Unicode Math in ConTEXt

Abstract
This article is complementary to Taco Hoekwater's article about the upgrade of the math subsystem in LuaTEX. In parallel (also because we needed a testbed) the math subsystem of ConTEXt has been upgraded. In this article I will describe how we deal with Unicode math using the regular Latin Modern and TEXGyre fonts and how we were able to clean up some of the more nasty aspects of math.

Introduction

The LuaTEX project entered a new stage when end of 2008 and beginning of 2009 math got opened up. Although TEX can handle math pretty well we had a few wishes that we hoped to fulfill in the process. TEX's math machinery is a rather independent subsystem. This is reflected in the fact that after parsing there is an intermediate list of so called noads (math elements), which then gets converted into a node list (glyphs, kerns, penalties, glue and more). This conversion can be intercepted by a callback and a macro package can do whatever it likes with the list of noads as long as it returns a proper list.

Of course ConTEXt does support math and that is visible in its code base:

- Due to the fact that we need to be able to switch to alternative styles the font system is quite complex and in ConTEXt MkII math font definitions (and changes) are good for 50% of the time involved. In MkIV we can use a more efficient model.
- Because some usage of ConTEXt demands the mix of several completely different encoded math fonts, there is a dedicated math encoding subsystem in MkII. In MkIV we will use Unicode exclusively.
- Some constructs (and symbols) are implemented in a way that we find suboptimal. In the perspective of Unicode in MkIV we aim at all symbols being real characters. This is possible because all important constructs (like roots, accents and delimiters) are supported by the engine.
- In order to fit vertical spacing around math (think for instance of typesetting on a grid) in MkII we have ended up with rather messy and suboptimal code. (This is because spacing before and after formulas has to cooperate with spacing of structural components that surround it.) The expectation is that we can improve that.

In the following sections I will discuss a few of the implementation details of the font related issues in MkIV. Of course a few years from now the actual solutions we implemented might look different but the principles remain the same. Also, as with other components of LuaTEX Taco and I worked in parallel on the code and its usage, which made the tasks easier for both of us.

Transition

In TEX, math typesetting uses a special concept called families. Each math component (number, letter, symbol, et cetera) is member of a family. Because we have three sizes (text, script and scriptscript) this results in a family–size matrix of defined fonts. The number of glyphs in a font was limited to 256, which meant that we had quite some font definitions. The minimum number of families was 4 (roman, italic, symbol, and extension) but in practice several more could be active (sans, bold, mono-spaced, more symbols, et cetera) for specific alphabets or extra symbols (for instance ams set A and B). The total number of families in traditional TEX is limited to 16, and one easily hits this maximum. In that case, some 16 times 3 fonts are defined for one size of which in practice only a few are really used in the typesetting.

A potential source of confusion is bold math. Bold in math can either mean having some bold letters, or having the whole formula in bold. In practice this means that for a complete bold formula one has to define the whole lot using bold fonts. A complication is that the math symbols are kind of bound to families and so we end up with either redefining symbols, or reusing the families (which is easier and faster). In any case there is a performance issue involved due to the rather massive switch from normal to bold.

In Unicode all alphabets that make sense, as well as all math symbols are part of the definition, although unfortunately some alphabets have their letters spread over the Unicode vector and not in a range (like blackboard). This forces all applications that want to support math to implement similar hacks to deal with it.

In MkIV we will assume that we have Unicode aware math fonts, like OpenType. The font that sets the standard is Microsoft Cambria. The upcoming (I'm writing this in January 2009) TEXGyre fonts will be compliant to this standard but they're not yet there and so we have a problem. The way out is to define virtual fonts and now that LuaTEX math is extended to cover all of Unicode, as well as provides access to the (intermediate) math lists, this has become feasible. This also permits us to test LuaTEX with both Cambria and Latin Modern Virtual Math.

The advantage is that we can stick to just one family for all shapes which simplifies the underlying TEX code enormously. First of all we need to define way less fonts (which is partially compensated by loading them as part of the virtual font) and all math aspects can now be dealt with using the character data tables.

One tricky aspect of the new approach is that the Latin Modern fonts have design sizes, so we have to define several virtual fonts. On the other hand, fonts like Cambria have alternative script and scriptscript shapes which is controlled by the `ssty` feature, a gsub alternate that provides some alternative sizes for a couple of hundred characters that matter.

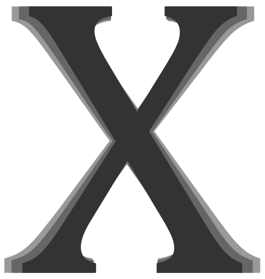
text	`lmmi12 at 12pt`	`cambria at 12pt with ssty=no`
script	`lmmi8 at 8pt`	`cambria at 8pt with ssty=1`
scriptscript	`lmmi6 at 6pt`	`cambria at 6pt with ssty=2`

So Cambria not so much has design sizes but shapes optimized relative to the text variant: in the following example we see text in red, script in green and scriptscript in blue.

```
\definefontfeature[math][analyze=false,script=math,language=dflt]

\definefontfeature[text]        [math][ssty=no]
\definefontfeature[script]      [math][ssty=1]
\definefontfeature[scriptscript][math][ssty=2]
```

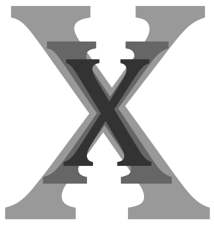
Let us first look at Cambria:

```
\startoverlay
    {\definedfont[name:cambriamath*scriptscript at 150pt]\mkblue  X}
    {\definedfont[name:cambriamath*script       at 150pt]\mkgreen X}
    {\definedfont[name:cambriamath*text         at 150pt]\mkred   X}
\stopoverlay
```



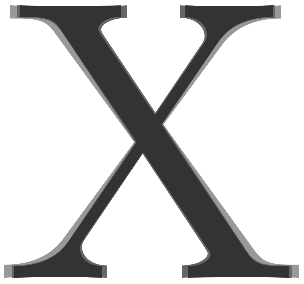
When we compare them scaled down as happens in real script and scriptscript we get:

```
\startoverlay
    {\definedfont[name:cambriamath*scriptscript at 120pt]\mkblue  X}
    {\definedfont[name:cambriamath*script       at  80pt]\mkgreen X}
    {\definedfont[name:cambriamath*text         at  60pt]\mkred   X}
\stopoverlay
```



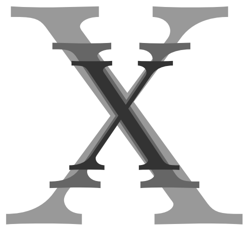
Next we see (scaled) Latin Modern:

```
\startoverlay
    {\definedfont[LMRoman8-Regular  at 150pt]\mkblue  X}
    {\definedfont[LMRoman10-Regular at 150pt]\mkgreen X}
    {\definedfont[LMRoman12-Regular at 150pt]\mkred   X}
\stopoverlay
```



In practice we will see:

```
\startoverlay
    {\definedfont[LMRoman8-Regular  at 120pt]\mkblue  X}
    {\definedfont[LMRoman10-Regular at  80pt]\mkgreen X}
    {\definedfont[LMRoman12-Regular at  60pt]\mkred   X}
\stopoverlay
```



Both methods probably work out well, although you need to keep in mind that the OpenType `ssty` feature is not so much a design size related feature.

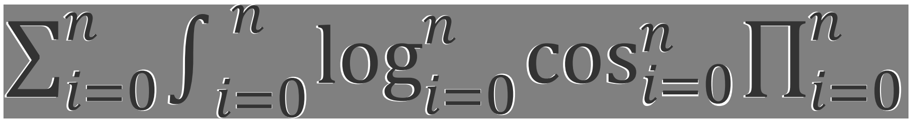
An OpenType font can have a specification for the script and scriptscript size. By default we listen to this specification instead of the one imposed by the bodyfont environment. When you turn on tracing

```
\enabletrackers[otf.math]
```

you will get messages like:

```
asked scriptscript size: 458752, used: 471859.2 (102.86 %)
asked script size: 589824, used: 574095.36 (97.33 %)
```

The differences between the defaults and the font recommendations are not that large so by default we listen to the font specification.



In this overlay the white text is scaled according to the specification in the font, while the black text is scaled according to the bodyfont environment (12/7/5 points).

Going virtual

The number of math fonts (used) in the TEX community is relatively small and of those only Latin Modern (which builds upon Computer Modern) has design sizes. This means that the amount of Unicode compliant virtual math fonts that we have to make is not that large. We could have used an already present virtual composition mechanism but instead we made a handy helper function that does a more efficient job. This means that a definition looks (a bit simplified) as follows:

```
mathematics.make_font ( "lmroman10-math", {
  { name="lmroman10-regular", features="virtualmath", main=true },
  { name="lmmi10", vector="tex-mi", skewchar=0x7F },
  { name="lmsy10", vector="tex-sy", skewchar=0x30, parameters=true } ,
  { name="lmex10", vector="tex-ex", extension=true } ,
  { name="msam10", vector="tex-ma" },
  { name="msbm10", vector="tex-mb" },
  { name="lmroman10-bold", "tex-bf" } ,
  { name="lmmib10", vector="tex-bi", skewchar=0x7F } ,
  { name="lmsans10-regular", vector="tex-ss", optional=true },
  { name="lmmono10-regular", vector="tex-tt", optional=true },
} )
```

For the TEXGyre Pagella it looks this way:

```
mathematics.make_font ( "px-math", {
  { name="texgyrepagella-regular", features="virtualmath", main=true },
  { name="pxr", vector="tex-mr" } ,
  { name="pxmi", vector="tex-mi", skewchar=0x7F },
  { name="pxsy", vector="tex-sy", skewchar=0x30, parameters=true } ,
  { name="pxex", vector="tex-ex", extension=true } ,
  { name="pxsya", vector="tex-ma" },
  { name="pxsyb", vector="tex-mb" },
} )
```

As you can see, it is possible to add alphabets, given that there is a suitable vector that maps glyph indices onto Unicodes. It is good to know that this function only defines the way such a font is constructed. The actual construction is delayed till the font is needed.

Such a virtual font is used in typescripts (the building blocks of typeface definitions in ConTEXt) as follows:

```
\starttypescript [math] [palatino] [name]
  \definefontsynonym [MathRoman] [pxmath@px-math]
  \loadmapfile[original-youngryu-px.map]
\stoptypescript
```

If you are familiar with the way fonts are defined in ConTEXt, you will notice that we no longer need to define MathItalic, MathSymbol and additional symbol fonts. Of course users don't have to deal with these issues themselves. The `@` triggers the virtual font builder.

You can imagine that in MkII switching to another font style or size involves initializing (or at least checking) some 30 to 40 font definitions when it comes to math (the number of used families times 3, the number of math sizes.). And even if we take into account that fonts are loaded only once, this checking and enabling takes time. Keep in mind that in ConTEXt we can have several math font sets active in one document which comes at a price.

In MkIV we use one family (at three sizes). Of course we need to load the font (and more than one in the case of virtual variants) but when switching bodyfont sizes we only need to enable one (already defined) math font. And that really saves time. This is one of the areas where we gain back time that we loose elsewhere by extending core functionality using Lua (like OpenType support).

Dimensions

By setting font related dimensions you can control the way TEX positions math elements relative to each other. Math fonts have a few more dimensions than regular text fonts. But OpenType math fonts like Cambria have quite some more. There is a nice booklet published by Microsoft, 'Mathematical Typesetting', where dealing with math is discussed in the perspective of their word processor and TEX. In the booklet some of the parameters are discussed and since many of them are rather special it makes no sense (yet) to elaborate on them here. Figuring out their meaning was quite a challenge.

I am the first to admit that the current code in MkIV that deals with math parameters is somewhat messy. There are several reasons for this:

- We can pass parameters as a `MathConstants` table in the tfm table that we pass to the core engine.
- We can use some named parameters, like `x_height` and pass those in the `parameters` table.
- We can use the traditional font dimension numbers in the `parameters` table, but since they overlap for symbol and extensible fonts, that is asking for troubles.

Because in MkIV we create virtual fonts at run-time and use just one family, we fill the `MathConstants` table for traditional fonts as well. Future versions may use the upcoming mechanisms of font parameter sets at the macro level. These can be defined for each of the sizes (display, text, script and scriptscript, and the last three in cramped form as well) but since a font only carries one set, we currently use a compromise.

Tracing

One of the nice aspects of the opened up math machinery is that it permits us to get a more detailed look at what happens. It also fits nicely in the way we always want to visualize things in ConTEXt using color, although most users are probably unaware of many such features because they don't need them as I do.

```
\enabletrackers[math.analyzing]
\ruledhbox{$a = \sqrt{b^2 + \sin{c} - {1 \over \gamma}}$}
\disabletrackers[math.analyzing]
```

$$\boxed{a = \sqrt{b^2 + \sin c - \frac{1}{\gamma}}}$$

This tracker option colors characters depending on their nature and the fact that they are remapped. The tracker also was handy during development of LuaTEX especially for checking if attributes migrated right in constructed symbols.

For over a year I had been using a partial Unicode math implementation in some projects but for serious math the vectors needed to be completed. In order to help the ‘math department’ of the ConTEXt development team (Aditya Mahajan, Mojca Miklavec, Taco Hoekwater and myself) we have some extra tracing options, like

```
\showmathfontcharacters[][0x0007B]
```

U+0007B: { left curly bracket
  width: 253760, height: 463680, depth: 146560, italic: 0
  mathclass: open, mathname: lbrace
  next: U+F03B0 { => U+F04CE { => U+F03B1 { => U+F04D4 { => U+F03B2 { => U+F04DA { => U+F03B3 { => variants: U+023A9 ⎩ => U+023AA ⎪ => U+023A8 ⎨ => U+023AA ⎪ => U+023A7 ⎧

The simple variant with no arguments would have extended this document with many pages of such descriptions.

Another handy command (defined in module `fnt-25`) is the following:

```
\ShowCompleteFont{name:cambria}{9pt}{1}
\ShowCompleteFont{dummy@lmroman10-math}{10pt}{1}
```

For Cambria this will generate between 50 and 100 pages of character tables.

If you look at the following samples you can imagine how coloring the characters and replacements helped figuring out the alphabets. We use the following input (stored in a buffer):

```
$abc \bf abc \bi abc$
$\mathscript abcdefghijklmnopqrstuvwxyz $
$\mathscript 1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
$\mathfraktur abcdefghijklmnopqrstuvwxyz$
$\mathfraktur 1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
$\mathblackboard abcdefghijklmnopqrstuvwxyz $
$\mathblackboard 1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
```

```
$\mathscript abc IRZ \mathfraktur abc IRZ $
$\mathblackboard abc IRZ \ss abc IRZ 123$
```

For testing Cambria we say:

```
\usetypescript[cambria]
\switchtobodyfont[cambria,11pt]
\enabletrackers[math.analyzing]
\getbuffer[mathtest] % the input shown before
\disabletrackers[math.analyzing]
```

And we get:

𝑎𝑏𝑐𝐚𝐛𝐜𝒂𝒃𝒄
𝒶𝒷𝒸𝒹ℯ𝒻ℊ𝒽𝒾𝒿𝓀𝓁𝓂𝓃ℴ𝓅𝓆𝓇𝓈𝓉𝓊𝓋𝓌𝓍𝓎𝓏
1234567890𝒜ℬ𝒞𝒟ℰℱ𝒢ℋℐ𝒥𝒦ℒℳ𝒩𝒪𝒫𝒬ℛ𝒮𝒯𝒰𝒱𝒲𝒳𝒴𝒵
𝔞𝔟𝔠𝔡𝔢𝔣𝔤𝔥𝔦𝔧𝔨𝔩𝔪𝔫𝔬𝔭𝔮𝔯𝔰𝔱𝔲𝔳𝔴𝔵𝔶𝔷
1234567890𝔄𝔅ℭ𝔇𝔈𝔉𝔊ℌℑ𝔍𝔎𝔏𝔐𝔑𝔒𝔓𝔔ℜ𝔖𝔗𝔘𝔙𝔚𝔛𝔜ℨ
𝕒𝕓𝕔𝕕𝕖𝕗𝕘𝕙𝕚𝕛𝕜𝕝𝕞𝕟𝕠𝕡𝕢𝕣𝕤𝕥𝕦𝕧𝕨𝕩𝕪𝕫
𝟙𝟚𝟛𝟜𝟝𝟞𝟟𝟠𝟡𝟘𝔸𝔹ℂ𝔻𝔼𝔽𝔾ℍ𝕀𝕁𝕂𝕃𝕄ℕ𝕆ℙℚℝ𝕊𝕋𝕌𝕍𝕎𝕏𝕐ℤ
𝒶𝒷𝒸ℐℛ𝒵𝔞𝔟𝔠ℑℜℨ
𝕒𝕓𝕔𝕀ℝℤ𝖺𝖻𝖼𝖨𝖱𝖹123

For the virtualized Latin Modern we say:

```
\usetypescript[modern]
\switchtobodyfont[modern,11pt]
\enabletrackers[math.analyzing]
\getbuffer[mathtest] % the input shown before
\disabletrackers[math.analyzing]
```

This gives:

𝑎𝑏𝑐abc𝒂𝒃𝒄
abcdefghijklmnopqrstuvwxyz
1234567890𝒜ℬ𝒞𝒟ℰℱ𝒢ℋℐ𝒥𝒦ℒℳ𝒩𝒪𝒫𝒬ℛ𝒮𝒯𝒰𝒱𝒲𝒳𝒴𝒵
𝔞𝔟𝔠𝔡𝔢𝔣𝔤𝔥𝔦𝔧𝔨𝔩𝔪𝔫𝔬𝔭𝔮𝔯𝔰𝔱𝔲𝔳𝔴𝔵𝔶𝔷
1234567890𝔄𝔅ℭ𝔇𝔈𝔉𝔊ℌℑ𝔍𝔎𝔏𝔐𝔑𝔒𝔓𝔔ℜ𝔖𝔗𝔘𝔙𝔚𝔛𝔜ℨ
abcdefghij𝕜lmnopqrstuvwxyz
1234567890𝔸𝔹ℂ𝔻𝔼𝔽𝔾ℍ𝕀𝕁𝕂𝕃𝕄ℕ𝕆ℙℚℝ𝕊𝕋𝕌𝕍𝕎𝕏𝕐ℤ
abcℐℛ𝒵𝔞𝔟𝔠ℑℜℨ
abc𝕀ℝℤ𝖺𝖻𝖼𝖨𝖱𝖹123

These two samples demonstrate that Cambria has a rather complete repertoire of shapes which is no surprise because it is a recent font that also serves as a showcase for Unicode and OpenType driven math.

Commands like `\mathscript` set an attribute. When we post-process the noad list and encounter this attribute, we remap the characters to the desired variant. Of course this happens selectively. So, a capital A (`0x0041`) becomes a capital script A (`0x1D49C`). Of course this solution is rather ConTEXt-specific and there are other ways to achieve the same goal (like using more families and switching family.)

Special cases

Because we now are operating in the Unicode domain, we run into problems if we keep defining some of the math symbols in the traditional TEX way. Even with the ams fonts available, we still end up with some characters that are represented by combining others. Take for instance ≠ which is composed of two characters. Because in MkIV we want to have all characters in their pure form, we use a virtual replacement for them. In MkIV speak it looks like this:

```
local function negate(main,unicode,basecode)
    local characters = main.characters
    local basechar = characters[basecode]
    local ht, wd = basechar.height, basechar.width
    characters[unicode] = {
        width    = wd,
        height   = ht,
        depth    = basechar.depth,
        italic   = basechar.italic,
        kerns    = basechar.kerns,
        commands = {
            { "slot", 1, basecode },
            { "push" },
            { "down",    ht/5},
            { "right", - wd/2},
            { "slot", 1, 0x2215 },
            { "pop" },
        }
    }
end
```

In case you're curious, there are indeed kerns, in this case the kerns with the Greek Delta.

Another thing we need to handle is positioning of accents on top of slanted (italic) shapes. For this TEX uses a special character in its fonts (set with `\skewchar`). In its kerning table, any character can have a kern towards this special character. From this kern we can calculate the `top_accent` variable that we can pass for each character. This variable lives at the same level as `width`, `height`, `depth` and `italic` and is calculated as: $w/2 + k$, so it defines the horizontal anchor. A nice side effect is that (in the ConTEXt font management subsystem) this saves us passing information associated with specific fonts such as the skew character.

A couple of concepts are unique to TEX, like having `\hat` and `\widehat` where the wide one has sizes. In OpenType and Unicode we don't have this distinction so we need special trickery to simulate this. We do so by adding extra code points in a private Unicode space which in return results in them being defined automatically and the relevant first size variant being used for `\hat`. For some users this might still be too wide but at least it's better than a wrongly positioned ascii variant. In the future we might use this private space for similar cases.

Arrows, horizontal extenders and radicals also fall in the category ‘troublesome’ if only because they use special dimensions to get the desired effect. Fortunately OpenType math is modeled after TEX, so in LuaTEX we introduce a couple of new constructs to deal with this. One such simplification at the macro level is in the definition of `\root`. Here we use the new `\Uroot` primitive. The placement related parameters are those used by traditional TEX, but when they are available the OpenType variants are applied. The simplified plain definitions are now:

```
\def\rootradical{\Uroot 0 "221A }

\def\root#1\of{\rootradical{#1}}

\def\sqrt{\rootradical{}}
```

The successive sizes of the root will be taken from the font in the same way as traditional TEX does it. In that sense LuaTEX is not doing anything differently, it only has more parameters to control the process. The definition of `\sqrt` in ConTEXt permits an optional first argument that sets the degree.

U+0221A: √ square root
width: 430400, height: 603520, depth: 27200, italic: 0
mathclass: radical, mathname: surd

next: U+F03F8 √ => U+F03F9 √ => U+F03FA √ => U+F03FB √

=> U+F03FC √ => variants: U+023B7 √ => U+020D3 □ => U+F04A1 □

Note that we have collected all characters in family 0 (simply because that is what TEX defaults characters to) and that we use the formal Unicode slots. When we use the Latin Modern fonts we just remap traditional slots to the right ones.

Another neat trick is used when users choose among the bigger variants of some characters. The traditional approach is to create a box of a certain size and create a fake delimited variant which is then used.

```
\definemathcommand [big]  {\choosemathbig\plusone  }
\definemathcommand [Big]  {\choosemathbig\plustwo  }
\definemathcommand [bigg] {\choosemathbig\plusthree}
\definemathcommand [Bigg] {\choosemathbig\plusfour }
```

Of course this can become a primitive operation and we might decide to add such a primitive later on so we won't bother you with more details.

Attributes are also used to make live easier for authors who have to enter lots of pairs. Compare:

```
\setupmathematics[autopunctuation=no]

$ (a,b) = (1.20,3.40) $
```

$(a, b) = (1.20, 3.40)$

with:

```
\setupmathematics[autopunctuation=yes]

$ (a,b) = (1.20,3.40) $
```

$(a{,}b) = (1.20{,}3.40)$

So we don't need to use this any more:

```
$ (a{,}b) = (1{.}20{,}3{.}40) $
```

Features like this are implemented on top of an experimental math manipulation framework that is part of MkIV. When the math font system is stable we will rework the rest of math support and implement additional manipulating frameworks.

Control

As with all other character related issues, in MkIV everything is driven by a character table (consider it a database). Quite some effort went into getting that one right and although by now math is represented well, more data will be added in due time.

In MkIV we no longer have huge lists of TEX definitions for math related symbols. Everything is initialized using the mentioned table: normal symbols, delimiters, radicals, with or without name. Take for instance the square root:

U+0221A: √ square root
width: 430400, height: 603520, depth: 27200, italic: 0
mathclass: radical, mathname: surd

next: U+F03F8 √ => U+F03F9 √ => U+F03FA √ => U+F03FB √ => U+F03FC √ => variants: U+023B7 √ => U+020D3 □ => U+F04A1 □

Its entry is:

```
[0x221A] = {
    adobename = "radical",
    category = "sm",
    cjkwd = "a",
    description = "SQUARE ROOT",
    direction = "on",
    linebreak = "ai",
    mathclass = "radical",
    mathname = "surd",
    unicodeslot = 0x221A,
}
```

The fraction symbol also comes in sizes (this symbol is not to be confused with the negation symbol `0x2215` – which is known as `\not` in TEX terminology):

U+02044: ⁄ fraction slash
width: 362880, height: 457920, depth: 137600, italic: 0
mathclass: binary, mathname: slash
mathclass: close, mathname: solidus

next: U+F03AC ⁄ => U+F03AD ⁄ => U+F03AE ⁄ => U+F03AF ⁄

```
[0x2044] = {
    adobename = "fraction",
    category = "sm",
    contextname = "textfraction",
    description = "FRACTION SLASH",
    direction = "cs",
    linebreak = "is",
    mathspec = {
        { class = "binary", name = "slash" },
        { class = "close", name = "solidus" },
    },
    unicodeslot = 0x2044,
}
```

However, since most users don't have this symbol visualized in their word processor, they expect the same behavior from the regular slash. This is why we find a reference to the real symbol in its definition.

U+0002F: / solidus
width: 321280, height: 457920, depth: 137600, italic: 0
mathsymbol: U+02044 ⁄

The definition is:

```
[0x002F] = {
    adobename = "slash",
    category = "po",
    cjkwd = "na",
    contextname = "textslash",
    description = "SOLIDUS",
    direction = "cs",
    linebreak = "sy",
    mathsymbol = 0x2044,
    unicodeslot = 0x002F,
}
```

One problem left is that currently we have only one class per character (apart from the delimiter and radical usage which have their own definitions). Future releases of ConTEXt will provide support for math dictionaries (as in OpenMath and MathML 3). At that point we will also have a `mathdict` entry.

There is another issue with character mappings, one that will seldom reveal itself to the user, but might confuse macro writers when they see an error message.

In traditional TEX, and therefore also in the Latin Modern fonts, a chain from small to large character goes in two steps: the normal size is taken from one family and the larger variants from another. The larger variant then has a pointer to an even larger one and so on, until there is no larger variant or an extensible recipe is found. The default family is number 0. It is for this reason that some of the definition primitives expect a small and large family part.

However, in order to support OpenType in LuaTEX, the alternative method no longer assumes this split. After all, we no longer have a situation where the 256 limit forces us to take the smaller variant from one font and the larger sequence from another (so we need two family–slot pairs where each family eventually resolves to a font).

It is for that reason that the new `\U...` primitives expect only one family specification: the small symbol, which then has a pointer to a larger variant when applicable. However deep down in the engine, there is still support for the multiple family solution (after all, we don't want to drop compatibility). As a result, in error messages you can still find references (defaulting to 0) to large specifications, even if you don't use them. In that case you can simply ignore the large symbol (0,0), since it is not used when the small symbol provides a link.

Extensibles

In TEX fences can be told to become larger automatically. In traditional TEX a character can have a linked list of next larger shapes ending in a description of how to compose even larger variants.

A parenthesis in Cambria has the following list:

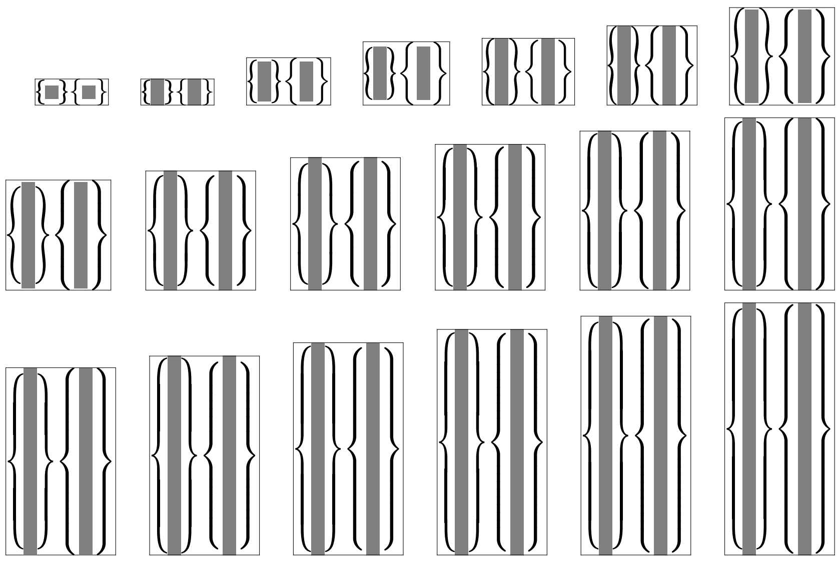
U+00028: (left parenthesis
width: 272000, height: 462400, depth: 144640, italic: 0
mathclass: open, mathname: lparent

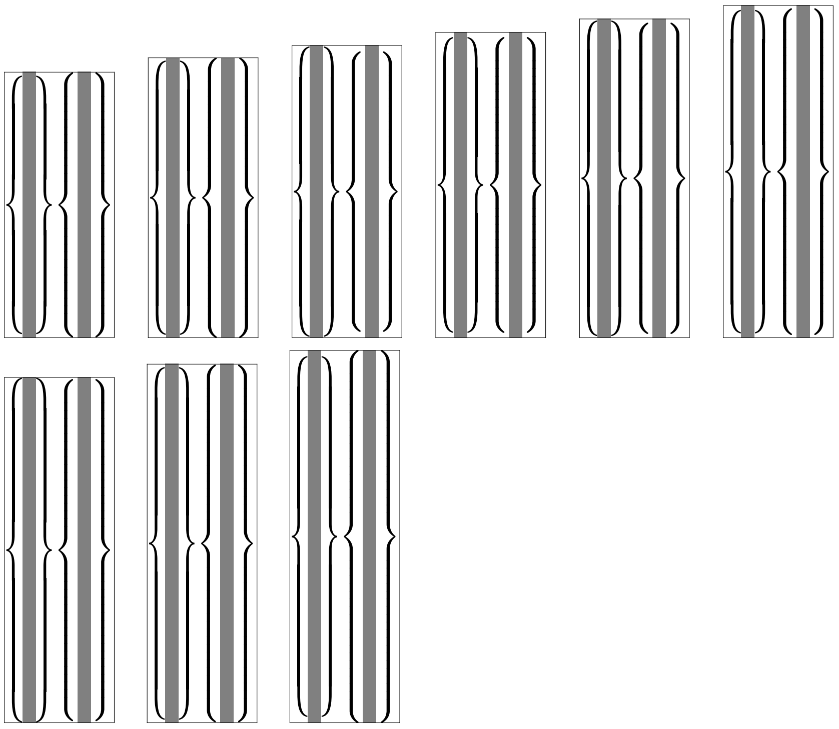
next: U+F03C0 (=> U+F04CA (=> U+F03C1 (=> U+F04D0 (=> U+F03C2 (=> U+F04D6 (=> U+F03C3 (=> variants: U+0239D ⎝ => U+0239C ⎜ => U+0239B ⎛

In Latin Modern we have:

U+00028: (left parenthesis
width: 254935.04, height: 491520, depth: 163840, italic: 0
mathclass: open, mathname: lparent
next: U+FF000 (=> U+FF010 (=> U+FF012 (=> U+FF020 (=> U+FF030 ⎛ => variants: U+FF040 ⎝ => U+FF042 ⎜ => U+FF030 ⎛

Of course, LuaTEX is downward compatible with respect to this feature, but the internal representation is now closer to what OpenType math provides (which is not that far from how TEX works, simply because it is inspired by TEX). Because Cambria has different parameters we get slightly different results. In the following list of pairs, you see Cambria on the left, Latin Modern on the right. Both start with stepwise larger shapes, followed by a more gradual growth. The thresholds for a next step are driven by parameters set in the OpenType font or by TEX's default.





In traditional TEX horizontal extensibles are not really present. Accents are chosen from a linked list of variants and don't have an extensible specification. This is because most such accents grow in two dimensions and the only extensible accents are rules and braces. However, in Unicode we have a few more and also because of symmetry we decided to add horizontal extensibles too. Take:

```
$ \overbrace {a+1} \underbrace {b+2} \doublebrace {c+3} $ \par
$ \overparent{a+1} \underparent{b+2} \doubleparent{c+3} $ \par
```

This gives:

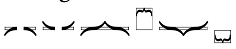
$\overbrace{a+1}\,\underbrace{b+2}\,\overbrace{\underbrace{c+3}}$

$\overparen{a+1}\,\underparen{b+2}\,\overparen{\underparen{c+3}}$

Contrary to Cambria, Latin Modern Math, which is just like Computer Modern Math, has no ready overbrace glyphs. Keep in mind that in the latter we are dealing with fonts that have only 256 slots and that the traditional font mechanism has the same limitation. For this reason, the (extensible) braces are traditionally made from snippets as is demonstrated below.

```
\hbox\bgroup
  \ruledhbox{\getglyph{lmex10}{\char"7A}}
  \ruledhbox{\getglyph{lmex10}{\char"7B}}
  \ruledhbox{\getglyph{lmex10}{\char"7C}}
  \ruledhbox{\getglyph{lmex10}{\char"7D}}
  \ruledhbox{\getglyph{lmex10}{\char"7A\char"7D\char"7C\char"7B}}
  \ruledhbox{\getglyph{name:cambriamath}{\char"23DE}}
  \ruledhbox{\getglyph{lmex10}{\char"7C\char"7B\char"7A\char"7D}}
  \ruledhbox{\getglyph{name:cambriamath}{\char"23DF}}
\egroup
```

This gives:



The four snippets have the height and depth of the rule that will connect them. Since we want a single interface for all fonts we no longer will use macro based solutions. First of all fonts like Cambria don't have the snippets, and using active character trickery (so that we can adapt the meaning to the font) has no preference either. This confines us to virtual glyphs.

It took us a bit of experimenting to get the right virtual definition because it is a multi–step process:

- The right Unicode character (0x23DE) points to a character that has no glyph itself but only horizontal extensibles.
- The snippets that make up the extensible don't have the right dimensions (as they define the size of the connecting rule), so we need to make them virtual themselves and give them a size that matches LuaTEX's expectations.
- Each virtual snippet contains a reference to the physical snippet and moves it up or down as well as fixes its size.
- The second and fifth snippet are actually not real glyphs but rules. The dimensions are derived from the snippets and it is shifted up or down too.

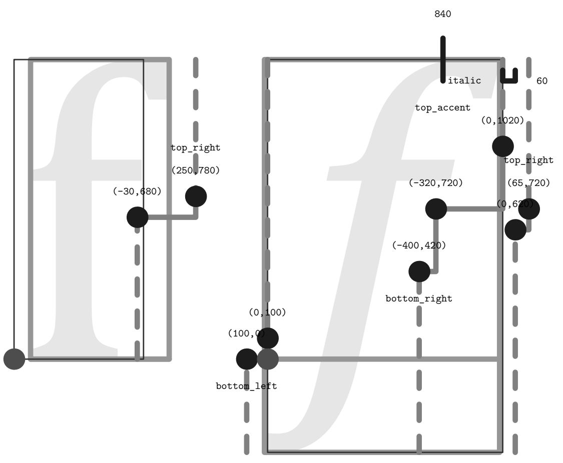
You might wonder if this is worth the trouble. Well, it is if you take into account that all upcoming math fonts will be organized like Cambria.

Math kerning

While reading Microsofts orange booklet, it became clear that OpenType provides advanced kerning possibilities and we decided to put it on the agenda for LuaTEX.

It is possible to define a ladder–like boundary for each corner of a character where the ladder more or less follows the shape of a character. In theory this means that when we attach a superscript to a base character we can use two such ladders to determine the optimal spacing between them.

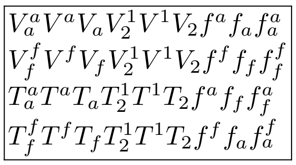
Let's have a look at a few characters, the upright f and its italic cousin.

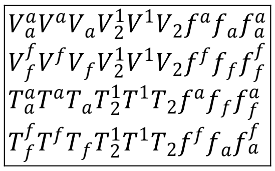


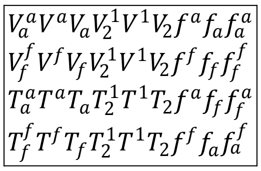
U+00066 0x1D453

The ladders on the right can be used to position a super- or subscript, that is, they are positioned in the normal way but the ladder, as well as the boundingbox and/or left ladders of the scripts, can be used to fine tune the positioning.

Should we use this information? I made this visualizer for checking some Arabic fonts anchoring and cursive features and then it made sense to add some of the information related to math as well. (Taco extended the visualizer for his presentation at Bachotek 2009 so you might run into variants.) The orange booklet shows quite advanced ladders, and when looking at the 3500 shapes in Cambria, it quickly becomes clear that in practice there is not that much detail in the specification. Nevertheless, because without this feature the result is not acceptable, LuaTEX gracefully supports it.


latin modern


cambria
without kerning


cambria with kerning

Faking glyphs

A previous section already discussed virtual shapes. In the process of replacing all shapes that lack in Latin Modern and are composed of snippets instead, we ran into the dots. As they are a nice demonstration of something that, although somewhat of a hack, survived 30 years without problems we show the definition used in ConTEXt MkII:

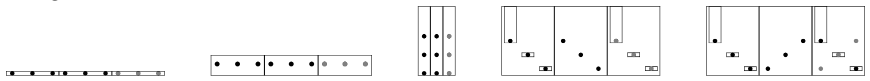
```
\def\PLAINldots{\ldotp\ldotp\ldotp}
\def\PLAINcdots{\cdotp\cdotp\cdotp}

\def\PLAINvdots
  {\vbox{\forgetall\baselineskip.4\bodyfontsize\lineskiplimit\zeropoint
         \kern.6\bodyfontsize\hbox{.}\hbox{.}\hbox{.}}}

\def\PLAINddots
  {\mkern1mu%
   \raise.7\bodyfontsize\ruledvbox{\kern.7\bodyfontsize\hbox{.}}%
   \mkern2mu%
   \raise.4\bodyfontsize\relax\ruledhbox{.}%
   \mkern2mu%
   \raise.1\bodyfontsize\ruledhbox{.}%
   \mkern1mu}
```

This permitted us to say:

```
\definemathcommand [ldots] [inner]   {\PLAINldots}
\definemathcommand [cdots] [inner]   {\PLAINcdots}
\definemathcommand [vdots] [nothing] {\PLAINvdots}
\definemathcommand [ddots] [inner]   {\PLAINddots}
```

However, in MkIV we use virtual shapes instead.

The following lines show the virtual shapes in greyscale. In each triplet we see the original, the virtual and the overlaid character.



As you can see here, the virtual variants are rather close to the originals. At 12pt there are no real differences but (somehow) at other sizes we get slightly different results but it is hardly visible. Watch the special spacing above the shapes. It is probably needed for getting the spacing right in matrices (where they are used).

Hans Hagen