How to Draw (String Diagrams)

Draft Tutorial v0.3

Ralf Hinze and Dan Marsden

October 9, 2022

1 Introduction

This brief note describes how to draw a particular style of string diagrams, appearing prolifically in [HM21, HM22], using the TikZ LaTeX package [TM]. Hopefully this tutorial will make it easier for others to get started with presenting string diagrams within their own work. The current version is certainly not particularly polished, and is intended as a first draft to solicit feedback.

Our approach will be to use example code, and some discussion of the salient details. We assume the reader will consult the excellent TikZ manual for further information about the library features being used, rather than duplicating that information here.

1.1 Perspective

Generating string diagrams with TikZ has at least three aspects.

1. It is an artistic endeavour - In our examples we shall aim for visually pleasing diagrams suitable for papers and other publications.
2. Communication of mathematical ideas - The examples will focus on layouts that occur in practice when drawing string diagrams that are expected to be easy to read by humans. As such we shall avoid discussion of alternative strategies such as automatic layout tools.
3. Software engineering - Each TikZ diagram is a small program, and should be treated as such. Good software engineering practices should be adopted, when constructing string diagrams, as the code is often modified or reused by others. Our sample code will aim for good quality TikZ code, within the limitations of our current understanding of the library.

Within this note, we shall restrict our techniques to what is achievable within the TikZ library. More code reuse and elimination of duplication could be achieved, for example by extensive use of LaTeX macros. We avoid this as it would obscure the ideas used in the code. Hopefully this will lay a solid foundation on which users can build more advanced capabilities if they so desire.

2 A Simple Two-Cell

We begin by drawing the following two-cell with a single input and output. Although simple, this diagram will already involve a lot of the main ideas needed to efficiently construct diagrams.

This diagram was drawn with the following code:

```
\begin{tikzpicture}[scale=0.5]
    % Coordinate layout and vertex positioning
    \path coordinate[
     circle, draw=black, fill=black, minimum size=1mm, inner sep=0mm, label=right:$\alpha$] (
          alpha)
    +(0,1) coordinate[label=above:$F$] (top)
    +(0,-1) coordinate[label=below:$G$] (bot);

    % Line drawing
    \draw (bot) -- (top);

    % Region filling
    \begin{scope}[on background layer]
        \fill[yellow!50] (bot) rectangle (top -| 1,0);
        \fill[orange!50] (bot) rectangle (top -| -1,0);
    \end{scope}
\end{tikzpicture}
```

The code is slightly verbose to make it self-contained. This will be tidied up later with some code reuse features.

Conceptually, the code has been structured into three stages:

1. Firstly, using the path command, we place the coordinates that we shall need to draw our diagram. It is also convenient to draw dots to mark two-cells in our diagrams during this stage, and to add various labels associated with positions in the diagram. For example, the code

   ```
   \path coordinate[circle, draw=black, fill=black, minimum size=1mm, inner sep=0mm,
        label=right:$\alpha$] (alpha)
   ```

   creates a coordinate `alpha` at the start position of a path. This is where the α two-cell will be positioned, so we add code to draw a black dot at this position, and place a label naming the vertex to its right. We then place a `top` and `bot` coordinate marking positions at the top and bottom of the diagram where lines will be connected. Their positions calculated relative to the coordinate `alpha`. We strongly recommended calculating positions in this relative manner. Absolute coordinates may seem slightly easier to work out, but they are fragile to subsequent adjustments.

2. Secondly, we draw the lines corresponding to one-cells in our diagrams. In this case, as the geometry is so simple, we only need to draw a line from top to bottom with the code:

```
\draw (bot) -- (top);
```

3. Finally, we fill in the coloured regions corresponding to zero-cells in our diagram. The code

```
\begin{scope}[on background layer]
    ...
\end{scope}
```

introduces a scope within which we work on a background layer. For this to work, we must include the backgrounds library:

```
\usetikzlibrary{backgrounds}
```

This use of layers greatly simplifies our code, as otherwise we need to be very careful about the order in which we draw and fill parts of our diagram to avoid overwriting our work. We fill the rectangular region on the right of the diagram yellow with

```
\fill[yellow!50] (bot) rectangle (top -| 1,0)
```

The parameter `yellow!50` sets a lightened yellow colour. The rectangle has bottom left corner `bot`, and top right hand corner calculated with

```
(top -| 1,0)
```

This second coordinate is at the intersection of a horizontal line through `top` and a vertical line through `(1,0)`. This sort of intersection calculation is very useful for calculating additional positions in a diagram. Note also that we have been lazy and used the absolute position `(1,0)` for convenience [1].

2.1 Getting Stylish

To avoid the duplicating the instructions to draw a black dot for a two-cell, we define a TikZ style:

```
\tikzstyle{dot}=[
 circle, draw=black, fill=black, minimum size=1mm, inner sep=0mm]
```

We can also use styles to fix some zero-cell colours:

```
\tikzstyle{catc}=[yellow!50]
\tikzstyle{catd}=[orange!50]
\tikzstyle{cate}=[red!50]
\tikzstyle{catf}=[color=blue!25]
```

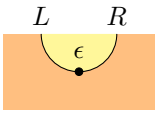
With our more concise notation, the code becomes:

```
\begin{tikzpicture}[scale=0.5]
   \path coordinate[dot, label=right:$\alpha$] (alpha)
   +(0,1) coordinate[label=above:$F$] (top)
   +(0,-1) coordinate[label=below:$G$] (bot);
   \draw (bot) -- (top);
   \begin{scope}[on background layer]
      \fill[catc] (bot) rectangle (top -| 1,0);
      \fill[catd] (bot) rectangle (top -| -1,0);
   \end{scope}
\end{tikzpicture}
```

[1] Do as we say, not as we do!

3 A Cup

As a second example, we draw a cup, as we might find as the counit of an adjunction:



This is done with the following code:

```
\begin{tikzpicture}[scale=0.5]
    \path coordinate[dot, label=above:$\epsilon$] (epsilon)
    +(-1,1) coordinate[label=above:$L$] (tl)
    +(1,1) coordinate[label=above:$R$] (tr);
    \draw (tl) to[out=-90, in=180] (epsilon.center) to[out=0, in=-90] (tr);
    \begin{scope}[on background layer]
        \fill[catd] (tl -| -2,0) rectangle +(4,-2);
        \fill[catc] (tl) to[out=-90, in=180] (epsilon.center) to[out=0, in=-90] (tr) --
              cycle;
    \end{scope}
\end{tikzpicture}
```

We have structured our code as in section 2, exploiting the styles introduced in section 2.1. The drawing code has now become more interesting, as we form a curve. The step `(tl)to[out=-90, in=180] (epsilon.center)` draws a curved line, leaving at -90 degrees from the coordinate `tl`, targeting specifically the centre of `epsilon`, at an angle of 180 degrees. The explicit targeting of the centre when drawing curves like this ensures that there are no gaps in our curves. Without adding the explicit `.center` strange behaviour may occur when filling regions bounded by our curved edge. Experience has shown that building up lines piece-wise in this way scales well to larger diagrams.

Notice that we draw our curved line with

```
\draw (tl) to[out=-90, in=180] (epsilon.center) to[out=0, in=-90] (tr);
```

and then describe the fill of the interior of the cup with:

```
\fill[catc] (tl) to[out=-90, in=180] (epsilon.center) to[out=0, in=-90] (tr) -- cycle;
```

A few things are worth pointing out here:

1. When describing the fill region, we traverse the curved edge in the same direction as when we drew it the first time. This is important, if you are inconsistent about this unpleasant gaps will appear around your fill regions, as TikZ calculates slightly different curves depending on the direction of travel.

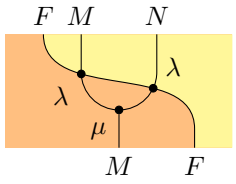
2. We use `cycle` to return to the start of a complicated curve and close off a region. This is recommended TikZ practice, although occasionally this feature can produce odd results.

3. Notice there is a large amount of code duplication here, as we have to spell out the details of the same path twice. We are not aware of a feature

in the TikZ library that avoids this issue. To avoid this issue it may be necessary to use techniques beyond the scope of the library, such as introducing macros.

4 Getting More Complicated

A slightly more complicated diagram, as might appear as part of a distributive law axiom:



This diagram was produced by the code:

```
\begin{tikzpicture}[scale=0.5]
       \path coordinate (tlll)
       ++(1,0) coordinate[label=above:$F$] (tll)
       ++(1,0) coordinate[label=above:$M$] (tl)
       ++(0,-1) coordinate (a)
       ++(1,-1) coordinate[dot, label=-135:$\mu$] (mu) +(0,-1) coordinate[label=below:$M$] (
            bl)
       ++(1,1) coordinate (b)
       ++(0,1) coordinate[label=above:$N$] (tr)
       (bl) ++(2,0) coordinate[label=below:$F$] (br) +(0,0.5) coordinate (c) +(1,0)
            coordinate (brr);
       \draw[name path=cup] (tl) -- (a) to[out=-90, in=180] (mu.center) to[out=0, in=-90] (b
            ) -- (tr);
       \draw (mu) -- (bl);
       \draw[name path=diag] (tll) to[out=-90, in=90] (c) -- (br);
       \path[name intersections={of=cup and diag}] coordinate[dot, label=-135:$\lambda$] (
            lambda) at (intersection-1);
       \path[name intersections={of=cup and diag}] coordinate[dot, label=45:$\lambda$] (
            lambda') at (intersection-2);
       \begin{scope}[on background layer]
          \fill[catd] (tlll) rectangle (brr);
          \fill[catc] (tll) to[out=-90, in=90] (c) -- (br) -- (brr) -- (brr |- tll) --
               cycle;
       \end{scope}
   \end{tikzpicture}
```

The method for this diagram is broadly similar to the previous simpler examples. We have two main components is our diagram, a "tuning fork" shape in the middle of the diagram, and diagonal curve running from the top left to the bottom right. Individually, these are positioned and drawn using the previous techniques. The difficulty is finding the positions for the two λ two-cells. To do this, we must introduce the intersections library:

```
\usetikzlibrary{backgrounds, intersections}
```

As before, we place most of the coordinates, dots, and vertex labels in the first phase of our code. The code:

```
\draw[name path=cup] (tl) -- (a) to[out=-90, in=180] (mu.center) to[out=0, in=-90] (b) -- (
     tr);
```

draws the cup shape, writing `[name path=cup]` to give the path a name. Similar code draws and names the diagonal curve across the diagram. With this in place, the code:

```
\path[name intersections={of=cup and diag}] coordinate[dot, label=-135:$\lambda$] (lambda)
     at (intersection-1);
\path[name intersections={of=cup and diag}] coordinate[dot, label=45:$\lambda$] (lambda') at
      (intersection-2);
```

calculates coordinates for the two points at which `cup` and `diag` intersect. We have now interleaved the layout and drawing phases of our code. This could be avoided, but at the cost of some code duplication to repeat paths in the drawing phase. Interleaving layout and drawing seems the more satisfactory trade-off.

5 Alignment

String diagrams rarely appear in isolation. More commonly, they appear in equations, and if we're not careful we will get vertical alignment issues. For example, the code below:

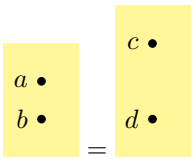
```
\begin{equation*}
    \begin{tikzpicture}[scale=0.5]
    \path coordinate[dot, label=left:$a$] (high) ++(0,-1) coordinate[dot, label=left:$b$] (
          low);
    \scoped[on background layer] \fill[catc] ($(low) + (-1,-1)$) rectangle ($(high) + (1,1)$)
          ;
    \end{tikzpicture}
    =
    \begin{tikzpicture}[scale=0.5]
    \path coordinate[dot, label=left:$c$] (high) ++(0,-2) coordinate[dot, label=left:$d$] (
          low);
    \scoped[on background layer] \fill[catc] ($(low) + (-1,-1)$) rectangle ($(high) + (1,1)$)
          ;
    \end{tikzpicture}
\end{equation*}
```

generates an unpleasant looking equation between two very simple string diagrams chosen to illustrate our point. The code `($(high)+ (1,1)$)` calculates a node one across and one up from `high` using the TikZ calculation library:

```
\usetikzlibrary{backgrounds, intersections, calc}
```

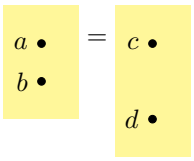
This library is very useful for calculating relative positions outside of path construction in a TikZ diagram.

The examples paid no attention to alignment issues, and the default behaviour is not particularly satisfactory.



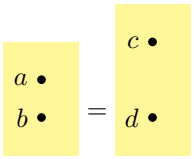
To get more control over how our diagrams are aligned, we can explicit set their baseline. By adding `[scale=0.5, baseline=(high)]` to the properties of both diagrams,

we set the baseline to the higher vertices, yielding the following effect:

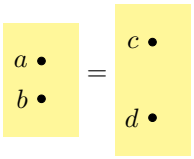


Notice this affects both how the diagrams are aligned relative to each other, and how LaTeX places the equals sign.

Similarly, setting `[scale=0.5, baseline=(low)]` sets the baseline to the lower vertices, yielding:

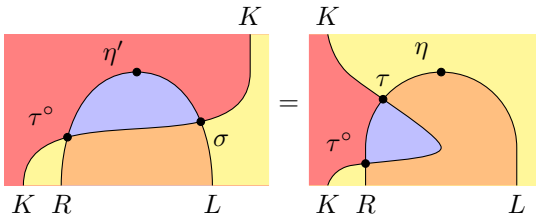


Using the calculation library, we can set the baseline to the midpoint of the two vertices in each diagram, setting `[scale=0.5, baseline=($(high)!0.5!(low)$)]`. This leads to probably the best choice of alignment in this simple case:



6 Something Fancy

We conclude with an equation between two quite complicated diagrams, which uses many of the techniques we have used previously.



The code uses much of what we have previously seen:

```
\begin{equation*}
  \begin{tikzpicture}[scale=0.5, baseline=($(bl)!0.5!(tr)$)]
    \path coordinate[label=below:$R$] (bm) ++(0,1) coordinate (a) ++(2,2) coordinate[dot,
          label=135:$\eta'$] (eta)
    ++(2,-2) coordinate (b) ++(0,-1) coordinate[label=below:$L$] (br)
    (bm) ++(-1,0) coordinate[label=below:$K$] (bl) ++(6,3) coordinate (c) ++(0,1) coordinate[
          label=above:$K$] (tr);
    \draw[name path=curv] (bl) to[out=90, in=-90] (c) -- (tr);
    \draw[name path=bend] (bm) to[out=90, in=180] (eta.center) to[out=0, in=90] (br);
```

```
    \path[name intersections={of=bend and curv}]
    coordinate[dot, label=135:$\tau^\circ$] (tauinv) at (intersection-1)
    coordinate[dot, label=-45:$\sigma$] (sigma) at (intersection-2);
    \begin{scope}[on background layer]
      \fill[cate] ($(bl) + (-0.5,0)$) rectangle ($(tr) + (0.5,0)$);
      \fill[catc] (bl) to[out=90, in=-90] (c) -- (tr) -- ++(0.5,0) -- ++(0,-4) -- cycle;
      \begin{scope}
        \clip (bm) to[out=90, in=180] (eta.center) to[out=0, in=90] (br) -- cycle;
        \fill[catf] ($(bl) + (-0.5,0)$) rectangle ($(tr) + (0.5,0)$);
        \fill[catd] (bl) to[out=90, in=-90] (c) -- (tr) -- ++(0.5,0) -- ++(0,-4) -- cycle;
      \end{scope}
    \end{scope}
  \end{tikzpicture}
  =
  \begin{tikzpicture}[scale=0.5, baseline=($(bl)!0.5!(top)$)]
    \path coordinate[label=below:$R$] (bm) ++(0,1) coordinate (a) ++(2,2) coordinate[dot,
         label=135:$\eta$] (eta) ++(2,-2)
    coordinate (b) ++(0,-1) coordinate[label=below:$L$] (br)
    (bm) ++(-1,0) coordinate[label=below:$K$] (bl) +(0.5,0.5) coordinate (bl') ++(0.5,3)
         coordinate (c) ++(-0.5,1) coordinate[label=above:$K$] (top)
    (eta) ++(0,-2) coordinate (centre);
    \draw[name path=bend] (bm) -- (a) to[out=90, in=180] (eta.center) to[out=0, in=90] (b) --
          (br);
    \draw[name path=curv] plot[smooth, tension=0.5] coordinates {(bl) (bl') (centre) (c) (top
         )};
    \path[name intersections={of=bend and curv}]
    coordinate[dot, label=135:$\tau^\circ$] (tauinv) at (intersection-1)
    coordinate[dot, label=above:$\tau$] (tau) at (intersection-2);
    \begin{scope}[on background layer]
      \fill[catc] ($(top) + (-0.5,0)$) rectangle ($(br) + (0.5,0)$);
      \fill[cate] plot[smooth, tension=0.5] coordinates {(bl) (bl') (centre) (c) (top)} --
           ++(-0.5,0) -- ++(0,-4) -- (bl);
      \begin{scope}
        \clip (bm) -- (a) to[out=90, in=180] (eta.center) to[out=0, in=90] (b) -- (br) --
             cycle;
        \fill[catd] ($(top) + (-0.5,0)$) rectangle ($(br) + (0.5,0)$);
        \fill[catf] plot[smooth, tension=0.5] coordinates {(bl) (bl') (centre) (c) (top)} --
             ++(-0.5,0) -- ++(0,-4) -- (bl);
      \end{scope}
    \end{scope}
  \end{tikzpicture}
\end{equation*}
```

Here we have used many of the previous methods:

- We specify `baseline` to get good alignment between the two diagrams and the equals sign between them.
- The calculation library is used for convenient coordinate calculations.
- The intersection library is used to calculate awkward points where edges coincide.
- The use of a background layer allows us to structure the code with the region filling after the layout and drawing code.

Another useful feature is the code

```
\lstinline{label={[yshift=-0.5ex]135:$\eta'$}]}
```

This places a label at 135 degrees from the current coordinate. The default position chosen by TikZ is too close to the top of the diagram as the prime makes the label quite tall, so we adjust this using the `yshift` feature. When diagrams become cramped, judicious use of such shifts can vastly improve readability.

The code also uses clipping to restrict the regions in some of the fills. It also uses the TikZ plot capability to draw a more complex curve. We refer readers to the manual for the details of these features, and to explore further.

7 Conclusion

We have only described the drawing of some basic diagrams in this note. Hopefully this will be useful to those new to laying out this type of diagram.

We do not claim the code used to produce the diagrams is the best TikZ code possible, although we would like to get as close as possible to this ideal. Feedback about better strategies and TikZ techniques that improves either the code or visual quality of our diagrams would be gladly welcomed.

References

[HM21] Ralf Hinze and Dan Marsden. *The Art of Category Theory. Part I: Introducing String Diagrams.* Cambridge University Press, 2021. To appear.

[HM22] Ralf Hinze and Dan Marsden. *The Art of Category Theory. Part II: Exploring String Diagrams.* 2022. In development.

[TM] Till Tantau and Henri Menke. *The TikZ and PGF Packages.*