SheetShow

Release 0.1.0

Aug 28, 2021

Contents

We explain here the YAML format used to represent the morphisms. The first step to understand the format is to explain the data structure we use to represent monoidal string diagrams. These are simpler than bimonoidal diagrams as they can be drawn in the plane.

CHAPTER 1

Data structure for monoidal string diagrams

A string diagrams in a monoidal category is in *general position* when no two nodes share the same height. Any string diagram can be put in general position without changing its meaning.

A diagram in general position can be decomposed as a sequence of horizontal *slices*, each of which contains exactly one node.

One can therefore encode the diagram as follows:

- The number of wires crossing the input boundary (or a list of the objects annotating them, in a typed context);
- The list of slices, each of which can be described by the following data:
- The number of wires passing to the left of the node in the slice. We call this the *offset*;
- The number of input wires consumed by the node;
- The number of output wires produced by the node (or again, their list of types).

Therefore, one can encode the sample diagram above as follows, assuming that inputs are at the bottom:

```
inputs: 2
slices:
- offset: 0
  inputs: 1
  outputs: 2
- offset: 1
  inputs: 2
  outputs: 1
- offset: 0
  inputs: 2
  outputs: 1
```

Each slice can be augmented to store details about the morphism in that slice (such as a label, for instance). This data structure is well suited to reason about string diagrams and there are efficient algorithms to determine if two diagrams are equivalent up to exchanges.

For more details about this data structure, refer to:

- Antonin Delpeuch and Jamie Vicary. Normalization for planar string diagrams and a quadratic equivalence algorithm, 2018, arXiv:1804.07832.

CHAPTER 2

Adding symmetry

To obtain a language for symmetric monoidal categories, we do not need to add much. We can simply allow a particular type of slice which represent a swap of two adjacent wires. To simplify the format, instead of writing the swap fully as:

```
offset: 3
inputs: 2
outputs: 2
```

We introduce a shorter notation, which at the same time encodes the particular role of the symmetry:

```
swap: 3
```

It now becomes possible to encode symmetric monoidal diagrams:

```
inputs: 2
slices:
- offset: 0
  inputs: 1
  outputs: 2
- swap: 1
- offset: 1
  inputs: 2
  outputs: 1
- offset: 0
  inputs: 2
  outputs: 1
```

Which can be rendered as:

CHAPTER 3

Bimonoidal diagrams

Sheet diagrams in bimonoidal categories are obtained by extruding symmetric monoidal string diagrams for the additive monoidal structure $(\mathcal{C}, \oplus, O)$. Therefore our data structure for bimonoidal diagrams is based on that for monoidal diagrams.

A bimonoidal diagram is described by:

- The number of input sheets, and the number of input wires on each of these input sheets (in the example above: *[1, 2, 2]* at the bottom);
- The slices of the bimonoidal diagram, which are seams between sheets. They are each described by:
  - The number of sheets passing to the left of the seam. We call this, again, the *offset*;
  - The number of input sheets joined by the seam;
  - The number of output sheets produced by the seam;
  - The nodes present on the seam.

Each seam can have multiple nodes on it. Each of these can connect to some wires on each input sheet (not necessarily the same number of wires for each input sheet) and similarly for output sheets. We describe them with the following data:

- The number of wires passing through the seam without touching a node, to the left of the node being described. We call this the *offset* of the node;
- For each input sheet, the number of wires connected to the node;
- For each output sheet, the number of wires connected to the node.

Which is encoded in YAML as:

```
inputs:
- 1
- 2
- 2
```

(continues on next page)

(continued from previous page)

```
slices:
- offset: 1
  inputs: 1
  outputs: 2
  nodes:
  - offset: 0
    inputs:
    - 1
    outputs:
    - 1
    - 1
- offset: 2
  inputs: 2
  outputs: 2
  nodes:
  - offset: 0
    inputs:
    - 2
    - 2
    outputs:
    - 1
    - 1
```

View in SheetShow.

CHAPTER 4

Symmetry for the multiplicative structure

Symmetry is also supported for the multiplicative (although it is not required in bimonoidal categories in general). It can be used as a special node (for which you will need to introduce a seam):

```
inputs:
- 2
slices:
- offset: 0
  inputs: 1
  outputs: 1
  nodes:
  - swap: 0
```

View in SheetShow.

As for the additive symmetry, the value of the *swap* attribute is the offset of the corresponding node.

CHAPTER 5

Typing and labeling

Rendering the geometry of a diagram is not enough: we also want to be able to annotate its nodes and and edges with morphisms and objects.

At the moment only wires on the diagram boundary can be labeled by objects. For inputs, it is done by replacing the number of wires on an input sheet by the list of types, as follows:

```
inputs:
- [A]
- [B,C]
- [D,E]
```

or equivalently:

```
inputs:
- - A
- - B
  - C
- - D
  - E
```

The same syntaxes can be used for outputs:

```
outputs:
- []
- [X,Y,Z]
- [F,F]
```

Each node can be labeled by adding a *label* key in it:

```
offset: 0
inputs:
  - 2
  - 2
outputs:
```

(continues on next page)

(continued from previous page)

```
  - 1
  - 1
label: g
```

CHAPTER 6

Embedding diagrams in LaTeX

We explain here the recommended workflow to embed diagrams generated by SheetShow in LaTeX documents.

6.1 Workflow overview

- Compose your diagram in SheetShow. If you need to use mathematical expressions in labels, use *label: α*. The two dollars enable the math mode.
- Export your diagram as SVG in SheetShow.
- Use Inkscape to convert your diagram to LaTeX+PDF
- Embed the generated LaTeX file in your document with *\input*.

6.2 Inkscape

Inkscape is an open source vector graphics editor, available on most platforms. It can be used as a command line tool to convert between different vector and raster image formats. In our case, we use it to convert from SVG to LaTeX as follows (for Inkscape 1.0 and above):

```
inkscape -D sheet_diagram.svg --export-filename=sheet_diagram.pdf --export-latex --
↪export-area-drawing
```

Before Inkscape 1.0, the syntax was:

```
inkscape -D -z --file=sheet_diagram.svg --export-pdf=sheet_diagram.pdf --export-latex␣
↪--export-area-drawing
```

When invoked like this, Inkscape will generate two files:

- One PDF file, which contains all the surfaces and paths in the SVG, without the text nodes;
- One LaTeX file, which imports the PDF and adds the text nodes on top of it.

Let us take the following diagram as example:

It is converted to a PDF file where different layers in the picture are split into pages, and the accompanying LaTeX code is generated:

```
%% Creator: Inkscape inkscape 0.92.4, www.inkscape.org
%% PDF/EPS/PS + LaTeX output extension by Johan Engelen, 2010
%% Accompanies image file 'sheet_diagram.pdf' (pdf, eps, ps)
%%
%% To include the image in your LaTeX document, write
%%   \input{<filename>.pdf_tex}
%%  instead of
%%   \includegraphics{<filename>.pdf}
%% To scale the image, write
%%   \def\svgwidth{<desired width>}
%%   \input{<filename>.pdf_tex}
%%  instead of
%%   \includegraphics[width=<desired width>]{<filename>.pdf}
%%
%% Images with a different path to the parent latex file can
%% be accessed with the `import' package (which may need to be
%% installed) using
%%   \usepackage{import}
%% in the preamble, and then including the image with
%%   \import{<path to file>}{<filename>.pdf_tex}
%% Alternatively, one can specify
%%   \graphicspath{{<path to file>/}}
%%
%% For more information, please see info/svg-inkscape on CTAN:
%%   http://tug.ctan.org/tex-archive/info/svg-inkscape
%%
\begingroup%
\makeatletter%
\providecommand\color[2][]{%
    \errmessage{(Inkscape) Color is used for the text in Inkscape, but the package
↪'color.sty' is not loaded}%
    \renewcommand\color[2][]{}%
}%
\providecommand\transparent[1]{%
    \errmessage{(Inkscape) Transparency is used (non-zero) for the text in Inkscape,␣
↪but the package 'transparent.sty' is not loaded}%
    \renewcommand\transparent[1]{}%
}%
\providecommand\rotatebox[2]{#2}%
\newcommand*\fsize{\dimexpr\f@size pt\relax}%
\newcommand*\lineheight[1]{\fontsize{\fsize}{#1\fsize}\selectfont}%
\ifx\svgwidth\undefined%
    \setlength{\unitlength}{159.7625407bp}%
    \ifx\svgscale\undefined%
    \relax%
    \else%
    \setlength{\unitlength}{\unitlength * \real{\svgscale}}%
    \fi%
\else%
    \setlength{\unitlength}{\svgwidth}%
\fi%
\global\let\svgwidth\undefined%
```

(continues on next page)

(continued from previous page)

```
\global\let\svgscale\undefined%
\makeatother%
\begin{picture}(1,1.04467035)%
    \lineheight{1}%
    \setlength\tabcolsep{0pt}%
    \put(0,0){\includegraphics[width=\unitlength,page=1]{sheet_diagram.pdf}}%
    \put(0.10097764,1.00588598){\color[rgb]{0,0,0}\makebox(0,0)[lt]{\lineheight{1.25}
↪\smash{\begin{tabular}[t]{l}$A$\end{tabular}}}}%
    \put(0,0){\includegraphics[width=\unitlength,page=2]{sheet_diagram.pdf}}%
    \put(0.71548248,0.25671683){\color[rgb]{0,0,0}\makebox(0,0)[lt]{\lineheight{1.25}
↪\smash{\begin{tabular}[t]{l}$g$\end{tabular}}}}%
    \put(0,0){\includegraphics[width=\unitlength,page=3]{sheet_diagram.pdf}}%
    \put(0.51339268,1.00002171){\color[rgb]{0,0,0}\makebox(0,0)[lt]{\lineheight{1.25}
↪\smash{\begin{tabular}[t]{l}$C$\end{tabular}}}}%
    \put(0,0){\includegraphics[width=\unitlength,page=4]{sheet_diagram.pdf}}%
    \put(0.4355847,0.65279324){\color[rgb]{0,0,0}\makebox(0,0)[lt]{\lineheight{1.25}
↪\smash{\begin{tabular}[t]{l}$f$\end{tabular}}}}%
    \put(0,0){\includegraphics[width=\unitlength,page=5]{sheet_diagram.pdf}}%
    \put(0.42783026,0.98479786){\color[rgb]{0,0,0}\makebox(0,0)[lt]{\lineheight{1.25}
↪\smash{\begin{tabular}[t]{l}$B$\end{tabular}}}}%
    \put(0,0){\includegraphics[width=\unitlength,page=6]{sheet_diagram.pdf}}%
    \put(0.92302724,0.97646438){\color[rgb]{0,0,0}\makebox(0,0)[lt]{\lineheight{1.25}
↪\smash{\begin{tabular}[t]{l}$E$\end{tabular}}}}%
    \put(0.85543138,0.95720957){\color[rgb]{0,0,0}\makebox(0,0)[lt]{\lineheight{1.25}
↪\smash{\begin{tabular}[t]{l}$D$\end{tabular}}}}%
\end{picture}%
\endgroup%
```

We can then include the diagram in a LaTeX document as:

```
\input{sheet_diagram.pdf_tex}
```

This produces the following output:

The generated LaTeX file require the *graphicx* and *color* packages. To render a minimal document with LaTeX, you can use the following template:

```
\documentclass{standalone}
\usepackage{graphicx}
\usepackage{color}
\begin{document}
\input{sheet_diagram.pdf_tex}
\end{document}
```

6.3 Automation

It is reportedly possible to automate the conversion, making it possible to include the SVG files in LaTeX directly. For this, use the svg package from CTAN, which wraps Inkscape nicely. You will need to run *pdflatex* in *–shell-escape* mode for it to be able to call Inkscape.

6.4 Alternative workflows

We could also render mathematical formulae in JavaScript. MathJax supports SVG rendering, KaTeX does not yet.

This would have the advantage of producing more readable rendering in web pages.

CHAPTER 7

Embedding diagrams in web pages

We explain here the recommended workflow to embed diagrams generated by SheetShow in HTML pages. This will let your readers rotate the diagrams directly inside the page.

7.1 Workflow overview

- Compose your diagram in SheetShow.
- Add SheetShow as a script to your HTML page with *<script type="text/javascript" src="https://wetneb.github.io/sheetshow/embed/sheetshow.js"></script>*.
- Copy the YAML representation of your diagram inside a *<script type="text/sheetshow"></script>* tag, at the location in your document where you want the image to be inserted. You can use many of these in the same page.

This will insert SVG images in place of all the scripts you inserted, animated by Javascript.

See this example for a demo.

7.2 Configuration options

It is possible to set the width and height of the generated SVG by setting the *data-width* and *data-height* parameters on the *<script type="text/sheetshow"></script>* tag.

7.3 Alternative workflows

You can also save the diagram as an SVG file from SheetShow and include it as an image anywhere else. This will not be animated but has the benefit of not loading Javascript code.

CHAPTER 8

Rendering

We describe the architecture behind the renderer.

8.1 Monoidal diagrams

We use the following layout strategy to compute the geometry of monoidal diagrams. Each diagram slice has a fixed height: the horizontal spacing between two adjacent nodes is constant. Each edge is drawn as a straight vertical wire, except towards the ends where splines bind them to the nodes they connect to.

To determine the horizontal position of nodes and edges, we generate a system of linear equations and inequations. We assign a variable to each node and each edge, denoting their horizontal positions. We also add variables for the positions of the left and right boundaries:

We then generate the system of constraints using the following rules:

- At a given level in the diagram, between two consecutive slices, consecutive wires must be spaced by at least a fixed margin. The first and last wire must also be spaced away from the boundary of the drawing rectangle.
- Each node must be positioned at the average position of all its input wires, which must also be equal to the average position of all its output wires.
- Nodes with no inputs or outputs (known as *scalars*) must be located between the wires passing to the left and to the right of them.

We minimize the sum of all variables involved under these constraints.

For the diagram above, this gives the following constraint system:

$$
\begin{aligned}
lb + m &\leq e_0 \\
e_0 + m &\leq rb \\
v_0 &= e_0 \\
v_0 &= \frac{e_1 + e_2}{2} \\
lb + m &\leq e_1 \\
e_1 + s &\leq e_2 \\
e_2 + m &\leq rb \\
v_1 &= e_2 \\
v_1 &= \frac{e_3 + e_4}{2} \\
e_1 + s &\leq e_3 \\
e_3 + s &\leq e_4 \\
e_4 + m &\leq rb \\
v_3 &= \frac{e_1 + e_3}{2} \\
v_3 &= e_5 \\
lb + m &\leq e_5 \\
e_5 + s &\leq e_4
\end{aligned}
$$

This system can then be solved using an off-the-shelf solver (we use Glpk.js currently). The solutions to the system determine the horizontal positions of all objects in the diagram: we just need to link up edges to nodes using Bezier curves, and we can render all this in SVG.

It would be nice to have a proof that the constraint system is always satisfiable.

8.2 Bimonoidal diagrams

The basic 3D shape of bimonoidal diagrams is obtained by extruding a monoidal diagram, so we can use the same constraint system to compute this basic layout. Then, we need to determine the position of paths on the sheets. This is again done using a constraint system, which can be solved independently of the first one.

Again, we assume that paths are straight lines drawn on the sheets, except at the ends where Bezier curves are used to link them to the nodes they connect to. We add the same spacing constraint on any two paths embedded in the same sheet. To link paths to vertices, we have two options:

- the *strict mode*, where nodes are required to be positioned at the average of paths connecting to it in a given sheet, for all input and output sheets;
- the *lax mode*, where instead nodes are required to be positioned at the average of all their input paths which must be equal to the average of all their output paths.

The strict mode generally gives visually pleasing results, but it is not solvable for all diagrams. Consider the following diagram:

It would be impossible to lay out this diagram with the strict mode.

When rendering a diagram, the renderer tries to solve its strict constraint system first, and falls back on the lax constraint system if that fails.

It would be nice to have a proof that the lax constraint system is always satisfiable, or a counter-example of that.

8.3 Geometry rendering

Once both constraint systems have been generated and solved, we generate the 3D geometry according to the coordinates induced by the solutions. To generate 3D models, Bezier curves are discretized to generate faces.

We then render the model with the Seen 3D renderer, which produces an SVG image of the model.

CHAPTER 9

Building

9.1 Getting started

To make changes to the application, you will need Node.js and NPM, its package manager. Install them here.

9.2 Running locally

To run the application locally, use:

```
npm start
```

This will spin a development web server and you will be able to view the application at http://localhost:8080/. The application will reload automatically when you change files.

9.3 Deploying

To publish the app to GitHub Pages, use:

```
npm run deploy
```

This will compile the app and upload it to the *gh-pages* branch.

9.4 Compiling the embeddable script

To compile the embeddable script used to render diagrams in third-party web pages, run:

```
npm run embed
```

This will generate the script at *dist/embed/sheetshow.js*.

CHAPTER 10

Testing

SheetShow comes with a test suite for the underlying data structures used to represent the diagrams and compute the layout. It is written with the Jest framework.

To run the tests once, use:

```
npm test
```

To run the tests continuously, as you change the files, use:

```
npm test -- --watch
```

CHAPTER 11

Documenting

This manual is written using Sphinx and the source files can be found in the *docs* folder of the repository for this application. Any contributions to the docs are of course most welcome, as are any suggestions to improve the coverage of a particular subject.

To build the docs, you first need to install Sphinx, preferably in a Python virtualenv:

```
python3 -m venv .venv
source .venv/bin/activate
pip install sphinx
```

You can then go to the docs and generate the HTML:

```
make html
```

The docs are then available in *docs/_build/html/*. Docs are built automatically when pushed to the repository and are then available on ReadTheDocs, so you only need to compile them locally if you want to have a preview of what they will look like after pushing them.