mathordinarycode
```

With `\resetmathspacing` we get an all-zero state but that might become more refined in the future. What is not clear from the above is that there is also an inheritance mechanism. The three special muskip registers are actually shortcuts so that changing the register value is reflected in the spacing. When a regular muskip value is (verbose or as register) that value is sort of frozen. However, the `\inherited` prefix will turn references to registers and constants into a delayed value: as with the

predefined we now have a more dynamic behavior which means that we can for instance use reserved muskip registers as we can use the predefined. A bonus is that one can also use regular glue or dimensions, just in case one wants the same spacing in all styles (a muskip adapts to the size).

When you look at all of the above you might wonder how users are supposed to deal with math spacing. The answer is that often they can just assume that TEX does the right thing. If something somehow doesn't feel right, looking at solutions by others will probably lead a new user to just copy a trick, like injecting a `\thinmuskip`. But it can be that atoms depend on the already applied (or not) spacing, which in turn depends on values in the atom spacing matrix that probably only a few users have seen. So, in the end it all boils down to trust in the engine and one's eyesight combined with hopefully some consistency in adding space directives and often with TEX it is consistency that makes documents look right. In ConTEXt we have many more classes even if only a few characters fit in, like differential, exponential and imaginary.

Fractions again

We now return to the fraction molecule. With the mechanisms at our disposal we can change the fixed margins to more adaptive ones:

```
\inherited\setmathspacing \mathbinarycode    \mathfractioncode
    \allmathstyles \thickermuskip
\inherited\setmathspacing \mathfractioncode \mathbinarycode
    \allmathstyles \thickermuskip
\nulldelimiterspace\zeropoint
$x + \frac{1}{x+2} + x$
```

Here `\thickermuskip` is defined as `7mu plus 5mu` where the stretch is the same as a `\thickmuskip` and the width `2mu` more. We start out with three variants, where the last two have `\nulldelimiterspace` set to `0pt` and the first one uses the `1.2pt`.

$$x + \frac{1}{x+2} + x$$

$$x + \frac{1}{x+2} + x$$

$$x + \frac{1}{x+2} + x$$

When we now apply the new settings to the last one, and overlay them we get the following output: the first and last case are rather similar which is why this effort was started in the first place.

$$x + \frac{1}{x+2} + x$$

Of course these changes are not upward compatible but as they are tiny they are not that likely to change the number of lines in a paragraph. In display mode changes in horizontal dimensions also have little effect.

Penalties

An inline formula can be broken across lines, and for sure there are places where you don't want to break or prefer to break. In TEX line breaks can be influenced by using penalties. At the outer level of an inline math formula, we can have a specific penalty before and after a binary and/or relation. The defaults are such that there are no penalties set, but most macro packages set the so called `\relpenalty` and `\binoppenalty` (the op in this name does not relate to the operator class) so a value between zero and 1000. In LuaTEX we also have `\pre` variants of these, so we have four penalties that can be set, but that is not enough in our new approach.

These penalties are class bound and don't relate to styles, like atom spacing does. That means that while atom spacing involves $64 \times 64 \times 8$ potential values, an amount that we can manage by using the discussed inheritance. The inheritance takes less values because which store 4 style values per class in one number. For penalties we only need to keep 64×2 in mind, plus a range of inheritance numbers. Therefore it was decided to also generalize penalties so that each class can have them. The magic commands are shown with some useless examples:

```
\letmathparent \mathdigitcode
    \mathbincode   % pre penalty
    \mathbincode   % post penalty
    \mathdigitcode % options
    \mathdigitcode % reserved
```

By default the penalties are on their own, like:

```
\letmathparent \mathdigitcode
  \mathdigitcode % pre penalty
  \mathdigitcode % post penalty
  \mathdigitcode % options
  \mathdigitcode % reserved
```

The options and reserved parent mapping are not (yet) discussed here. Unless values are assigned they are ignored.

```
\setmathprepenalty  \mathordcode 100
\setmathpostpenalty \mathordcode 600
\setmathprepenalty  \mathbincode 200
\setmathpostpenalty \mathbincode 700
\setmathprepenalty  \mathrelcode 300
\setmathpostpenalty \mathrelcode 800
```

As with spacing, when there is no known value, the parent will be consulted. An unset penalty has a value of 10000.

After discussing the implications of inline math crossing lines, Mikael and I decided there can be two solutions. Both can of course be implemented in Lua, but on the other hand, they make good extensions, also because it sort of standardized it. The first advanced control feature tweaks penalties:

```
\mathforwardpenalties  2  200 100
\mathbackwardpenalties 2  100  50
```

This will add 200 and 100 to the first two math related penalties, and 100 and 50 to the last two (watch out: the 100 will be assigned to the last one found, the 50 to the one before it). As with all things penalty and line break related, you need to have

some awareness of how non-linear the badness calculation is as well of the fact that the tolerance and stretch related parameters play a role here.

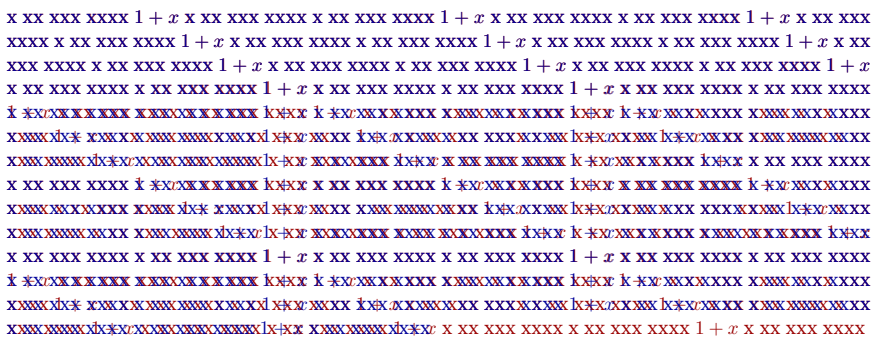
The second tweak is setting `\maththreshold` to some value. When set to for instance `40pt`, formulas that take less space than this will be wrapped in a `\hbox` and thereby will never break across a page.[3] Actually that second tweak has a variant so we have three tweaks! Say that we have this sample formula wrapped in some bogus text and repeat that snippet a lot of times:

```
x xx xxx xxxx $1 + x$ x xx xxx xxxx
```

Now look at the example below. You will notice that the red and blue text have different line breaks. This is because we have given the threshold some stretch and shrink. The red text has a zero threshold so it doesn't do any magic at all, while the second has this setup:

```
\setupmathematics[threshold=medium]
```

That setting set the threshold to `4em plus 0.75em minus 0.50em` and when the formula size exceeds the four quads the line break code will use the real formula width but with the given stretch and shrink. Eventually the calculated size will be used to repackage the formula. In the future we will also provide a way to define slack more relative to the size and/or number of atoms.



Another way to influence line breaks is to use the two inline math related penalties that have been added at Mikael's suggestion:

```
\setupalign[verytolerant]
{\dorecurse{25}{test $\darkred   #1^{#1} + x_{#1}^{#1}$ test }\blank}
{\preinlinepenalty  500 \postinlinepenalty -500
 \dorecurse{25}{test $\darkgreen #1^{#1} + x_{#1}^{#1}$ test }\blank}
{\postinlinepenalty 500 \preinlinepenalty  -500
 \dorecurse{25}{test $\darkblue  #1^{#1} + x_{#1}^{#1}$ test }\blank}
```

To get an example that shows the effect takes a bit of trial and error because TEX does a very good job in line breaking. This is why we've set the tolerance and also use negative penalties.

In addition to the `\mathsurround` (kern) and `\mathsurroundskip` (glue) parameters this is a property of the nodes that mark the beginning and end of an inline math formula.

test $1^{1} + x_{1}^{1}$ test test $2^{2} + x_{2}^{2}$ test test $3^{3} + x_{3}^{3}$ test test $4^{4} + x_{4}^{4}$ test test $5^{5} + x_{5}^{5}$ test test $6^{6} + x_{6}^{6}$ test test $7^{7} + x_{7}^{7}$ test test $8^{8} + x_{8}^{8}$ test test $9^{9} + x_{9}^{9}$ test test $10^{10} + x_{10}^{10}$ test test $11^{11} + x_{11}^{11}$ test test $12^{12} + x_{12}^{12}$ test test $13^{13} + x_{13}^{13}$ test test $14^{14} + x_{14}^{14}$ test test $15^{15} + x_{15}^{15}$ test test $16^{16} + x_{16}^{16}$ test test $17^{17} + x_{17}^{17}$ test test $18^{18} + x_{18}^{18}$ test test $19^{19} + x_{19}^{19}$ test test

3. A future version might inject severe penalties instead, time will learn.

$20^{20}+x_{20}^{20}$ test test $21^{21}+x_{21}^{21}$ test test $22^{22}+x_{22}^{22}$ test test $23^{23}+x_{23}^{23}$ test test $24^{24}+x_{24}^{24}$ test test $25^{25}+x_{25}^{25}$ test

test $1^{1}+x_{1}^{1}$ test test $2^{2}+x_{2}^{2}$ test test $3^{3}+x_{3}^{3}$ test test $4^{4}+x_{4}^{4}$ test test $5^{5}+x_{5}^{5}$ test test $6^{6}+x_{6}^{6}$ test test $7^{7}+x_{7}^{7}$ test test $8^{8}+x_{8}^{8}$ test test $9^{9}+x_{9}^{9}$ test test $10^{10}+x_{10}^{10}$ test test $11^{11}+x_{11}^{11}$ test test $12^{12}+x_{12}^{12}$ test test $13^{13}+x_{13}^{13}$ test test $14^{14}+x_{14}^{14}$ test test $15^{15}+x_{15}^{15}$ test test $16^{16}+x_{16}^{16}$ test test $17^{17}+x_{17}^{17}$ test test $18^{18}+x_{18}^{18}$ test test $19^{19}+x_{19}^{19}$ test test $20^{20}+x_{20}^{20}$ test test $21^{21}+x_{21}^{21}$ test test $22^{22}+x_{22}^{22}$ test test $23^{23}+x_{23}^{23}$ test test $24^{24}+x_{24}^{24}$ test test $25^{25}+x_{25}^{25}$ test

test $1^{1}+x_{1}^{1}$ test test $2^{2}+x_{2}^{2}$ test test $3^{3}+x_{3}^{3}$ test test $4^{4}+x_{4}^{4}$ test test $5^{5}+x_{5}^{5}$ test test $6^{6}+x_{6}^{6}$ test test $7^{7}+x_{7}^{7}$ test test $8^{8}+x_{8}^{8}$ test test $9^{9}+x_{9}^{9}$ test test $10^{10}+x_{10}^{10}$ test test $11^{11}+x_{11}^{11}$ test test $12^{12}+x_{12}^{12}$ test test $13^{13}+x_{13}^{13}$ test test $14^{14}+x_{14}^{14}$ test test $15^{15}+x_{15}^{15}$ test test $16^{16}+x_{16}^{16}$ test test $17^{17}+x_{17}^{17}$ test test $18^{18}+x_{18}^{18}$ test test $19^{19}+x_{19}^{19}$ test test $20^{20}+x_{20}^{20}$ test test $21^{21}+x_{21}^{21}$ test test $22^{22}+x_{22}^{22}$ test test $23^{23}+x_{23}^{23}$ test test $24^{24}+x_{24}^{24}$ test test $25^{25}+x_{25}^{25}$ test

Flattening

The traditional engine has some code for flattening math constructs that in the end are just one character. So in the end, `\tilde{u}` and `\tilde {uu}` become different objects even if both are in fact accents. In fact, when an accent is constructed there is a special code path for single characters so that script placement adapts to the shape of that character.

However because of interaction with primes, which themselves are sort of superscripts and due to the somewhat weird way fonts provide them when it comes to positioning and sizes, in ConTEXt we already are fooling around a bit with these characters. For understandable reasons of memory usage, complexity and eightbitness primes are not a native TEX thing but more something that is handled at the macro level (although not in MkIV and LMTX).

In the end it was script placements on (widely) accented math characters that made us introduce a dedicated `\Umathprime` primitive that adds a prime to a math atom. It permits an uninterrupted treatment of scripts while in the final assembly of the molecule the prime, superscript, subscript and maybe even prescripts that prime gets squeezed in. Because the concept of primes is missing in OpenType math an additional font parameter `PrimeTopRaisePercent` has been introduced as well as an `\Umathprimeraise` primitive. In retrospect I should have done that earlier but one tends to stick to the original as much as possible. However, at some point Mikael and I reached a state where we decided that proper (clean) engine extensions make way more sense than struggling with border cases and explaining users why things are so complicated.

The input `$ X \Uprimescript{'} ^2 _3 $` gives this:

Latin Modern Cambria Pagella Dejavu

With `\tracingmath = 1` this nicely traces as:

```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\mathchar[ord] family "0, character "58
.\superscript
..\mathchar[dig] family "0, character "32
.\subscript
..\mathchar[dig] family "0, character "32
.\primescript
..\mathchar[ord] family "0, character "27
```

Of course this feature can also be used for other prime like ornaments and who knows how it will evolve over time.

You can influence the positioning with `\Umathprimesupshift` which adds some kern between a prime and superscript. The `\Umathextraprimeshift` moves a prime up. The `\Umathprimeraise` is a font parameter that defaults to 25 which means a raise of 25%of the height. These are all (still) experimental parameters.

Fences

Fences can be good for headaches. Because the math that I (or actually my colleague) deal with is mostly school math encoded in presentation MathML (sort or predictable) or some form of sequential ascii based input (often rather messy and therefore unpredictable due to ambiguity) fences are a pain. A TEXie can make sure that left and right fences are matched. A TEXie also knows when something is an inline parenthesis or when a more high level structure is needed, for instance when parentheses have to scale with what they wrap. In that case the `\left` and `\right` mechanism is used. In arbitrary input missing one of those is fatal. Therefore, handling of fences in ConTEXt is one of the more complex sub mechanisms: we not only need to scale when needed, but also catch asymmetrical usage.

A side effect of the encapsulating fencing construct is that it wraps the content in a so called inner (as in `\mathinner`) which means that we get a box, and it is a well known property of boxes that they don't break across lines. With respect to fences, a way out is to not really fence content, but do something like this:

```
\left(\strut\right. x + 1 \left.\strut\right)
```

and hope for the best. Both pairs are coupled in the sense that their sizes will match and the strut is what determines the size. So, as long as there is a proper match of struts all is well, but it is definitely a decent hack. The drawback is in the size of the strut: if a formula needs a higher one, larger struts have to be used. This is why in plain TEX we have these commands:

```
\def\bigl {\mathopen \big } \def\bigm {\mathrel\big } \def\bigr {\mathclose\big }
\def\Bigl {\mathopen \Big } \def\Bigm {\mathrel\Big } \def\Bigr {\mathclose\Big }
\def\biggl{\mathopen \bigg} \def\biggm{\mathrel\bigg} \def\biggr{\mathclose\bigg}
\def\Biggl{\mathopen \Bigg} \def\Biggm{\mathrel\Bigg} \def\Biggr{\mathclose\Bigg}

\def\big #1{{\hbox{$\left#1\vbox to  8.5pt{}\right.\nomathspacing$}}}
\def\Big #1{{\hbox{$\left#1\vbox to 11.5pt{}\right.\nomathspacing$}}}
\def\bigg#1{{\hbox{$\left#1\vbox to 14.5pt{}\right.\nomathspacing$}}}
\def\Bigg#1{{\hbox{$\left#1\vbox to 17.5pt{}\right.\nomathspacing$}}}

\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt} % renamed
```

The middle is kind of interesting because it has relation properties, while the `\middle` introduced in ε-TEX got open properties, but we leave that aside.

In ConTEXt we have plenty of alternatives, including these commands, but they are defined differently. For instance they adapt to the font size. The hard coded point sizes in the plain TEX code relates to the font and steps available in there (either by

next larger or by extensible). The values thereby need to be adapted to the chosen body font as well as the body font size. In MkIV and even better in LMTX we can actually consult the font and get more specific sizes.

But, this section is not about how to get these fixed sizes. Actually, the need to choose explicitly is not what we want, especially because TEX can size delimiters so well. So, take this code snippet:

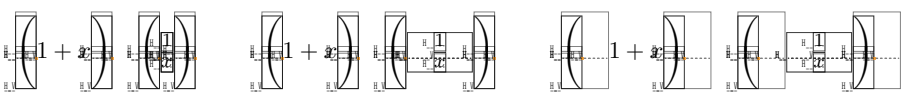
```
$ x = \left( \dorecurse{40}{\frac{x}{x+#1} +} x \right) $
```

When we typeset this inline, as in $x = \left(\frac{x}{x+1}+\frac{x}{x+2}+\frac{x}{x+3}+\frac{x}{x+4}+\frac{x}{x+5}+\frac{x}{x+6}+\frac{x}{x+7}+\frac{x}{x+8}+\frac{x}{x+9}+\frac{x}{x+10}+\frac{x}{x+11}+\frac{x}{x+12}+\frac{x}{x+13}+\frac{x}{x+14}+\frac{x}{x+15}+\frac{x}{x+16}+\frac{x}{x+17}+\frac{x}{x+18}+\frac{x}{x+19}+\frac{x}{x+20}+\frac{x}{x+21}+\frac{x}{x+22}+\frac{x}{x+23}+\frac{x}{x+24}+\frac{x}{x+25}+\frac{x}{x+26}+\frac{x}{x+27}+\frac{x}{x+28}+\frac{x}{x+29}+\frac{x}{x+30}+\frac{x}{x+31}+\frac{x}{x+32}+\frac{x}{x+33}+\frac{x}{x+34}+\frac{x}{x+35}+\frac{x}{x+36}+\frac{x}{x+37}+\frac{x}{x+38}+\frac{x}{x+39}+\frac{x}{x+40}+x\right)$, we get nicely scaled fences but in a way that permits line breaks. The reason is that the engine has been extended with a `fenced` class so that we can recognize later on, when TEX comes to injecting spaces and penalties, that we need to unpack the construct. It is another beneficial side effect of the generalization.

The Plain TEX code can be used to illustrate some of what we discussed before about fractions. In the next code we use excessive delimiter spacing:

```
\def\Bigg#1{%  watch the wrapping in a box
  {%
    \hbox {%
        $\normalleft#1\vbox to 17.5pt{}\normalright.\nomathspacing$%
    }%
  }%
}

\nulldelimiterspace0pt
\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par

\nulldelimiterspace10pt
\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par

\nulldelimiterspace10pt
\def\nomathspacing{\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par
```

This renders as follows. We explicitly set `\nulldelimiterspace` to values because in ConTEXt it is now zero by default.



0pt with reset at end | 10pt with reset at end | 10pt without reset at end

Radicals

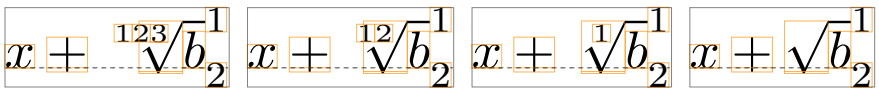
In traditional TEX a radical with degree is defined as macro. That macro does some measurements and typesets the result in four sizes for a choice. The macro typesets the degree in a box that contains the degree as formula. There is a less guesswork going on than with respect to how the radical symbol is shaped but as we're talking plain TEX here it works out okay because the default font is well known.

Radicals are a nice example of a two dimensional 'extender' but only the vertical dimension uses the extension mechanism, which itself operates either horizontally or vertically, although in principle it could go both ways. The horizontal extension is a rule and the fact that the shape is below the baseline (as are other large symbols) will make the rule connect well: the radical shape sticks out a little, so one can think of the height reflecting the rule height.[4] In OpenType fonts there is a parameter and in LuaTEX we use the default rule thickness for traditional fonts, which is correct for Latin Modern. There are more places in the fonts where the design relates to this thickness, for instance fraction rules are supposed to match the minus, but this is a bit erratic if you compare fonts. This is one of the corrections we apply in the goodie files.

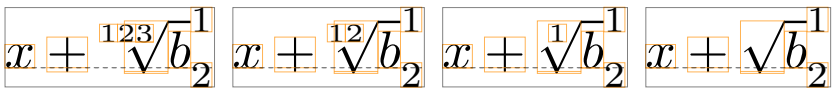
In OpenType the specification of the radical also includes spacing properties of the degree and that is why we have a primitive in LuaTEX that also handles the degree. It is what we used in ConTEXt MkIV. But . . . we actually end up with a situation that compares to the already discussed fraction: there is space added before a radical when there is a degree. However, because we now have a radical atom class, we can avoid using that one and use the new pairwise spacing. Some fuzzy spacing logic in the engine could therefore be removed and we assume that `\Umathradicaldegreebefore` is zero. For the record: the `\Umathradicaldegreeafter` sort of tells how much space there is above the low part of the root, which means that we can compensate for multi-digit degrees.

Zeroing a parameter is something that relates to a font which means that it has to happen for each math font which in turn can mean a family-style combination. In order to avoid that complication (or better: to avoid tracing clutter) we have a way to disable a parameter:

```
\ruledhbox{$x + \sqrt[123]{b}^1_2$}
\ruledhbox{$x + \sqrt[12] {b}^1_2$}
\ruledhbox{$x + \sqrt[1]  {b}^1_2$}
\ruledhbox{$x + \sqrt     {b}^1_2$}
```



```
\setmathignore\Umathradicaldegreebefore 0
```

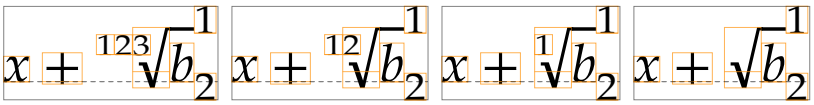


```
\setmathignore\Umathradicaldegreebefore 1
```

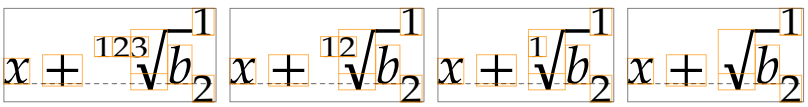
Latin Modern

One problem with these spacing parameters is that they are inconsistent across fonts. The Latin Modern has a rather large space before the degree, while Cambria and Pagella have little. That means that when you prototype a mechanism the chosen solution can look great but not so much when at some point you use another font.

4. When you zoom in you will notice that this is not always optimal because of the way the slope touched the rule.



\setmathignore\Umathradicaldegreebefore 0



\setmathignore\Umathradicaldegreebefore 1

Cambria

More fences

One of the reasons why the MkII and MkIV fence related mechanism is somewhat complex is that we want a clean solution for filtering fences like parenthesis by size, something that in the traditional happens via a fake fence pair that encapsulates a strut of a certain size. In LMTX we use the same approach but have made the sequence more configurable. In practice that means that the values 1 up to 4 are just that but for some fonts we use the sequence `1 3 5 7`. There was no need to adapt the engine as it already worked quite well.

Integrals

The Latin Modern fonts have only one size of big operators and one reason can be that there is no need for more. Another reason can be that there was just no space in the font. However, an OpenType font has plenty slots available and the reference font Cambria has integral signs in sizes as well as extensibles.

In LuaTEX we already have generic vertical extensibles but that only works well with specified sizes. And, cheating with delimiters has the side effect that we get the wrong spacing. In LuaMetaTEX however we have ways to adapt the size to what came or what comes. In fact, it is a mechanism that is available for any atom that we support. However, it doesn't play well with script and this whole `\limits` and `\nolimits` is a bit clumsy so Mikael and I decided that different route had to be followed. For adaptive large operators we provide this interface:

```
$ x + \integral [color=darkred,top={t},bottom={b}] {\frac{1}{x}} = 10 $

$ x + \startintegral [color=darkblue,top={t},bottom={b}]
                                          \frac{1}{x} \stopintegral = 10 $

$ x + \startintegral [color=darkgreen,top={t},bottom={b},method=vertical]
                                           \frac{1}{x} \stopintegral= 10 $
```

This text is not about the user interface so we won't discuss how to define additional large operators using one-liners.

$$x + \int_b^t \frac{1}{x} = 10 \qquad x + \int_b^t \frac{1}{x} = 10 \qquad x + \int_b^t \frac{1}{x} = 10$$

The low level LuaMetaTEX implementation handles this input:

```
\Uoperator          \Udelimiter "0 \fam "222B {top} {bottom} {...}
\Uoperator   limits \Udelimiter "0 \fam "222B {top} {bottom} {...}
\Uoperator nolimits \Udelimiter "0 \fam "222B {top} {bottom} {...}
```

plus the usual keywords that fenced accept, because after all, this is just a special case of fencing.

Currently these special left operators are implemented as a special case of fences because that mechanism does the scaling. It means that we need a (bogus) right fence, or need to brace the content (basically create an atom). When no right fence is found one is added automatically. Because there is no real fencing, right fences are removed when processing takes place. When you specify a `class` that one will be used for the left and right spacing, otherwise we have open/close spacing.

Going details

When the next feature was explored Mikael tagged it as math micro typography and the reason is that you need not only to set up the engine for it but also need to be aware of this kind of spacing. Because we wanted to get rid of this script spacing that the font imposes we configured ConTEXt with:

```
\setmathignore\Umathspacebeforescript\plusone
\setmathignore\Umathspaceafterscript \plusone
```

This basically nils all these tiny spaces. But the latest configuration has this instead:

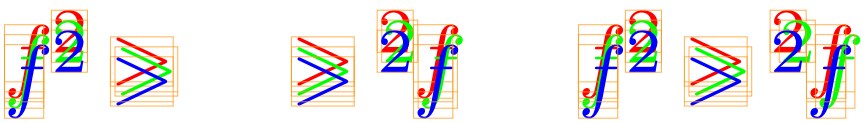
```
% \setmathignore \Umathspacebeforescript\zerocount % default
% \setmathignore \Umathspaceafterscript \zerocount % default

\mathslackmode \plusone

\setmathoptions\mathopcode      \plusthree
\setmathoptions\mathbinarycode  \plusthree
\setmathoptions\mathrelationcode\plusthree
\setmathoptions\mathopencode    \plusthree
\setmathoptions\mathclosecode   \plusthree
\setmathoptions\mathpunctcode   \plusthree
```

This tells the engine to convert these spaces into what we call slack: disposable kerns at the edges. But it also converts these kerns into a glue component when possible. As with all these extensions it complicates the machinery but users will never see that. Now, the last six lines do the magic that made us return to honoring the spaces: we can tell the engine to ignore this slack when there are specific classes at the edges. These options are a bitset and `1` means “no slack left” and `2` means “no slack right” so `3` sets both.

```
\def\TestSlack#1%
  {\vbox\bgroup
      \mathslackmode\zerocount
      \hbox\bgroup
         \setmathignore\Umathspacebeforescript\zerocount
         \setmathignore\Umathspaceafterscript \zerocount
         #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\red
        \setmathignore\Umathspacebeforescript\plusone
        \setmathignore\Umathspaceafterscript \plusone
         #1
      \egroup
   \egroup}

\startcombination[nx=3]
    {\showglyphs\TestSlack{$f^2 >     $}} {}
    {\showglyphs\TestSlack{$    > f^^2$}} {}
    {\showglyphs\TestSlack{$f^2 > f^^2$}} {}
\stopcombination
```

Because this overall removal of slack is not granular enough a while later we introduced a way to set this per class, as is demonstrated in the following example.

```
\def\TestSlack#1%
  {\vbox\bgroup
      \mathslackmode\plusone
      \hbox\bgroup\red
        \setmathignore\Umathspacebeforescript\zerocount
        \setmathignore\Umathspaceafterscript \zerocount
        #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\green
        \setmathoptions\mathrelationcode \zerocount
        #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\blue
        \setmathoptions\mathrelationcode \plusthree
        #1
      \egroup
   \egroup}

\startcombination[nx=3]
    {\showglyphs\TestSlack{$f^2 >     $}} {}
    {\showglyphs\TestSlack{$    > f^^2$}} {}
    {\showglyphs\TestSlack{$f^2 > f^^2$}} {}
\stopcombination
```



Of course we need to experiment a lot with real documents and it might take a while before all this is stable (in the engine and in ConTEXt). And as we don't need to conform to the decades old golden TEX math standards we have some degrees of freedom in this: for Mikael and me it is pretty much a visual thing where we look closely at large samples. Of course in practice details get lost when we print at 10 point but that doesn't mean we can't provide the best experience.[5]

Ghosts

As plain TEX has macros like `\vphantom` you also find them in macro packages that came later. These create a boxes that have their content removed after the dimensions are set. They take space and are invisible but there's also nothing there.

A variant in the upgraded math machinery are ghosts: these are visible in the sense that they show up but ignored when it comes to spacing. Here is an example. The option bit set here tells the engine that we ghost at the right, so we have ghosts around the relation (it controls where the spacing ends up).

5. Whenever I look at (my) old (math) school books I realize that Don Knuth had very good reasons to come up with TEX and, it being hard to beat, TEX still sets the standard!

```
$
  x
  \mathatom class \mathghostcode                  {!!}
  >
  \mathatom class \mathghostcode options "00000020 {!!}
  1
  \quad
  x
  \mathatom class \mathghostcode              {\hbox{\smallinfofont ord}}
  >
  \mathatom class \mathghostcode options "00000020
                                              {\hbox{\smallinfofont dig}}
  1
$
```

You never know when this comes in handy but it fits in the new, more granular approach to spacing. The code above shows that it's just a class, this time with number 17.

Struts

In order to get consistent spacing the ConTEXt macro package makes extensive use of struts in text mode as well as math mode. The normal way to implement that is either an empty box or a zero width rule, both with a specifically set height and depth. In ConTEXt MkII and MkIV (and for a long time in LMTX too) they were rules so that we could visualize them: they get some width and kerns around them to compensate for that.

In order to not let them interfere with spacing we could wrap them into a ghost atom but it is kind of ugly. Anyway, before we had these ghost atoms an alternative to struts was already implemented: a special kind of rule. The reason is that I wanted a cleaner and more predictable way to adapt struts to the math style uses and sometimes predicting that is fragile. What we want is a delayed assignment of dimensions.

We have two solutions. The first one uses two math parameters that themselves adapt to the style, as do other parameters: `\Umathruleheight` and `\Umathruledepth`. The other solution relates a font (or family) and character with the strut rule which is then used as measure for the height and depth. Just for the record: this also works in text mode, which is why a recent LMTX also does use that for struts now. The optional visualization is just part of the regular visualization mechanism in ConTEXt which already had provisions for struts. A side effect of this is that the rule primitives now accept three more keywords: `font`, `fam` and `char`, in addition to the already present traditional ones `width`, `height` and `depth`, the (backend) margin ones `left` (or `top`) and `right` (or `bottom`) options, as well as `xoffset` and `yoffset`). The command that creates a rule with subtype `strut` is simply `\srule`. Because struts are rather macro package specific I leave it to this.

One positive side effect is that we could simplify the ConTEXt fraction mechanism a bit. Over time control over the (font driven) gaps was introduced but that is not really needed because we zero the gaps anyway. There was also a tolerance mechanism which again was not used. However, for skewed fractions we do use the new tolerance mechanism as well as gap control.

Atoms

Now that we have generic atoms (`\mathatom`) another, sometimes confusing aspect of the math parsing can be solved. Take this:

```
\def\MyBin{\mathbin{\tt mybin}}
$ x ^ \MyBin _ \MyBin $
```

The parser just doesn't like that which means that one has to use

```
\def\MyBin{\mathbin{\tt mybin}}
$ x ^ {\MyBin} _ {\MyBin} $
```

or:

```
\def\MyBin{{\mathbin{\tt mybin}}}
$ x ^ \MyBin _ \MyBin $
```

But the later has side effects: it creates a list that can influence spacing. It is for that reason that we do accept atoms where they were not accepted before. Of course that itself can have side effects but at least we don't get an error message. It fits well into the additional (user) classes model. And, given that in ConTEXt the `\frac` command is actually wrapped as `\mathfrac` the next will work too:

```
$ x^\frac{1}{2} + x^{\frac{1}{2}} $
```

but in practice you should probably use the braced version here for clarity.

The `vcenter` primitive

Traditionally this primitive is bound to math but it had already been adapted to also work in text mode. As part of the upgrade of math we can now also pass all the options that normal boxed take and we can also cheat with the axis. Here is an example:

```
\def\TEST{\hbox\bgroup
    \darkred   \vrule width  2pt height 4pt
    \darkgreen \vrule width 10pt depth  2pt
\egroup}
$
    x  - \mathatom \mathvcentercode {!!!} -
    + \ruledvcenter                          {\TEST}
    + \ruledvcenter                          {\TEST}
    + \ruledvcenter axis 1                   {\TEST}
    + \ruledvcenter xoffset  2pt yoffset  2pt {\TEST}
    + \ruledvcenter xoffset -2pt yoffset -2pt {\TEST}
    + x
$
```

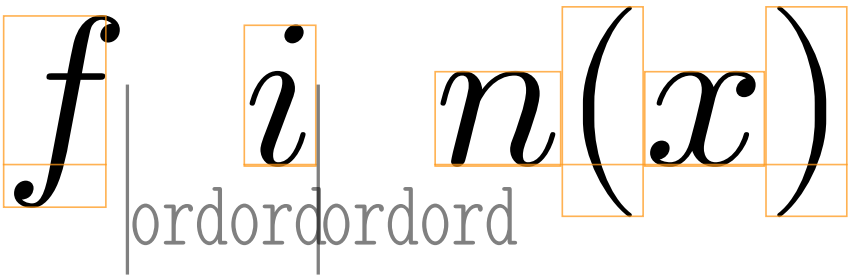
There was already a vcenter class available before we did this:

Text

Sometimes you want text in math, for instance `sin` or `cos` but text in math is not really text:

```
$\setmathspacing\mathordinarycode\mathordinarycode\textstyle 10mu fin(x)$
```

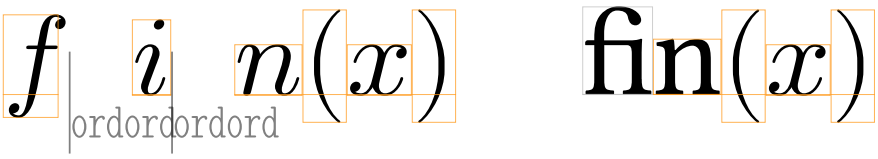
The result demonstrates that what looks like a word actually becomes three math atoms:



Okay, so how about then wrapping it into a text box:

```
$
    \setmathspacing\mathordinarycode\mathordinarycode\textstyle 10mu
    fin(x) \quad \hbox{fin}(x)
$
```

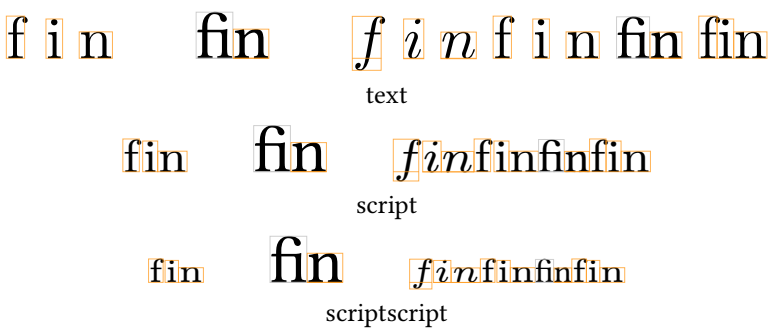
Here we get:



We even get a ligature which might be an indication that we're not using a math font which indeed is the case: the box is typeset in the regular text font.

```
\def\Test#1%
  {\setmathspacing\mathordinarycode\mathordinarycode\textstyle 5mu
   $\showglyphs
   #1% style
   {\tf fin} \quad
   \hbox{fin} \quad
   \mathatom class \mathordinarycode textfont {fin}
   \mathatom class \mathordinarycode textfont {\tf fin}
   \mathatom class \mathordinarycode textfont {\hbox{fin}}
   \mathatom class \mathordinarycode mathfont {\hbox{fin}}
   $}
```

When we feed this macro with the `\textstyle`, `\scriptstyle` and `\scriptscriptstyle` we get:



Here you see a new atom option action: `textfont` which does as much as setting the font to the current family font and the size to the one used in the style. For the

record: you only get ligatures when they are configured and provided by the font (and as math is a script itself it is unlikely to work).[6]

Tracing

I won't discuss the tracing features in ConTEXt here but for sure the visualizer helps a lot in figuring out all this. In LuaMetaTEX we carry a bit more information with the resulting nodes so we can provide more details, for instance about the applied spacing and penalties. Some is shown in the examples. A more recent tracing feature is this:

```
\tracingmath   1
\tracingonline 1
$
    \mathord (
    \mathord {(}
    \mathord \Udelimiter"4 0 `(
    \Udelimiter"4 0 `(
$
```

That gives us on the console (the dots represent detailed attribute info that we omit here):

```
7:3: > \inlinemath=
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathlist
7:3: ...\noad[open][...]
7:3: ....\nucleus
7:3: .....\mathchar[open] family "0, character "28
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
7:3: \noad[open][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
```

A tracing level of 2 will spit out some information about applied spacing and penalties between atoms (when set) and level 3 will show the math list before the first and second pass (a mix of nodes and noads) we well as the result (nodes) plus return some details about rules, spacing and penalties applied.

Is there more?

The engine already provides the option to circumvent the side effect of a change in a parameter acting sort of global: the last value given is also the one that a second pass starts with. The `\frozen` prefix will turn settings into local ones but that's another (already old) topic. There are many such improvements and options not mentioned here but you can find them mentioned and explained in older development stories. A lot has been around for a while but not been applied in ConTEXt yet.

When TEX was written one important property (likely related to memory consumption) is that node lists have only forward pointers. That means that the state of preceding material has to be kept track of: there is no going (or looking) back. In

6. The existing mechanisms in ConTEXt already dealt with this but it is nevertheless nice to have it as a clean engine feature.

LuaTEX we have double linked lists so in principle we can try to be more clever but so far I decided not to touch the math machinery in that way. But who knows what comes next.

Those italics

Right from the start of LuaTEX it became clear that the fact that TEX assumes the actual width of glyphs to be incremented by the italic correction that then selectively is removed has been an issue. It made for successive attempts to improve spacing in ConTEXt by providing pseudo features. But, when we moved from assembled Unicode math fonts to 'real' ones that became messy: what trick to apply when and even worse where? In the end there are only a very few shapes that actually are affected in the sense that when we don't deal with them it looks bad. It also happens that one of those shapes is the italic 'f', a letter that is used frequently in math. It might even be safe to say that the simple fact that the math italic f has this excessively wrong width and thereby pretty large italic correction is the cause of many problems.

In the LMTX approach Mikael and I settled on patching shapes in the so called font goodie files, aka `lfg` files and only a handful of entries needed a treatment. This makes a good case for removing the traditional font code path from LuaMetaTEX.

`modern`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`cambria`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`pagella`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`termes`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`bonum`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

One of the other very sloped symbol is the integral, although most fonts have them more upright than tex has. Of course there are many variants of these integrals in a math font. Here we also have some font parameters that we can tune, which is what we do.

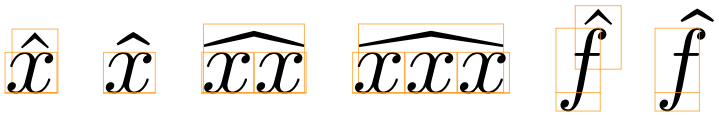
Accents

Accents are common in languages other than English and it's English that TEX was made for. Although the seven bit variant became eight bit handling accents never was sophisticated and one of the main reasons is of course that one could use pre-built composed characters. The OpenType format brought proper anchoring (aka marks) to font formats and when LuaTEX deals with text those kick in. In OpenType math however, anchoring is kind of limited to the top position only. Because the TEX Gyre fonts are based on traditional TEX fonts, their accents have not become better suited.

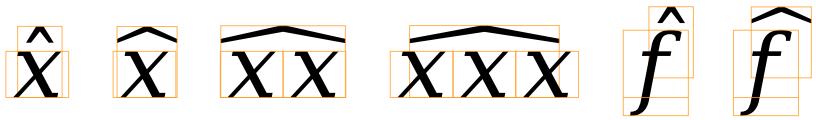
```
$\hat{x} \enspace \widehat{x} \enspace \widehat{xx} \enspace \widehat{xxx}
 \enspace \hat{f} \enspace \widehat{f} $
```

When looking at examples you need to be aware of the fact hat fonts can have been adapted in the goodie files.[7] So, for instance bounding boxes and such can differ from the original. Anyway, the previous code in Cambria looks as follows.

With Latin Modern we get:



And Dejavu comes out as:



As you can see there are some differences. In for instance Latin Modern the shape of the hat and smallest wide hat are different and the first wide one has zero dimensions combined with a negative anchor. When an accented character is followed by a superscript or prime the italic correction of the base kicks in but that cannot be enough to not let this small wide hat overflow into the script. We could compensate for it but then we need to know the dimensions. Of course we can consult the bounding box but it makes no sense to let heuristics enter the machinery here while we're in the process generalization. One option is to have two extra parameters that can be used when the width of the accent comes close to the width of the base (we then assume that zero accent width means that it has base width) we add an additional kern. In the end we settled for a (semi automatic) correction option in the goodie files.

There are actually three categories of extensible accents to consider: those that resemble the ones used in text (like tildes and hats), those wrapping something (like braces and bracket but also arrows) and rules (that in traditional TEX indeed are rules). In ConTEXt we have different interfaces for each of these in order to have a more extensive control. The text related ones are the simplest and closest to what the engine supports out of the box but even there we use tweaked glyphs to get better spacing because (of course) fonts have different and inconsistent spacing in the boundingbox above and below the real shape. This is again some tweak that we moved from being *automatic* to being *under goodie file control*. But this is all too ConTEXt specific to discuss here in more detail.

Decision time

After lots of tests Mikael and I came to the conclusion that we're facing the following situation. When typesetting math most single characters are italic and we already knew from the start of the LuaTEX project that the italics shapes are problematic when it comes to typesetting math. But it looks like even some upright characters can have italic correction: in TexGyreBonum for instance the bold upright `f` has italic correction, probably because it then can (somehow) kern with a following `i`. It anyhow assumes no italic correction to be applied between these characters.

7. Extreme examples can be found for Lucida Bright where we not only have to fix the extensible parts of horizontal braces but also have to provide horizontal brackets.

In the end the mixed math font model model got more and more stressed so one decision was to simply assume fonts to be used that are either Cambria like OpenType, or mostly traditional in metrics, or a hybrid of both. It then made more sense to change the engine control options that we have into ones that simply enable certain code paths, independent of the fact if a font is OpenType or not. It then become a bit "crap in, crap out", but because we already tweak fonts in the goodie files it's quite okay. Some fonts have bad metrics anyway or miss characters and it makes no sense to support abandoned fonts either. Also, when a traditional font is assembled one can set up the engine with different flags and we can deal with it as we wish. In the end it is all up to the macro package to configure things right, which is what we tried to do for months when rooting out all the artifacts that fonts bring.[8]

That said, the reason why some (fuzzy) mixed model works out okay (also in LuaTEX) is that proper OpenType fonts use staircase kerns instead of italic correction. They also have no ligatures and kerns. We also suspect that not that much attention is paid to the rendering. It's a bit like these "How many f's do you count in this sentence?" tests where people tend to overlook `of`, `if` and similar short words. Mathematicians loves `f`'s but probably also overlook the occasionally weird spacing and kerning.

A side effect is that mixing OpenType and traditional fonts is also no longer assumed which in turn made a few (newly introduced) state variables obsolete. Once everything is stable (including extensions discussed before) some further cleanup can happen. Another side effect is that one needs to tell the engine what to apply and where, like this:

```
\mathfontcontrol\numexpr \zerocount
   +\overrulemathcontrolcode
   +\underrulemathcontrolcode
   +\fractionrulemathcontrolcode
   +\radicalrulemathcontrolcode
   +\accentskewhalfmathcontrolcode
   +\accentskewapplymathcontrolcode
 % + checkligatureandkernmathcontrolcode
   +\applyverticalitalickernmathcontrolcode
   +\applyordinaryitalickernmathcontrolcode
   +\staircasekernmathcontrolcode
 % +\applycharitalickernmathcontrolcode
 % +\reboxcharitalickernmathcontrolcode
   +\applyboxeditalickernmathcontrolcode
   +\applytextitalickernmathcontrolcode
   +\checktextitalickernmathcontrolcode
 % +\checkspaceitalickernmathcontrolcode
   +\applyscriptitalickernmathcontrolcode
   +\italicshapekernmathcontrolcode
\relax
```

There might be more control options (also for tracing purposes) and some of the symbolic (ConTEXt) names might change for the better. As usual it will take some years before all is stable but because most users use the latest greatest version it will be tested well.

After this was decided and effective I also decided to drop the mapping from traditional font parameters to the OpenType derives engine ones: we now assume that the latter ones are set. After all, we already did that in ConTEXt for the virtual assemblies that we started out with in the beginning of LuaTEX and MkIV.

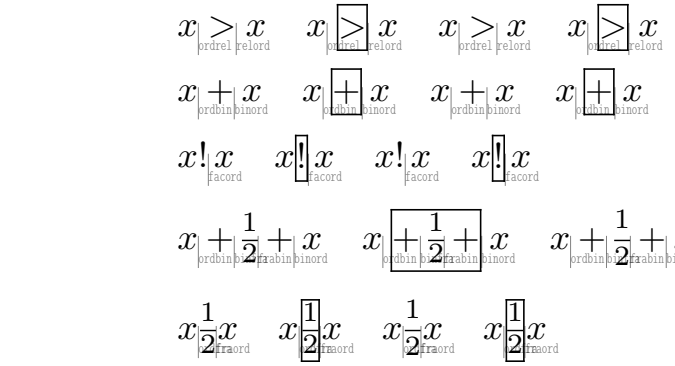
8. In previous versions one could configure this per font but that has been dropped.

Dirty tricks

Once you start playing with edge cases you also start wondering if some otherwise complex things can be done easier. The next macro brings together a couple of features discussed in previous sections. It also uses two state variables: `\lastleftclass` and `\lastrightclass` that hold the most recent edge classes.

```
\tolerant\permanent\protected\def\NiceHack[#1]#:#2% special arg. parsing
  {\begingroup
   \setmathatomrule
     \mathbegincode\mathbincode % context constants
     \allmathstyles
     \mathbegincode\mathbincode
   \normalexpanded
     {\setbox\scratchbox\hpack
        ymove \Umathaxis\Ustyle\mathstyle % an additional box property
        \bgroup
          \framed % a context macro
            [location=middle,#1]
            {$\Ustyle\mathstyle#2$}%
        \egroup}%
   \mathatom
     class 32 % an unused class
     \ifnum\lastleftclass <\zerocount\else leftclass  \lastleftclass\fi
     \ifnum\lastrightclass<\zerocount\else rightclass \lastrightclass\fi
     \bgroup
       \box\scratchbox
     \egroup
   \endgroup}

\def\MyTest#1%
  {$              x #1                          x $\quad
   $              x \NiceHack[offset=0pt]{#1} x $\quad
   $\displaystyle x #1                          x $\quad
   $\displaystyle x \NiceHack[offset=0pt]{#1} x $}

\scale[scale=1500]{\MyTest{>}}            \blank
\scale[scale=1500]{\MyTest{+}}            \blank
\scale[scale=1500]{\MyTest{!}}            \blank
\scale[scale=1500]{\MyTest{+\frac{1}{2}+}}\blank
\scale[scale=1500]{\MyTest{\frac{1}{2}}}  \blank
```

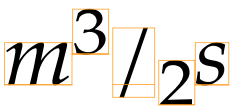
Of course this is not code you immediately come up with after reading this text, also because you need to know a bit of ConTEXt.



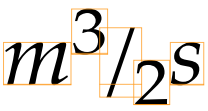
There are a few control options, like `\noatomruling` that can be used to prevent rules being applied to the next atom. We can use these in order to achieve more advanced alignment results, but discussing math alignments would demand many more pages than make sense here.

Tuned kerning

The ConTEXt distribution has dedicated code for typesetting units that dates back to the mid nineties of the previous century but was (code wise) upgraded from MkII to MkIV which made it end up in the physics name space. There is not much reason to redo that code but when we talk new spacing classes it might make sense at some point to see if we can use less code for spacing by using a 'unit' class. When Mikael pointed out that, for instance in Pagella:



doesn't space well the obvious answer is: use the units mechanism because this kind of rendering was why it was made in the first place. However, the question is of course, can we do better anyway. The chosen solution uses a combination of class options and tweaked shape kerning:



An example of a class setup in ConTEXt is:

```
\setmathoptions\mathdivisioncode\numexpr
    \nopreslackclassoptioncode     +\nopostslackclassoptioncode
   +\lefttopkernclassoptioncode    +\righttopkernclassoptioncode
   +\leftbottomkernclassoptioncode +\rightbottomkernclassoptioncode
\relax
```

and, although we don't go into the details of tweaking here, this is the kind if code you will find in the goodie file:

```
{
    tweak = "kerns",
    list  = {
        [0x2F] = {
            topleft     = -0.3,
            bottomright =  0.2,
       }
    }
}
```

where the numbers are a percentage of the width. This specification translates in a math staircase kerning recipe.

More font tweaks

Once you start looking into the details of these fonts you are likely to notice more issues. For instance, in the nice looking Lucida math fonts the relations have inconsistent widths and even shapes. This can partially be corrected by using a stylistic alternate but even that forced us to come up with a mechanism to selectively replace 'bad' shapes because there is not that much granularity in the alternates. And

once we looked at these alternates we noticed that the definition of of script versus calligraphic is also somewhat fuzzy and font dependent. That made for yet another tweak where we can swap alphabets and let the math machinery choose the expected shape. In Unicode this is handled by variant selectors which is rather cumbersome. Because these two styles are used mixed in the same document, a proper additional alphabet would have made more sense. As we already support variant selectors it was no big deal to combine that mechanism with a variant selector features over a range of calligraphic or script characters, which indeed is what mathematicians use (Mikael can be very convincing). With this kind of tweaks the engine doesn't really play a role: we always could and did deal with it. It's just that upgrading the engine made us look again at this.

Final words

One can argue that all these new features can make a document look better. But you only have to look at what Don Knuth produces himself to see that one always could do a good job with TEX, although maybe at the cost of some extra spacing directives. It is the fact that OpenType showed up as well as many more math fonts, all with their own (sometimes surprising) special effects, that made us adapt the engine. Of course there are also new possibilities that permit better and more robust macro support. The TEXbook has a chapter on "the fine points of mathematics typesetting" for a reason.

There has never been an excuse to produce bad looking documents. It is all about care. For sure there is a category of users who are forced to use TEX, so they are excused. There are also those who have no eye for typography and rely on the macro package, so there we can to some extent blame the authors of those packages. And there are of course the sloppy users, those who don't enter a revision loop at all. They could as well use any system that in some way can handle math. One can also wonder in what way massive remote editing as well as collaborative working on documents make things better. It probably becomes less personal. At meetings and platforms TEX users like to bash the alternatives but in the end they are part of the same landscape and when it comes to math they dominate. Maybe there is less to brag about then we like: just do your thing and try to do it as good as possible. Rely on your eyes and pay attention to the details, which is possible because the engine provided the means. The previous text shows a few things to pay attention to.

Once all the basics that have to do with proper dimensions, spacing, penalties and logic are dealt with, we will move on to the more high level constructs. So, expect more.

Hans Hagen & Mikael Sundqvist