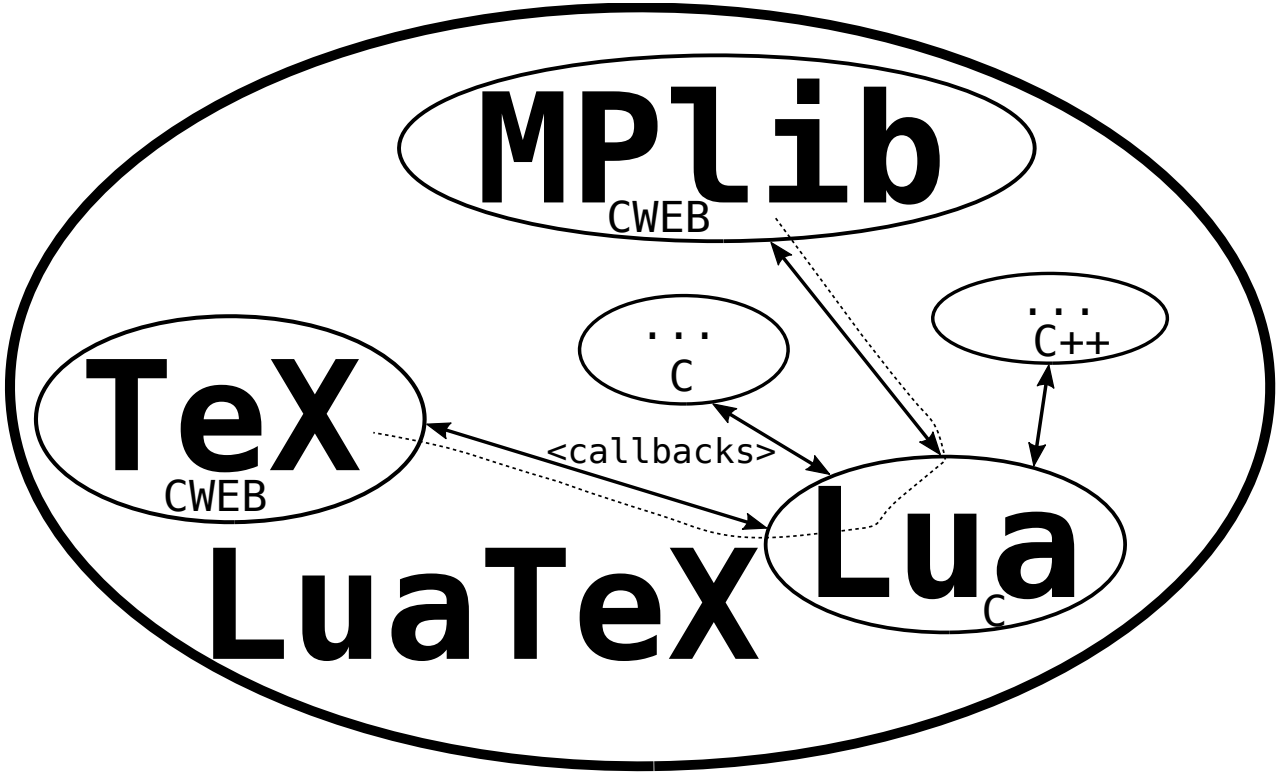
MFLua 0.8

Variable fonts

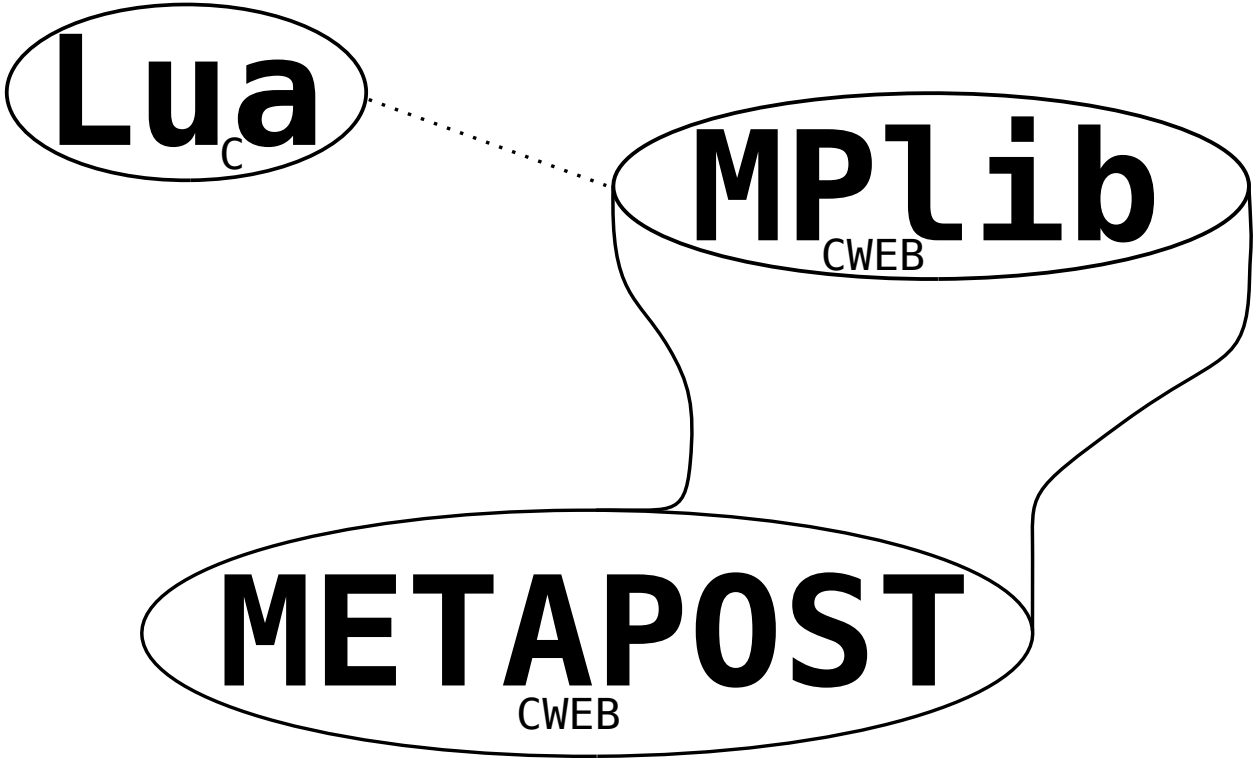
MFLUA

- A ‘small’ superset of METAFONT;
- A METAFONT program can be executed unmodified by MFLUA, giving the same result if MFLUA doesn’t use Lua scripts.
- A MFLUA program can be executed by METAFONT almost without modification, giving the same result if MFLUA doesn’t use Lua scripts.

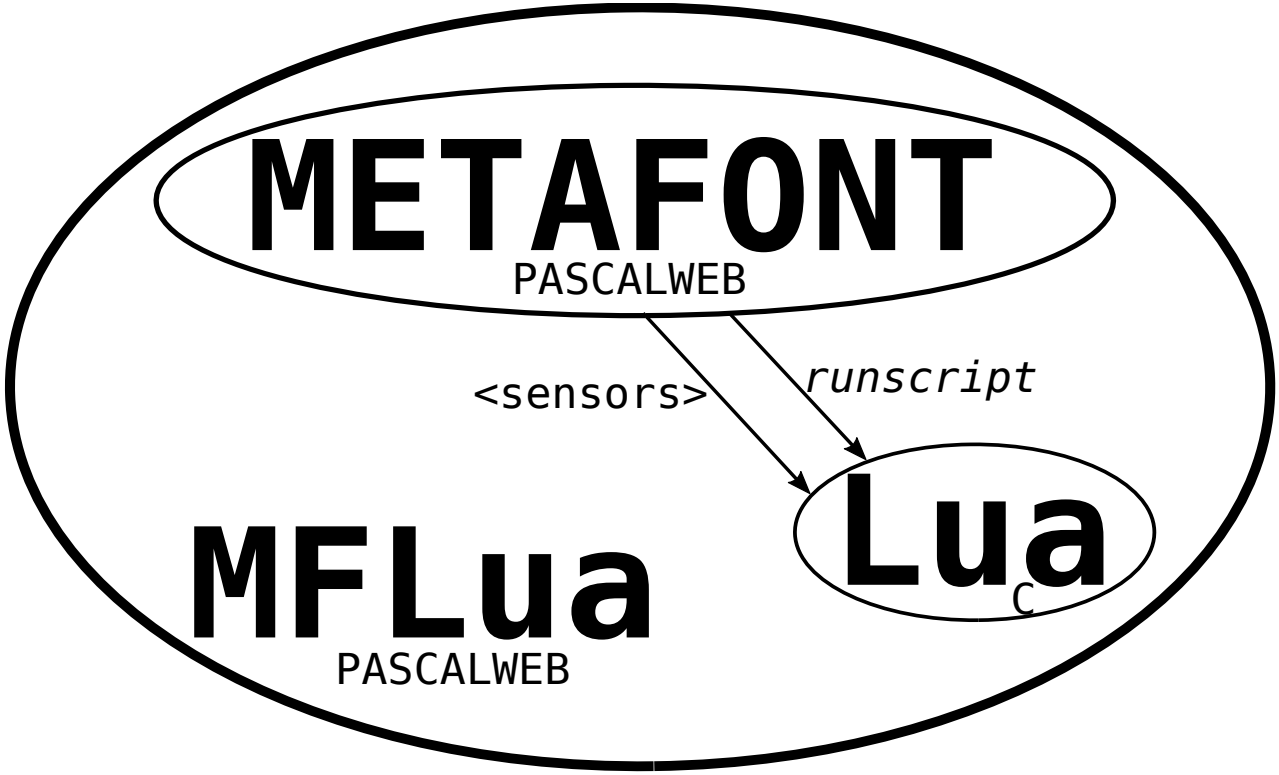
LuaTEX, MetaPost and MFLUA



LuaTEX, MetaPost and MFLUA



LuaTEX, MetaPost and MFLUA



MFLUA

Why another implementation of METAFONT ?

- The main output of METAFONT is a bitmap version of the draw described by mathematical and vectorial instructions;
- but ``METAFONT works internally with outlines'' (cubic Bézier curve);
- these outlines can be recorded in the log file, but their post-processing is difficult. MFLUA can save bitmap and curves into Lua tables — and these are easier to process.

MFLUA

MFLUA can:

- work with bitmaps and curves;
- use Lua to connect METAFONT to a (C) external library (i.e. GUI library);
- use Lua to export the curves into a font format;
- use Lua to enhance the math of METAFONT (by default the math of libc).

MFLUA

Callback vs. sensor:

- LuaTEX uses callbacks, i.e. functions that can read and modify the state of TEX; it's written in CWEB.

- MFLUA uses readonly callbacks (a.k.a. 'sensors'): a Lua script can only read a value from METAFONT but cannot modify it. The sensors are inserted modifying the original PASCAL source by mean of a change file.

METAFONT and MFLUA are essentially the same program: MFLUA adds only one new primitive runscript to talk to the Lua interpreter .

MFLUA

Example: putting a sensor around make_ellipse. In PASCAL the mflua_ sensor looks like a PASCAL procedure:

```
@x [41.866] MFLua
q:=make_ellipse(major_axis,minor_axis,theta);
if (tx<>0)or(ty<>0) then @<Shift the coordinates of path
|q|@>;
@y
mfluaPRE_make_ellipse(major_axis,minor_axis,theta,tx,ty,0);
q:=make_ellipse(major_axis,minor_axis,theta);
if (tx<>0)or(ty<>0) then @<Shift the coordinates of path
|q|@>;
mfluaPOST_make_ellipse(major_axis,minor_axis,theta,tx,ty,q);
@z
```

MFLUA

but it calls a C function, which pass the values to the Lua interpreter:

```
int mfluaPREmakeellipse(integer major_axis, integer minor_axis, integer theta,
                                                integer tx, integer ty,integer q)
 {
   lua_State *L;
   int res;

     L = Luas[0];
     GETGLOBALTABLEMFLUA(mfluaPREmakeellipse);
     if (lua_istable(L, -1)) {
       lua_getfield(L,-1,"PRE_make_ellipse");
       lua_pushnumber(L, major_axis);   /* push 1st argument */
       lua_pushnumber(L, minor_axis);   /* push 2nd argument */
       lua_pushnumber(L, theta);   /* push 3th argument */
       lua_pushnumber(L, tx);   /* push 4th argument */
       lua_pushnumber(L, ty);   /* push 5th argument */
       lua_pushnumber(L, q);   /* push 6th argument */
       /* do the call (6 arguments, 0 result) */
       if(res = lua_pcall(L, 6, 1, 0)){
         lua_pushstring(L,"error in PRE_make_ellipse:");
         lua_swap(L);lua_concat (L, 2);
         priv_lua_reporterrors(L, res);
       }
     }
     lua_settop(L,0);
     return 0;
 }
```

MFLUA

which in turn calls a function in the `mflua.lua` file:

```
local function PRE_make_ellipse(major_axis,minor_axis,theta,
                                                    tx,ty,q)
    PRINTDBG("PRE_make_ellipse")
end

local function POST_make_ellipse(major_axis,minor_axis,theta,
                                                     tx,ty,q)
   PRINTDBG("POST_make_ellipse")
    :
   res = res ..print_two(x_coord(p),y_coord(p))
   p=link(p)
   if p==q then flag=false end
    end
    mflua.pen[res] = {print_two(major_axis,minor_axis),
                      theta*(2^-20),print_two(tx,ty)}
 end
```

MFLUA

The more important sensor is end_program, because it's positioned just before the natural exit point of METAFONT. At this point all the files are written, so the sensor can call a user function to process the data collected so far and can also read the gf and tfm file.

MFLUA 0.8

Example of runscript:

```
numeric r;
 numeric t[];
 r:=0;
 r:=runscript(
  "return (math.sqrt(5)*math.sqrt(3))"
 );
 message "DEBUG r=" & decimal r; message "";
 runscript(
  "local t={2.2,3.3,1.1};                  "&
  "table.sort(t);                          "&
  "local s = 't[1]:=%f;t[2]:=%f;t[3]:=%f;' "&
  "return string.format(s,t[1],t[2],t[3]) "
  );
 message "DEBUG t[]=" & decimal t[1] & "," &
         decimal t[2] & "," & decimal t[3];
 message "";
 end.
 DEBUG r=3.87299
 DEBUG t[]=1.1,2.2,3.3
```

MFLUA 0.8

Example of runscript and MFLUAJIT:

```
numeric r;
 r = runscript(
 "local ffi = require('ffi')    " & char(10)   &
 "ffi.cdef[[                    " & char(10)   &
 "double      erf( double arg );" & char(10)   &
 "]]                            " & char(10)   &
 "return ffi.C.erf(-3)          "
 );
 message "DEBUG erf(-3)=" & decimal(r);
 end

 DEBUG erf(-3)=-0.99998
```

MFLUA 0.8

MFLUA was already used to produce an OpenType font, starting from the METAFONT sources of the concrete font. It was an experiment on using only Lua without touching the original sources, and preserving the original curves (i.e. no contours from the bitmap ala Potrace). The conclusion was that it takes too much to try to solve all the intersections, and the final result ended using ”per glyph” ugly tricks.
FontForge was used to convert from SVG to OpenType.

(the text is typesets with the Concrete OpenType made with MFLUA)

MFLUA 0.8

New in 0.8:

- new sensors around make_spec: it’s now possible to store the curves before the subdivision in quadrants;
- cleanup of the code: all the sensors are now in mflua.lua;
- a new (experimental) backend, ttx.

MFLUA 0.8

A different approach:

- use METAFONT to produce clean outlines (no pens!);
- use Lua to build backends, no modification of the outlines;
- use an external program to translate the backend to OTF.

Currently there are two backends: svg and ttx.

MFLUA 0.8

The sourcecode-regular font by C. Vincoletto is a METAFONT font built in two months to test the MFLUA new backends. The end_program sensor translate the data to svg and a FontForge script (almost equal to that one used by mf2pt1) translates the svg font into an OTF. The font was also used to test the new ttx backend: the ttx output can be converted to an OTF font, and this can be converted again into a ttx type without errors.

(the text is typesets with the sourcecode-regular made with MFLUA and svg backend)

MFLUA 0.8

For the BachoTEX meeting I have started to play with variable fonts. The ttx program can disassemble/assemble an OTF variable font, so I modified the ttx backend to make a CFF2 font. From the same METAFONT source of `sourcecode-regular`, redefining `fill` and `unfill` and applying a scaling transformation to each curve the backend setup a single axes (wdth) to reduce a glyph (a condensed font). The font passes the ttx checker, and of course it was of fundamental importance that ConTEXt can load variable fonts.

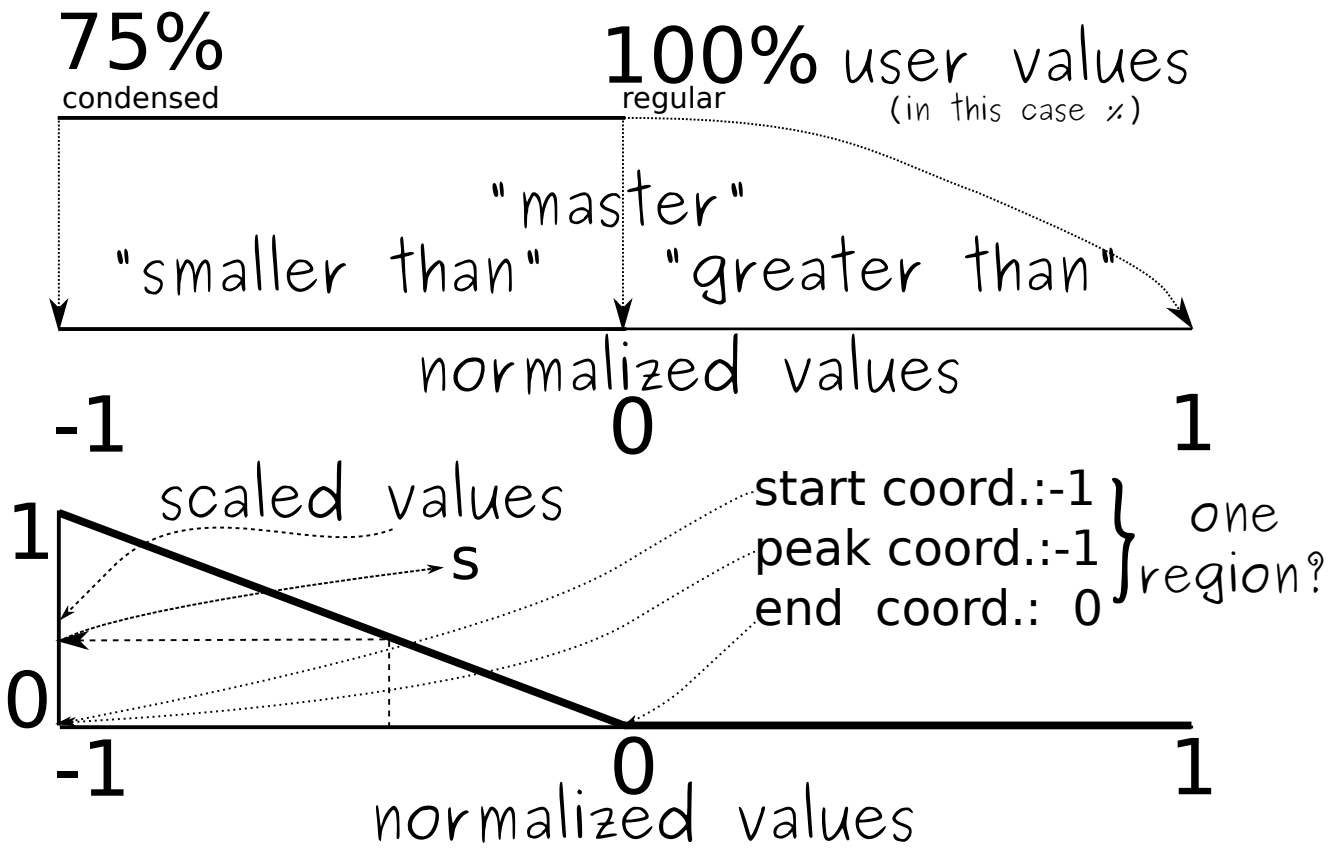
MFLUA 0.8

abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz

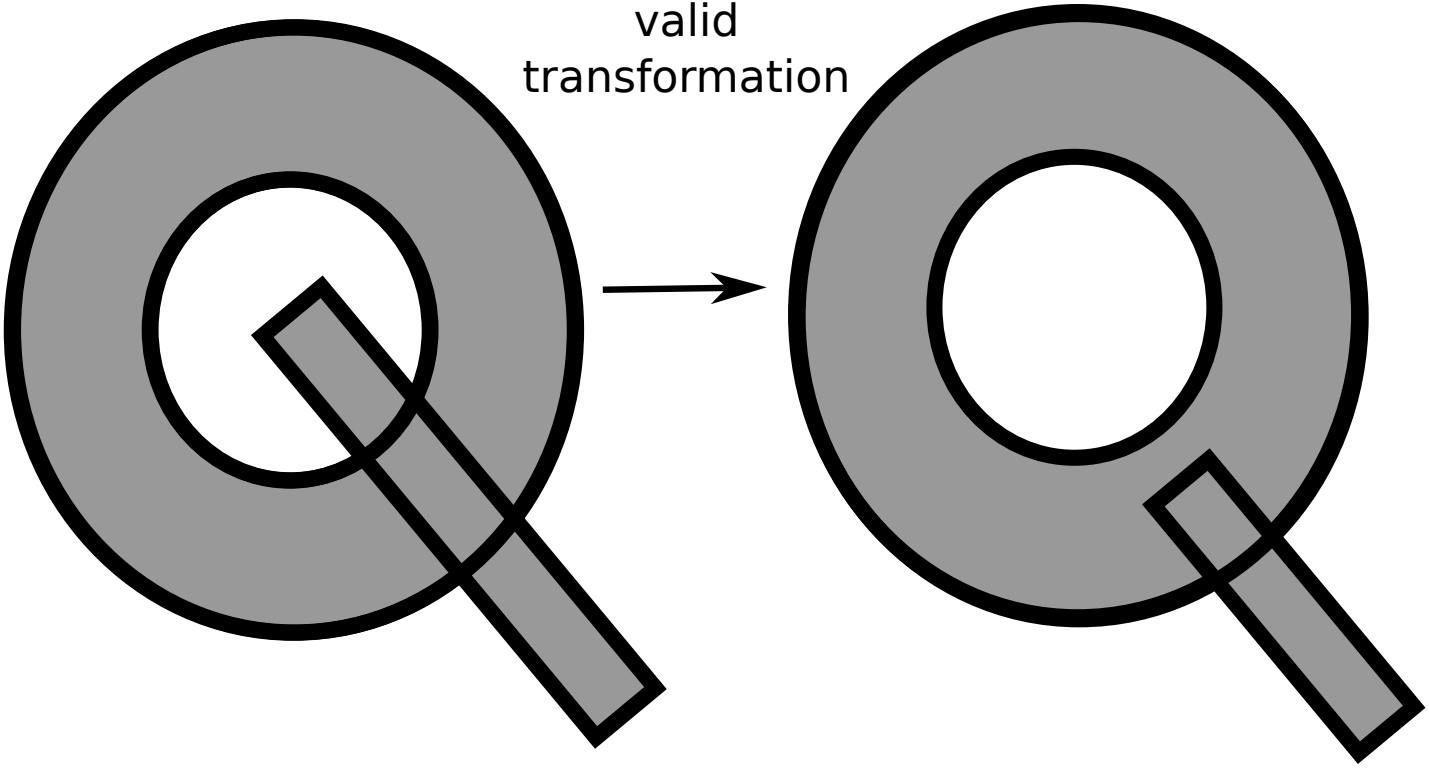
black:condensed:75
black:condensed:80
black:condensed:90
red:condensed:100 (default)
(yes, the ``i'' and ``j'' are wrong...)

MFLUA 0.8

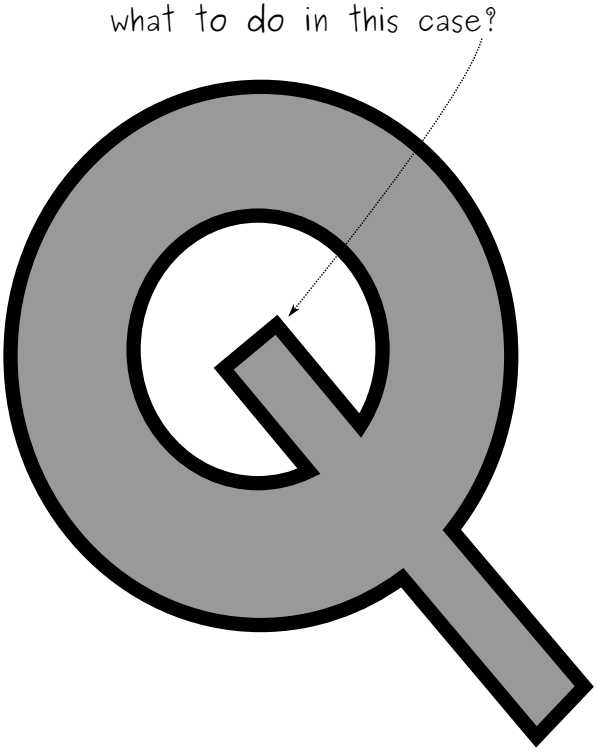
wdth axes



MFLUA 0.8

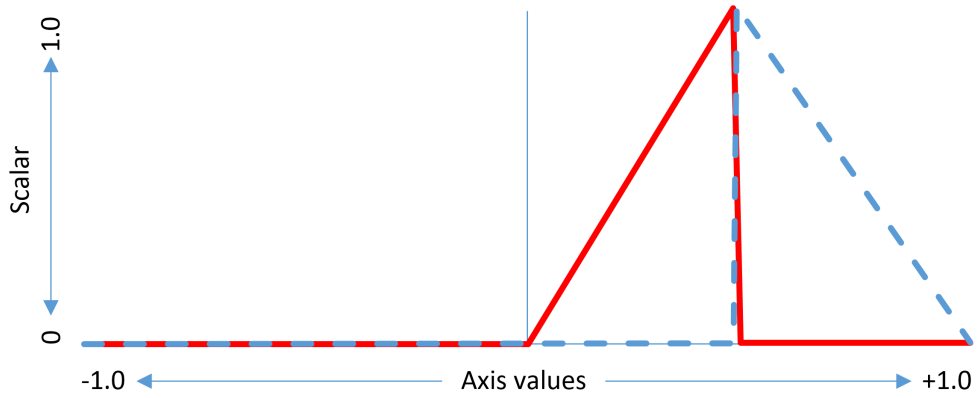


MFLUA 0.8



MFLUA 0.8

Piecewise transformations:



Alternative solution: perform glyph substitutions when a variation instance is selected in some range along one or more axes.

(source: `https://www.microsoft.com/typography/otspec/otvaroverview.htm`)

That’s all !

Thank you Folks !