Rendering string diagrams with Haskell's Diagrams library

by

Celia Rubio Madrigal

In partial fulfilment of the requirements
for the degree of MSc in
Advanced Computer Science with Artificial Intelligence

University of Strathclyde Glasgow

Department of
Computer and Information Sciences

August, 2023

Abstract

String diagrams are a graphical language used to represent any process that can be composed sequentially or run in parallel, which translates graphically to joining them horizontally or vertically. This project introduces a new Haskell-based tool for rendering string diagrams using the Diagrams library. The key innovation is storing the diagrams in binary space-partition trees, depending on their horizontal or vertical composition, and rendering them through recursive folding. This allows preserving locality principles and mirroring their mathematical definition. The recursion is achieved by maintaining a right trapezoidal shape for the diagram's outline as an invariant. The renderer supports adding semantics to diagrams to serve as a compiler, with an example for matrices given. It also provides a flexible interface for users to implement new monoidal categories and their diagrams' appearance. By comparing this tool to DisCoPy, the advantages of the new data structure are highlighted: the recursive rendering allows diagrams to be treated more mathematically as algebraic objects. This, in turn, can increase access to drawing string diagrams for applied category theory researchers.

Declaration

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc in Advanced Computer Science with Artificial Intelligence of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself.

Following normal academic conventions, I have made due acknowledgement to the work of others.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

Yes [X]   No [   ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices) is 13549.

I confirm that I wish this to be assessed as a Type 1 2 [✓ 3] 4 5 Dissertation.

Signature: Celia Rubio Madrigal

Date: August 14th, 2023

Acknowledgements

I wish to express my gratitude to my parents for their patience in listening to my endless technical conversations, and for providing much-needed pictures of my *perro* whenever I asked for them.

I owe much appreciation to my friends for the companionship and joyful times throughout this past year, with particular thanks to Girish, Zerksis, Austin, and Alicia.

I am also thankful to my supervisor, Jules Hedges, for introducing me to this compelling research area —delving into it has proven to be very rewarding.

This project has received the support of a fellowship from ”la Caixa” Foundation (ID 100010434, with fellowship code LCF/BQ/EU22/11930080). Without their generous support, pursuing this master’s degree would not have been possible.

Contents

List of Figures

List of Code snippets

1 Introduction

String diagrams are a 2-dimensional graphical language comprising ‘boxes’ that represent processes, and ‘wires’ coming to and from them that represent their input and output objects. These processes can be either stacked sequentially or run in parallel to form other processes. The geometry of these diagrams encodes properties of the processes they depict.

In mathematical terms, string diagrams are a representation of free monoidal categories, a result formally proven in the Joyal-Street coherence theorem (Joyal and Street, 1991). Many areas in applied computer science and mathematics are progressively being formalised using category theory; in these areas, string diagrams are often employed to both better understand the objects they work with, and once formalised, to construct mathematical proofs with them.

One significant benefit of applied category theory is its ability to identify similar structures across many disciplines that may then share a common language. In the case of monoidal categories, that language can be string diagrams. It could be very advantageous for researchers to access a common front-end interface for string diagrams, which would allow for different implementations of domain-specific back-end applications. Each application could add concrete semantics to the generating elements of a diagram, such that the meaning of the full object can be derived from them in a recursive manner.

There are already some software tools that manipulate, rewrite and normalise string diagrams, in a similar way that other proof assistants, such as Coq (Team, 2022), operate with conventional logic expressions. These tools, however, are restricted to working solely with the syntax of these diagrams; they usually lack the ability to incorporate semantics into the constructions and compile them depending on the given category at hand. Moreover, their functionality is usually limited to specific types of monoidal categories, such as compact closed categories for the tool Quantomatic (Kissinger and Zamdzhiev, 2015).

A significant distinction between the currently available software and a tool that would act as a compiler is the data structure that would store the diagram. Rather than working directly with graphs, it has been suggested to store them in recursive structures such as k-d trees (Hedges and Herold, 2019). The goal of this project is to expand on that idea by developing a tool with those characteristics. This tool is implemented in Haskell and uses the Diagrams library (Manual, 2023) to render the string diagrams' images. The source code is available in the following repository:

`https://github.com/celrm/stringdiagrams`

1.1 Research goals

This project's main aim is to develop a new tool for rendering string diagrams using Haskell's Diagrams library. The tool addresses limitations in existing software, providing more flexibility for researchers who use monoidal categories in their fields. It can also act as a compiler for these diagrams, supporting the addition of semantics onto them.

In addition to using a different programming language, this renderer diverges fundamentally from traditional methods in that the diagrams are stored in binary space partition trees (De Berg et al., 1997) in a fully recursive manner. As such, the primary research question of the project explores the viability of generating string diagrams simply by folding over binary tree structures.

The Diagrams library provides the `|||` operator for horizontally joining diagrams, and `===` for its vertical counterpart. Our goal is to structure the code to closely mirror the pseudocode shown in Snippet 1, as deviating from it reduces its mathematical equivalence. In section 3.3.3 we show that the actual code closely resembles this structure, although the subdiagrams must also be globally deformed to maintain their recursion invariants.

Preserving locality as much as possible is crucial —i.e., local information from a node should not be shared with other parts of the diagram where it does not belong. Each leaf in the tree should render only its components, without knowing where they will be placed higher up in the tree. Subdiagrams can only be modified in a global, general way, irrespective of the details they contain.

```
render (Leaf l) = draw l
render (Compose d1 d2) = d1 ||| d2
render (Tensor d1 d2) = d1 === d2
```

Snippet 1: Pseudocode for the main fold

This aim is accomplished by maintaining an invariant on the shape of the diagrams' outline: a right trapezoid. This recursive outline is called the *brick diagram* counterpart of the string diagram. In our case, the brick diagrams have the external shape of a right trapezoid with its two right angles facing down. We have developed the mathematical formulas of the transformations that preserve this shape when both tensoring and composing diagrams.

A secondary objective of this project is to provide a highly customisable codebase, so that users can implement new categories and visual styles as needed. The flexibility is achieved through an accessible interface design, composed of generalisable classes and data types. This allows, amongst other features, to have smooth connections and labelling texts.

Modularity and code reusability are prioritised to enable other researchers to build upon this framework for their own back-end software applications. Additionally, this project includes example templates for users to use as a reference when implementing them, such as calculating the semantics of a string diagram in the category of matrices with direct sum.

1.2 Overview

Section 2 provides all the necessary context and prerequisites for understanding this project. It begins by explaining the relevant mathematical concepts of string diagrams (Subsection 2.1). Next, it reviews the landscape of existing software tools for rendering diagrams, situating the current work in the literature (Subsection 2.2). Finally, it provides an in-depth look at the Diagrams library, whose internal functions for diagram construction will be thoroughly used throughout the project (Subsection 2.3).

Section 3 then outlines the steps taken to meet the project requirements. It starts by detailing the software development process (Subsection 3.1). The mathematical formulas that enable brick diagrams to be combined are described next (Subsection 3.2). The section concludes by showcasing the core algorithm, from parsing the diagram tree structure to executing the final render recursively with a fold (Subsection 3.3).

Section 4 details the implementation of the two core functionalities of the project: drawing diagrams (Subsection 4.1) and compiling diagrams (Subsection 4.2). It also describes testing of the codebase on randomly generated diagram examples to verify correctness (Subsection 4.3). Additionally, it provides a thorough comparative analysis between our approach and the most similar existing tool in the literature, DisCoPy (Subsection 4.4).

Finally, section 5 summarises the outcomes and implications of the project. It begins by presenting the resulting Cabal package that encapsulates the algorithm (Subsection 5.1). Recommendations for future enhancements are then suggested (Subsection 5.2). The section concludes with final commentary on the achievements of the project (Subsection 5.3).

2 Background analysis

2.1 Mathematical formalisation of string diagrams

String diagrams are a form of diagrammatic calculus —not only do they allow for a conceptual understanding of what composing processes mean, but they are also a proper mathematical tool with which one can generate equations and proofs (Coecke et al., 2015). Their formal backbone lies in category theory; specifically, they form a monoidal category.

A monoidal category $\mathcal{V} = (\mathcal{V}_0, \otimes, I, a, l, r)$ is a category $\mathcal{V}_0$ equipped with a functor $\otimes : \mathcal{V}_0 \otimes \mathcal{V}_0 \to \mathcal{V}_0$ called a tensor, together with an identity object I and the rules for how they all behave (Kelly, 1982). Objects in $\mathcal{V}_0$ are wires in string diagrams that run from left to right. Morphisms in $\mathcal{V}_0$ correspond to the boxes or processes, so their composition accounts for sequential or *horizontal* stacking of diagrams. The tensor functor or product, then, allows for the parallel or *vertical* stacking of diagrams.

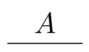
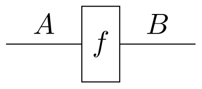
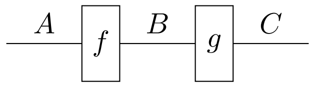
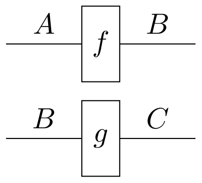
Object	A	
Morphism	$f : A \to B$	
Composition	$g \circ f$	
Tensor product	$g \otimes f$	

Figure 1: The graphical language of monoidal categories (Selinger, 2010)

The rest of the elements ensure that these operations are coherent with each other. The natural isomorphism a provides associativity to $\otimes$, while r and l act as neutral operations to the identity object I: for each $X \in V_0$ there are $r_X : X \otimes I \to X$ and $l_X : I \otimes X \to X$. In string diagrams, I is denoted by an absence of wire, as it can be integrated into everything else.

These rules conform to what we expect from the drawings: we can extend the length of wires, and, most importantly for this project, we can stack the same diagram in different orders. That is to say, the forming equations are equal as long as the resulting diagram remains the same, regardless of the order in which one chooses to stack the horizontal and vertical components while constructing it.

These operations correspond to a common data structure in computational geometry called binary space partition trees (De Berg et al., 1997), which recursively partition the space by hyperplanes. Specifically, k-d trees restrict these hyperplanes to those parallel to the coordinate axes.

In the two-dimensional space, tracing these hyperplanes —in this case, lines— results in the Poincaré dual of the string diagram, which are called brick diagrams (Hedges and Herold, 2019). These are rectangular tilings where the regions symbolize the morphisms, and the segments on their sides are their domain and codomain objects.

This project employs brick diagrams to recursively render string diagrams inside them.

However, their shape must be adapted as needed to maintain the recursion, therefore losing its rectangular shape for a trapezoid instead. We explain this invariant in section 3.2.

2.1.1 Extensions and applications

There can be several extensions of string diagrams constructed by allowing extra structures on top of the basic definition (Selinger, 2010). For example, braided monoidal categories add a *braiding* structural element that exchanges two wires. Balanced monoidal categories add a *twist* that makes wires upside down as if they were ribbons. Symmetric monoidal categories add a *crossing* element where the order of exchange is not important.

Other types of categories allow for wires to run in other directions that are not left to right. Traced categories let processes have loops. Autonomous or rigid categories let all objects have duals, so that their corresponding wires can run from right to left. This is reflected in the diagrams by half turns, one in each direction, which are called *unit* and *counit*. Many other extensions make use of geometric notions, such as pivotal for rotations, spherical for embeddings in 2-spheres, or dagger categories for mirror reflections. Finally, product categories are symmetric monoidal categories with *copy* and *delete* operations, while coproduct categories can apply *merge* and *create* operations to their objects.

The different types of monoidal categories appear in a broad range of applications. Even some of them have been identified independently more than once, leading to distinct naming conventions being used to refer to them. This is one of the reasons why a compiler for string diagrams that is flexible enough for many of them might be useful.

Some of these widespread applications are in: electrical circuits (Fong, 2016), concurrency theory (Abramsky, 1996), functional programming (Riley, 2018), databases (Patterson, 2017), control theory (Baez and Erbele, 2015), dynamical systems (Vagner et al., 2015), Markov kernels (Fritz, 2020), Bayesian inference (Coecke and Spekkens, 2012), game theory (Ghani et al., 2018), machine learning (Fong et al., 2019), cog-

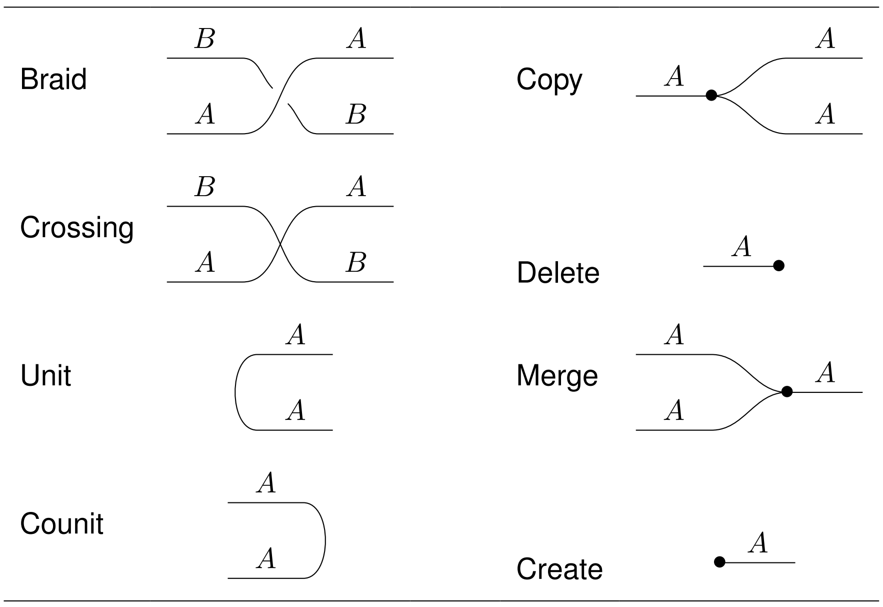


Figure 2: Some extensions for the graphical language of monoidal categories

nition (Bolt et al., 2017), and even the science of consciousness (Signorelli et al., 2021). Perhaps the most fruitful area has been quantum mechanics (Abramsky and Coecke, 2008; Coecke and Kissinger, 2018), with applications such as quantum natural language processing for music generation (Miranda et al., 2022) already being developed.

2.1.2 The monoidal category of matrices

Let us give an example of a symmetric monoidal category of great importance: the category of matrices. To fully characterize this category, we must enumerate its objects, morphisms, composition operation, and tensor product. Furthermore, since it is a symmetric category, we need to specify an element that denotes crossings of wires.

The objects of this category are simply the natural numbers. The morphisms are matrices with real number entries. A matrix of the form $n \times m$ corresponds to a morphism with a left arity (or domain) of n and a right arity (or codomain) of m. We refer to the arity of a morphism as the number of wires entering and leaving the morphism on each side. This terminology will also apply to any composite string diagram.

Let us now define the composition operation. If one string diagram has arity (n, k), to compose it with a second diagram, the second one must have the same domain k as the codomain of the first. Therefore, its arity will need to be of the form (k, m). This parallels the requirement for matrix multiplication, which will serve as our composition operation.

There are various ways to define a tensor product for matrices; each of them yields a different category. Depending on the application and field, some tensor products can be more useful than others. For instance, one possibility is the Kronecker product (Hedges and Herold, 2019).

We will take the tensor product to be the direct sum of two matrices. This operation places the two input matrices, m_1 and m_2, as blocks on the diagonal of the resulting matrix, with the two off-diagonal blocks containing only zeros. The arity of the result is the sum of the input arities. This also reflects graphically when placing one string diagram on top of another: the input wires are the sum of all inputs, and the output wires are the sum of all outputs.

A two-by-two crossing is thus represented as the permutation matrix $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. More generally, a wire crossing of any permutation of size n can be denoted by the corresponding $n \times n$ permutation matrix. This matrix is obtained by applying that permutation to the columns of the identity matrix.

Let us now check that this category fulfils some of the axioms mentioned in section 2.1. First, single wires can be extended. This corresponds to multiplying a matrix with the identity matrix of the corresponding size. Also, both operations are straightforwardly associative.

Finally, it is true that composing and tensoring matrices commute, in the sense that one can divide the operations vertically or horizontally as needed. Or, as we called it before, we can stack the same string diagram in different orders, which is key for the correct performance of our rendering algorithm. Denoting matrix multiplication by $\cdot$ and the direct sum by $\oplus$:

$$(A \cdot B) \oplus (C \cdot D) = \begin{pmatrix} A \cdot B & 0 \\ 0 & C \cdot D \end{pmatrix} = \begin{pmatrix} A & 0 \\ 0 & C \end{pmatrix} \cdot \begin{pmatrix} B & 0 \\ 0 & D \end{pmatrix} = (A \oplus C) \cdot (B \oplus D)$$

2.2 Existing software tools

There are currently some proof assistants that can manipulate string diagrams computationally. These tools internally use graph-based representations for the diagrams and work as graph rewriters. As an example, we have the ZX-calculus proof assistants Quantomatic (Kissinger and Zamdzhiev, 2015) and its successor PyZX (Kissinger and Van De Wetering, 2020), as well as the Catlab package for Julia (Halter et al., 2020). Since each node in the graph can be connected to any other, these tools are limited to compact closed categories, which is the common name for rigid symmetric monoidal categories. On the other hand, it is possible to go from compact closed to symmetric monoidal categories by restricting them to directed acyclic graphs (Hedges and Herold, 2019).

Several of the already mentioned application areas only make use of compact closed categories, such as in categorical quantum mechanics (Coecke and Kissinger, 2018) or distributional semantics (Coecke et al., 2010). Others only require the monoidal category to be symmetrical, such as in dataflows (Delpeuch, 2020) or game theory (Ghani et al., 2018). Finally, some non-symmetric monoidal categories might appear in linguistics, topology, and parts of theoretical physics (Hedges and Herold, 2019), for which the current software tools are not sufficient.

Even if the category is symmetric, there might be reasons to represent it as a planar graph plus a structural crossing element, and not as an arbitrary graph. It could happen that its symmetry is not computationally trivial (Hedges and Herold, 2019), so it could be useful to specify it in its syntax, and it will probably be more efficient to compile.

Besides the graph-based tools, there is another kind of software to represent string diagrams: Homotopy.io (Reutter and Vicary, 2019), which constructs morphisms in

```
sierpinski 1 = triangle 1

sierpinski n =      s
                   ===
                (s ||| s) # centerX

    where s = sierpinski (n-1)
```

Snippet 2: The Sierpinski triangle in the Diagrams library (Rosenbluth, 2015)

higher categories. Although it works well as a proof assistant, it may not be the right approach for compiling diagrams, and there may still be benefits to staying in two dimensions (Hedges and Herold, 2019).

The software that most closely resembles a string diagram compiler is DisCoPy (De Felice et al., 2021). DisCoPy is a Python library that uses free categories as a foundation, and allows for the inclusion of additional structures —every extension accounted in Selinger (2010) has been included since their latest release this year (Toumi et al., 2023). We make a more thorough comparison of DisCoPy to our tool in section 4.4.

2.3 The Diagrams library

The Diagrams library is an embedded domain-specific language (EDSL) for creating vector graphics in Haskell. It takes a declarative approach, meaning that instead of manipulating individual points when constructing diagrams, it focuses on the higher-level relationships between elements.

For instance, the code to generate a Sierpinski triangle is displayed in Snippet 2 (Rosenbluth, 2015). The code's structure clearly mirrors the shape of the output diagram. This illustrates Diagrams' expressiveness in resembling the visual form. Key to this is the introduction of the operators === and ||| for horizontally and vertically placing diagrams (Manual, 2023, 3.3: Juxtaposing diagrams) —precisely the functionality needed for drawing string diagrams.

Most diagram-like types in the library are instances of Monoid, implementing the `mappend` operator <> for concatenating diagrams (Manual, 2023, 2.1: Semigroups and monoids). With this operator, diagrams are usually combined by matching their local origins.

To translate the origin of a `HasOrigin` diagram, lower-level options like `translate`, `moveOriginTo`, and `moveOriginBy` are available. However, for global-style operations, the preferred approach is to align an `Alignable` object to one of its extremes (Manual, 2023, 3.4: Alignment). Since our diagrams will have their origins in the bottom left, we can use the `alignBL` function to align them to that corner.

To determine the alignment, the library associates an Envelope with each diagram (Manual, 2023, 4.1: Envelopes). The Envelope is an (extensional) function that takes any vector and returns the distance from the origin to a perpendicular tangent line to that vector. A limitation is that relying solely on these perpendicular distances does not always precisely match the true diagram boundary.

To address this, the concept of Trace is introduced (Manual, 2023, 4.2: Traces). For `Traced` objects, its extensional function computes the true distance from the origin to the boundary. This will be necessary for our drawing algorithm.

As an example, consider constructing a honeycomb pattern as: `honeycomb = h # alignR <> h # alignBL <> h # alignTL where h = hexagon 1` (Rosenbluth, 2015). There will be a gap between the left hexagon and the other two, because of misalignment between their envelopes. Rather than `align`, we can use the `snug` command to eliminate this gap.

Another important operator in Diagrams is `#`, which performs reverse function application (Manual, 2023, 2.3: Postfix transformation). It is commonly used to chain together additional styles or transformations as a form of post-processing after the core construction commands. This operator enables cleanly layering on operations to a base object.

Objects that are of class `Transformable` can be linearly transformed, while objects that are `Deformable` can be deformed more generally (Manual, 2023, 3.4: Deformations). While the library provides more capabilities for linear transformations, non-linear deformation is required for our algorithm. As diagrams cannot be deformed, we will be mainly working with paths, which are `TrailLike` objects (Manual, 2023, 3.5: Paths).

Other relevant classes include `Located` objects (Manual, 2023, 3.5: Located), for elements that are only translated and not deformed —although we will need to override their deformations— and `Text` for labeling (Manual, 2023, 3.7: Text).

Diagrams has a pluggable back-end system, enabling easy rendering to different formats by simply importing the desired back-end —such as cairo, postscript, html5, pgf, etc (Manual, 2023, 8: Rendering backends). For this project, we will use the default, out-of-the-box SVG back-end, which has no external dependencies and outputs .svg images. Using the SVG back-end is convenient since it requires no configuration. However, switching to an alternative back-end in the future could enable additional features, like user interaction with buttons or the screen.

We will be using version 1.4 of the Diagrams library. This library also supports drawing in three dimensions and producing animations. However, for this project we will focus solely on its two-dimensional drawing capabilities.

3 Methodology

3.1 The development process

The software methodology that was used for this project is incremental, evolutionary, vertical prototyping.

The prototype methodology is a software development approach which involves constructing a functional model of the software in a gradual, incremental manner, which can later be refined in order to satisfy new requirements. This enables the creation of a working model early on in the project timeline. Furthermore, it allows for quick identification of any choices that are not viable, therefore saving time given this project's constrained scope (Budde et al., 1992, p. 89-92).

Evolutionary prototyping defines a specific goal for the prototyping process, in which the system adapts to flexible constraints, and the developers construct successive pilot systems based on these constraints (Floyd, 1984, p. 11). While this is typically employed for projects with rapidly changing constraints, the requirements for this project were already known from the beginning. What was unknown at the start, however, was the range of mechanisms the library would provide for their implementation. As a result, we adapted to the limitations of the library at the same time as we gradually added capabilities to the system.

Our methodological approach is also built on the vertical prototyping paradigm. This means that we fully implemented each layer of functionality on its own, so the code for each pilot system has been modified in its entirety with each increment (Budde et al., 1992, p. 94). The layers of functionality, meaning the list of prerequisites that this project required, are:

- Draw a brick diagram by pure recursion
- Add labels at the centre of each region
- Add string diagrams on top of the brick diagrams
- Aesthetic improvements: smooth the connections between the curves
- Read diagrams from files
- Other base cases: crossings
- Other base cases: naming the wires
- Add compilation capabilities

3.1.1 Milestones

The first practical milestone to be reached, which directly addressed the main research question, was to construct a brick diagram using solely a recursive algorithm. In order to accomplish this objective, we found it necessary to devise a theoretical invariant for the diagram's overall shape, so that the operations would be successfully executed all the way through the recursion process. This invariant was designed as a right trapezoid; the required modifications for the recursive algorithm are detailed thoroughly in section 3.2. Once the theory was formalised, we then proceeded to coding the algorithm.

We opted to draw the brick diagram as a basic `Path V2 Double`, without any intermediary classes, simply using a straightforward fold as well as the operators `|||` and `<>` —instead of the operator `===`, as explained in section 4.1.1. This effort later led to what is now the `FoldableDiagram` class (detailed in section 3.3.3). At this stage of the project, the input trees were hard-coded as a user-defined type with a data declaration.

The next requirement was to append a descriptive text label for each morphism, so that they could be identified visually in the diagram. For this, we devised a custom diagram type, which formed the foundation for the current `LabelsDiagram` type (found in section 4.1.3). However, the original type directly inherited every necessary class from the Diagrams library, as there was no custom diagram class yet at that point.

An additional component of the diagram was the inclusion of the string diagram's wires and boxes within the brick diagram. This part closely resembled the prior implementation, but the wires get translated differently than the text labels and boxes. For the latter, only their origin gets translated without deformation, which can be done with the library's `Located` class (Manual, 2023, 3.5: Located). However, the wires get deformed in their totality, akin to the external brick diagram.

With both the brick and string diagrams combined in the same drawing, the next step was to enable the user to toggle between them. Initially, this was accomplished by separating out the brick diagram, the wires, the boxes, and the labels into four distinct attributes.

After isolating the wires from the rest as a `Path V2 Double` attribute, we realised we needed to deform them slightly differently in comparison to the brick diagram. This was because the connections between regions were jagged rather than smooth. We rectified this issue by rendering all wire segments as cubic splines, with their two control points horizontally aligned with the two endpoints. This enables seamless connections at 180 degrees, as seen in Figure 7.

At this stage in the project, the diagrams rendered aesthetically, with an option to select either brick or string diagrams. Therefore, the primary functionality for drawing had been implemented.

The Read module was then created (section 3.3.1) to enable automatic processing and thorough testing of additional examples. The chosen format for the input files was JSON, as recommended by Hedges and Herold (2019).

Once reading trees from external files was enabled, we were then able to add base cases beyond just simple morphisms. We first added wire crossings of any arbitrary permutation possible. We also incorporated the capability to annotate each wire in the diagram with a specific name.

Up to this point, we had used a custom type inheriting classes directly. Our goal was to create a tool enabling users to implement their own diagram format, while maintaining the fold, compose, and tensor operations, and abstracting away the rest. To accomplish this, we built the `FoldableDiagram` typeclass, which empowers users to create their own diagram types by simply defining instances of it and providing the necessary functions.

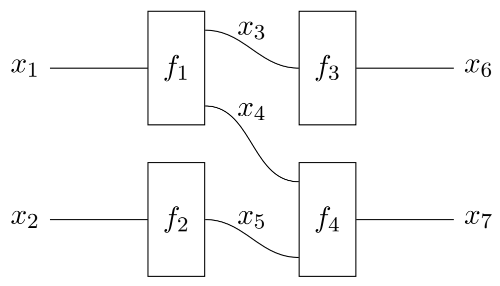
While `FoldableDiagram` can be instantiated as needed, it always requires an underlying brick diagram to compute dimensions and position regions appropriately. Therefore, we introduced a wrapping class `BrickWrapper` that automatically inherits `FoldableDiagram`, contains the brick diagram path, and makes it directly accessible in its implementation. Examples of it can be seen in appendix A.

As a final step, we incorporated the capability for users to carry along a monoidal category, whose semantics get computed simultaneously as the diagram is rendered. This was the last feature to implement for the library, exceeding current drawing capabilities described in the literature. The wrapper that unites all functionalities is the class `MonCatWrapper` (section 4.2.1), with examples of its usage presented in appendix B.

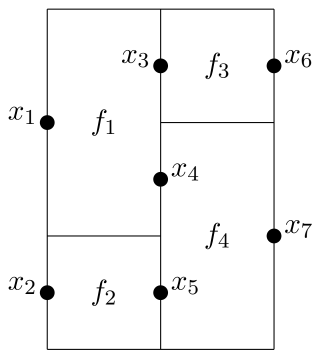
3.2 Designing brick diagrams

Before starting the implementation, we need to identify the properties of a diagram that will remain unchanged throughout the recursive process. Brick diagrams have to satisfy the following considerations:

- The base cases, i.e. the morphisms, are quadrilateral shapes with nothing else



(a) String diagram



(b) Brick diagram

Figure 3: Equivalent representations of a string diagram and a brick diagram from Hedges and Herold (2019)

inside. We call them regions. They compose horizontally and tensor vertically with each other.

- A brick diagram composes horizontally with another if and only if the first diagram's codomain matches the second's domain, and a region on the left adjoins another on its right if and only if they are truly connected in the diagram.

  As an example, we take Figure 3 from Hedges and Herold (2019), where f_1 is connected to f_3 and f_4 in the string diagram in 3a. Therefore, it only touches the vertical sides of those regions in the equivalent brick diagram in 3b.

In order for a recursive algorithm to execute over a brick diagram, certain conditions must persist at each recursive step. The subdiagrams involved in the process cannot have any information of their eventual positioning or connections within the larger structure. However, external modifications can be made to the diagrams as long as their internal composition remains unseen. To enable this isolated recursion, an invariant must be upheld regarding the shape of each diagram. By enforcing this consistency, changes can be safely made externally without compromising the integrity of the recursive process overall, or revealing internal details, which would break its locality.

To enable proper composition and tensoring operations, three key invariants must be upheld for each diagram:

1. All diagrams must have the shape of a right trapezoid, with right angles posi-

tioned on the bottom side. Essentially, this means a rectangle that has had its top side slanted at any angle to become a trapezoid.

2. The length of the trapezoid's left side must correspond to the diagram's left arity value, in distance units. Likewise, the length of the right side equals the diagram's right arity value.

3. The coordinate origin is fixed on the bottom left vertex of the trapezoid.

Notably, the width of the quadrilateral is not specified, and must be calculated separately whenever needed and taken into account for all formulas.

By enforcing these shape, side length, and origin invariants, the recursive composition and tensoring of diagrams becomes possible. The trapezoid shape invariant guarantees connections can be made cleanly while composing and tensoring. Meanwhile, the side length and origin invariants provide a consistent coordinate framework for the operations. Together they ensure the integrity of the recursion is maintained without exposing internal details prematurely.

However, to enable the parent diagram to fulfil the same invariants, the child diagrams will need to undergo certain transformations during these processes.

3.2.1 Tensoring

When tensoring two right trapezoidal diagrams, the bottom side of the first diagram, which is originally flat with width w_1, needs to be reshaped to match the slanted top side of the second diagram, if we assume it has a width of w_2. To achieve this reshape, a two-step transform can be applied: an X-scaling followed by a Y-shear, as illustrated in Figure 4.

The shear factor for the Y-shear is calculated as $\dfrac{\ell_2 - r_2}{w_2}$, where ℓ_2 and r_2 are the left and right arity values of the bottom diagram. The resulting (linear) transformation performed is the composite $Shear_Y \circ Scale_X$ where:

$$Scale_X(x, y) = \left(\frac{w_2}{w_1} \cdot x, y\right)$$

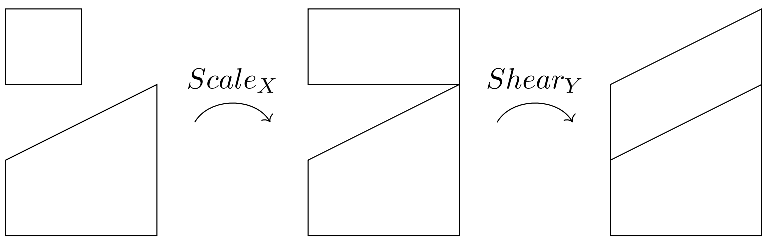


Figure 4: Transformations to tensor two right trapezoids

This scales the x-coordinate by the ratio of the bottom to top widths, fixing y.

$$Shear_Y(x, y) = \left(x, \frac{\ell_2 - r_2}{w_2} \cdot x + y\right)$$

This shears y by an amount proportional to the arity difference over the width, fixing x.

For aesthetic and practical reasons, we have determined it is preferential to scale both trapezoids to the maximum of widths w_1 and w_2 before shearing and tensoring. By scaling to the wider dimension, details inside the first diagram are not unintentionally compressed or overlapped due to cramming into a smaller area.

Specifically, when tensoring diagram 1 and diagram 2, both will first be scaled in the x direction to $\max(w_1, w_2)$. This expands both diagrams to the wider of the two widths.

After this initial scale, diagram 1 undergoes a y-shear by the factor $(\ell_2 - r_2)/\max(w_1, w_2)$, where ℓ_2 and r_2 are the left and right arity of diagram 2.

By first scaling both trapezoids to the larger width, and only then shearing diagram 1 vertically, the tensoring process avoids cramping the contents of diagram 1 during its geometric reshape. This improves the aesthetics and readability of the final diagram.

3.2.2 Composing

When horizontally composing two right trapezoidal diagrams, the right arity (r_1) of the first diagram must equal the left arity (ℓ_2) of the second, by the invariant rules. With their sides touching, this creates proper internal connections between them, with the i^{th} join occurring in the unit-length vertical interval between (w_1, i) and $(w_1, i + 1)$. Specifically, the connections are centred at points $(w_1, 0.5 + i)$.

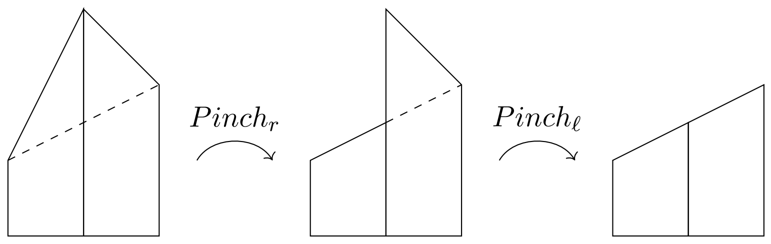


Figure 5: Pinch operations needed to compose two right trapezoids

However, after joining the trapezoids side-by-side, the resulting shape is a pentagon rather than a quadrilateral. To reform this into a trapezoid, we utilise a 'pinching' transformation that moves one top vertex of a trapezoid to a different height. As illustrated in Figure 5, this pinches both the right corner of the left diagram, and the left corner of the right diagram, to create a quadrilateral shape.

For a trapezoid with sides ℓ and r, width w, the top side lies on the line $f_a(x) = \frac{r-\ell}{w} \cdot x + \ell$. To pinch the top-right corner to new height h, the top side has to lie on the line $f_b(x) = \frac{h-\ell}{w} \cdot x + \ell$. Therefore, we define the required (non-linear) transformation as $Pinch_r(x, y) = \left(x, \frac{f_b(x)}{f_a(x)} \cdot y\right)$. A similar function $Pinch_\ell(x, y)$ is defined to pinch a top-left vertex, with $f_b(x) = \frac{r-h}{w} \cdot x + h$.

Let us now calculate what the parameter h should be in our case. When composing diagrams 1 and 2, with top vertices at $V_1 = (0, \ell_1)$ and $V_2 = (w_1 + w_2, r_2)$, diagram 1's top-right and diagram 2's top-left vertices —originally at $(w_1, r_1) = (w_1, \ell_2)$— must become collinear with V_1 and V_2. The equation of said line is $y = \frac{r_2 - \ell_1}{w_1 + w_2} \cdot x + \ell_1$. Therefore, the new vertices, at $x = w_1, x = w_1$, have a height of $y = \frac{r_2 - \ell_1}{w_1 + w_2} \cdot w_1 + \ell_1 = \frac{w_1 r_2 + w_2 \ell_1}{w_1 + w_2}$, the latter being our desired parameter.

3.3 Building the code

The core algorithms for this project reside in two key modules: Read and Draw. The Read module parses the input diagram files and constructs an internal tree representation. The Draw module then takes this tree as input and renders the final diagram by folding over it.

There are two main stages involved in the drawing process. First, `FoldableDiagram` defines the core operations for composing and tensoring diagrams. Second, concrete instances of `FoldableDiagram` implement the rendering of individual leaf elements.

The following subsections provide more details on these modules, as well as their types and classes.

3.3.1 The Read module

The Read module is tasked with parsing external diagram files, validating their structure, and constructing an internal tree representation of the diagrams.

The input files encode diagrams in JSON format. All objects have a `"type"` key, with the remaining attributes depending on the object's type value. There are three primary types of objects that can appear: leaves (or base cases), compositions, and tensors.

Leaf objects represent base case diagrams like morphisms and crossings. There are several subtypes of leaf, depending on the mathematical base case or the amount of information provided for drawing. They each have their own specific attributes:

- Type `"Morphism"`: it has two attributes. `"arity"` is a list of two numbers, the left and right arities of the morphism. `"label"` is a string that represents its name.
- Type `"MorphismWNames"`: a variant on `"Morphism"` where, instead of `"arity"`, we have attributes `"dom"` and `"cod"`. Both are lists of strings whose elements represent the name of each wire's names in their input (domain) and output (codomain), from top to bottom.
- Type `"Crossing"`: it has one attribute, `"permutation"`, which is a list of the numbers from 0 to $k-1$ in any order, and corresponds to the order in which k wires cross each other. Note that the identity is included in this leaf type.
- Type `"CrossingWNames"`: a variant on `"Crossing"` with an additional attribute `"dom"`, a list of strings whose elements represent the name of each wire's names in its input (domain).

Meanwhile, types `"Compose"` and `"Tensor"` both have two attributes: `"diagram1"` and "diagram2", which recursively contain another object of any type.

To parse the JSON input, the Data.Aeson module is imported. The parsed diagram structure is represented with the `Tree` type from Data.Tree. We create a custom datatype `NodeType` for its nodes, with constructors `Compose`, `Tensor`, or `Leaf LeafType`. `LeafType` is another custom datatype with one constructor per leaf variant from the above JSON specification. We also define a function `leafArity` that calculates the arity of each leaf, to be easily used when rendering them.

The key parsing functions are `readInputDiagram` and `readInputDiagramWN`. They take a `FilePath` and return a `Tree` wrapped in `Either` to represent errors occurring during parsing. Both parsing functions validate the object's structure, that the attributes are correct, that its arities match, and verify crossing permutations. The difference between them is that `readInputDiagramWN` also validates matching wire names, not just their quantity, as well as their order when crossing them. Therefore, this second parser only accepts leave types "MorphismWNames" and "CrossingWNames".

To support new diagram leaf types in the future, such as a braid instead of a crossing, the`LeafType` datatype would need a new constructor. The parsing functions would also need to be updated to handle the new type. Additionally, any drawing class (such as `FoldableDiagram`, described in section 3.3.3) would require extensions to render the new leaf type appropriately.

3.3.2 The Draw module: operations

The Draw module defines how an object with type `Tree NodeType` is converted into a `Diagram B` from the Diagrams library. To accomplish this, we first need to define the key operations that modify brick diagrams, as outlined in section 3.2.

In Snippet 3, the `pinch` function returns a deformed diagram where the top-right vertex is at height `h` if `h` $<= 0$, or the top-left vertex is at height $-$`h` if `h` > 0, by applying the $Pinch_r$ and $Pinch_\ell$ transformations discussed before in section 3.2.2.

Let us analyse its type signature, as these will outline the necessary classes that every diagram must inherit. Type `a` must have these constraints:

```
pinch :: (Deformable a a, Enveloped a, AType a) => N a -> a -> a
pinch h d = d # deform (Deformation $ \p -> scaleY (fxb p / fx a p) p)
    where a@(al, ar) = d # arity
          fx (l, r) pt =  pt^._x * (r - l) / (d # width) + l
          fxb = if h < 0 then fx (-h, ar) else fx (al, h)
```

Snippet 3: The Pinch function

- `Deformable a a`, due to `deform`, since it is a nonlinear transformation. As we will see in section 4.1.3, this causes some issues.
- `Enveloped a`, due to `width`.
- The constraints originating from `arity`, which we have named `AType a`.

The `arity` function calculates the length of the vertical sides of the trapezoid. To accomplish this, we make use of the library's `maxRayTraceP p v` function (Manual, 2023, 4.2: Traces), which determines the length from point `p` to the trace of the object in the direction of vector `v`.

We use it at our origin in the Y direction for the left arity, and at its rightmost point for the right arity —by shifting the drawing with `alignR`. This is why we need constraints `Traced a`, `Alignable a` and `HasOrigin a`, as well as placing it in space `V2 Double`. Following the examples in the Manual (2023, 10.2: Poor man's type synonyms), we have defined the type synonym `AType` to summarize them.

Finally, we define the operations for tensoring and composing in Snippet 4. They closely adhere to the mathematical definitions outlined in section 3.2. The parameter of `composeOps` is $(w_1 * r_2 + w_2 * \ell_1)/(w_1 + w_2)$, and the shear factor for `tensorOps` is $(r_2 - \ell_2)/mw$, as previously discussed.

3.3.3 The Draw module: FoldableDiagram

The main fold where the diagram is drawn resides in the function `inputToOutput = foldTree drawNode` (Snippet 5), where `drawNode` invokes three distinct methods depending on whether the node is a leaf, a composition, or a tensor. These methods are defined in the central class of this module, `FoldableDiagram`.

```haskell
tensorOps :: (Deformable a a, Enveloped a, AType a, Transformable a)
    => a -> a -> (a -> a, a -> a)
tensorOps d1 d2 = (shearY sh . scaleX (mw/w1), scaleX (mw/w2))
    where [w1, w2, mw] = [width d1, width d2, max w1 w2]
          sh = (d2 # arity # snd - d2 # arity # fst)/mw

composeOps :: (Deformable a a, Enveloped a, AType a)
    => a -> a -> (a -> a, a -> a)
composeOps d1 d2 = (pinch middle, pinch (-middle))
    where [w1, w2] = [width d1, width d2]
          [h1, h2] = [d1 # arity # fst, d2 # arity # snd]
          middle = (w1*h2 + w2*h1)/(w1 + w2)
```

Snippet 4: Tensoring and composing operations

```haskell
inputToOutput :: FoldableDiagram a => Tree NodeType -> a
inputToOutput = foldTree drawNode
    where drawNode (Leaf l) _ = leaf l
          drawNode Compose [d1,d2] = compose d1 d2
          drawNode Tensor [d1,d2] = tensor d1 d2
```

Snippet 5: The fold: inputToOutput

`FoldableDiagram` is displayed in Snippet 6. This code warrants an in-depth analysis, as it is the most significant one in terms of its theoretical value. It inherits multiple classes: the same previously discussed for `pinch`; `Transformable a` for linear transformations; and `(Juxtaposable a, Semigroup a)` for operators `|||` and `===`, which position any two diagrams horizontally or vertically.

Its methods `compose` and `tensor` invoke `composeOps` and `tensorOps`, although they can be overriden if needed.

The `leaf` method draws leaf nodes on the input tree. It has to be defined for each new instance, as this is precisely what differentiates them in how they are rendered at the base cases.

The other method to be implemented by each instance is `strokeOutput`, inspired by the library's stroke methods (Manual, 2023, 3.5: Paths), which specifies how to convert any `FoldableDiagram a` into a renderable `Diagram B`, with `B` being the default SVG backend.

```haskell
class (Deformable a a, Enveloped a, AType a, Transformable a,
    Juxtaposable a, Semigroup a) => FoldableDiagram a where
    compose :: a -> a -> a
    compose d1 d2 = d1 # t1 ||| d2 # t2
        where (t1, t2) = composeOps d1 d2

    tensor :: a -> a -> a
    tensor d1 d2 = alignB $ d1 # t1 === d2 # t2
        where (t1, t2) = tensorOps d1 d2

    leaf :: LeafType -> a
    strokeOutput :: a -> Diagram B
```

Snippet 6: FoldableDiagram

Finally, there remain a few additional utility functions that can be used to draw or implement instances of `FoldableDiagram`, if desired. The following paragraphs provide a brief overview of these functions.

The function `flatCubic` draws a cubic `FixedSegment` in which the two control points share the same y coordinate as the two endpoints. This ensures that the tangent tends to being horizontal when approaching the endpoints, which is useful for deforming paths in a smoother way.

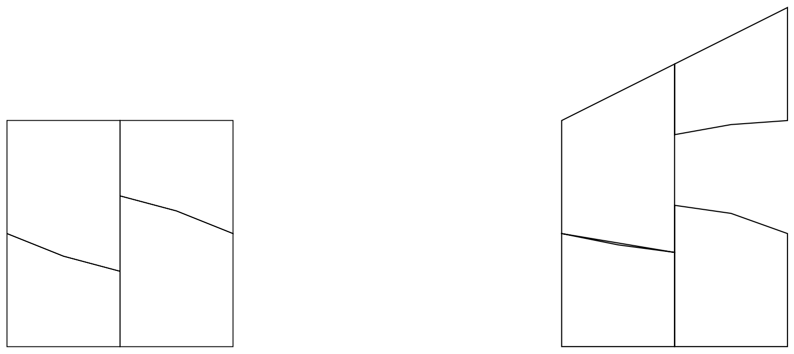
The function `connectionPoints` identifies the points of each connection interval along the sides of a base region —that is, the points at which wires can connect.

The `drawWires` and `drawCrossingWires` functions draw paths consisting of the wires required for a morphism and a crossing, respectively.

Finally, the functions `rectangify`, `squarify`, and `isoscelify` modify a completed diagram such that it forms a rectangle, square, or isosceles trapezoid rather than a right trapezoid.

4 Analysis

Now that the base code is complete, we shall examine how to implement its available classes and methods to customise the diagram's appearance and capabilities. During



(a) Rendering the brick diagram from 3b (b) Using === instead of <>

Figure 6: Instantiating Path V2 Double as a brick diagram

this analysis, we will identify any issues that have surfaced as a result of the library's characteristics and our choice of algorithm.

The most relevant snippets of code for this section are included in appendices A and B for this project's Haskell, and appendix C for DisCoPy's Python.

4.1 Drawing diagrams

4.1.1 Path V2 Double as FoldableDiagram

The simplest approach to implement an instance of `FoldableDiagram` is to draw a brick diagram as a `Path V2 Double`, as shown in Snippet 7 and Figure 6a. The `strokeOutput` function is precisely the library's `strokePath` function, while each leaf is a right trapezoid produced by pinching a square.

However, a problem arises when using === for tensoring. Although `|||` operates as expected, we can see in Figure 6b that === leaves a vertical gap. The key insight is that `juxtapose`, the method underlying this command, uses a default `juxtaposeDefault` that works with the envelope instead of the trace (Manual, 2023, 10.2: Classes for transforming and combining). The problem occurs because the path's envelope does not yield its boundary, as commented in section 2.3. For classes with an undesirable juxtaposition, the solution is to apply `snug` and basic concatenation with <> (Manual, 2023, 3.4: Alignment).

A second approach to instantiate `Path V2 Double` as a `FoldableDiagram` involves drawing only the string diagram's wires, with the morphism boxes remaining implicit at each region's centre. However, directly using `drawWires` for this purpose introduces another issue: the arity and width are not calculated correctly anymore, since they rely on the brick diagram, which is no longer included.

It could be overcome by keeping track of the diagram's dimensions. It could also be achieved by modifying the object's trace and envelope, although such operations only apply to `QDiagram` at the moment (Manual, 2023, 4.1: Envelope-related functions). Our solution consists in introducing a new wrapping type, `BrickWrapper`, and a new class, `Drawable`.

4.1.2 The BrickWrapper module

The class `Drawable a` is used for any supplementary elements that need to be included but are not part of the brick diagram itself. Its only methods, `draw` and `strokeDrawing`, are analogous to `leaf` and `strokeOutput`. The advantage of this class is that it only inherits `Deformable a a`, `Semigroup a`, and the vector space, and not the other classes needed for `FoldableDiagram`. Consequently, this class enables additional elements to be composed and deformed, but without being incorporated into the dimensions of the overall diagram.

The datatype `BrickWrapper a` handles drawing the brick diagram behind the scenes and uses it to perform necessary calculations, while simultaneously carrying any other `Drawable` elements in parallel: `data BrickWrapper a = BW {_wrapper :: Path V2 Double, _user :: a}`. These added elements can be extracted with `unwrap`, and the brick diagram itself can be accessed through `strokeBrick`.

As long as `a` is an instance of `Drawable`, the `BrickWrapper a` type is automatically an instance of every class required for `FoldableDiagram a`, as demonstrated in Snippet 8. The key advantage is that all methods for these class instances can be customized. Below is a high-level overview of them:

- `InSpace V2 Double`: Enables locating the diagram in 2D space.

- `Semigroup a => Semigroup (BrickWrapper a)`, and `Deformable a a => Deformable (BrickWrapper a) r`: Necessary for `a` to compose with itself and deform during the recursion.
- `Transformable` and `HasOrigin`: Relate to the previously defined `deform` method.
- `Alignable`, `Traced` and `Enveloped`: Only apply to the brick wrapper itself, ignoring other user elements.
- `Juxtaposable`: `juxtapose` is redefined by trace, enabling === to work properly and resolving both issues simultaneously.

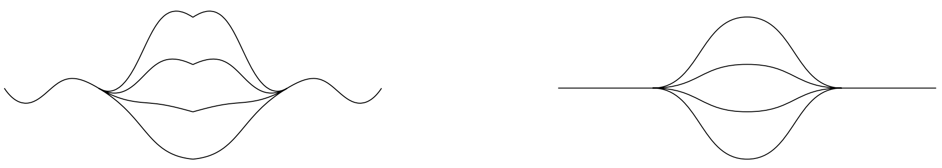
By making `Path V2 Double` an instance of `Drawable` and defining `WiresDiagram` as its wrapper type, `WiresDiagram` automatically satisfies `FoldableDiagram`, as shown in Snippet 9 and Figure 7a. This approach represents the primary pattern for constructing new diagram types.

Wrapping the brick diagram path and keeping it separate from other drawable elements, the `BrickWrapper` approach provides a flexible way to include extra visual components without disrupting the diagram's core structure and dimensions. The `Drawable` class streamlines this by defining a common interface for the additional elements. Together, they enable augmenting the diagrams with wires, annotations, or other features as needed, while still relying on the brick diagram for key calculations and properties. This abstraction makes building new diagram variants straightforward and modular.

4.1.3 A custom FoldableDiagram

If we want to make `Diagram B` an instance of `FoldableDiagram` a third issue arises: diagrams cannot be non-linearly deformed (Manual, 2023, 3.4: Deformations). This explains the initial choice to build the diagrams from paths, and, in fact the library itself recommends a similar approach:

> "Most functions that create some sort of shape (e.g. `square`, `pentagon`, `polygon`...) can in fact create any instance of the `TrailLike` class. You can often take advantage of this to do some custom processing of shapes



(a) Rendering a `WiresDiagram` (b) Using a smoother custom type

Figure 7: The difference between instantiating directly, or creating a new type

> by creating a *trail* instead of a diagram, doing some processing, and then turning the trail into a diagram.” (Manual, 2023, 5.4: Naming vertices)

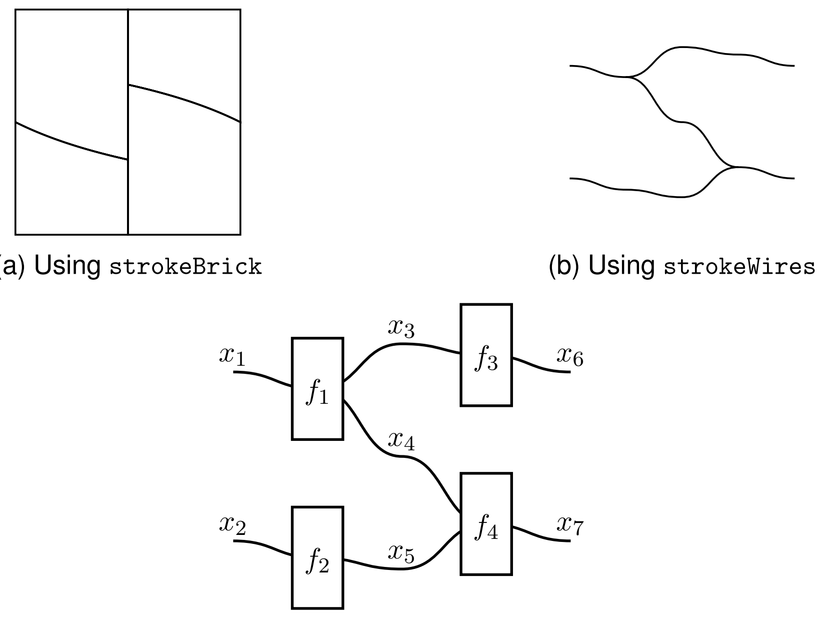
While paths enable folding and composing diagrams, their one-dimensional nature restricts styling options, preventing the addition of further detail or customisation beyond pure lines in the final rendering. The library documentation acknowledges this trade-off and recommends processing trails before converting them into diagrams.

Nonetheless, some use cases demand richer representations than paths allow. For instance, there is no way to include text labels, and connections between segments are not smooth. Note the difference between Figure 7a, using the basic path instance, and the use of a more complex datatype implementation at 7b. Overcoming this obstacle requires an alternative approach that preserves deformability while permitting customisation.

Our solution involves creating a custom type that is `Drawable`, that defines its `deform` method as needed, and that contains additional information to be rendered after recursion —then wrapping it with `BrickWrapper` later on.

The user can implement such a type, but we provide an example custom one: `type LabelsDiagram = BrickWrapper UserDiagram`, where `data UserDiagram = UD {_sd :: Path V2 Double, _ls :: [Located (Diagram B)]}`.

The only instances to make `UserDiagram` of are `V2`, `Double`, `Semigroup` —defined in the straigthforward way—, and a custom `Deformable` instance that forces all cubic splines to have horizontal control points, and deforms `Located` objects by just a translation. Finally, we define the `Drawable` instance to render as desired: drawing wires, crossings, placing text labels and boxes, etc. A `strokeWires` function provides access to just the string diagram’s wire drawings.



(a) Using `strokeBrick` (b) Using `strokeWires`

(c) Using `strokeOutput`

Figure 8: Drawing LabelsDiagram in three different ways

Figures 8a, 8b and 8c demonstrate the three rendering approaches for a `LabelsDiagram`. This example implements the string diagram from Hedges and Herold (2019). Comparing the fully drawn version in 8c to their original in 3a reveals strong similarities in appearance, with the key difference that this one has been produced in an exclusively recursive manner, without knowing the general placement of any element beforehand.

By wrapping a custom deformable type with extra data, this method allows detailed diagram rendering while preserving the pure recursive nature of the fold. The user can fine-tune behaviours, such as deformations, as needed through the custom instances. While some upfront design is required, the approach prevents their structure being locked in by design, and enables recursively constructing diagrams with any custom rendering.

4.2 Compiling diagrams

As discussed in section 2.2, various existing tools can already render string diagrams. However, none construct diagrams entirely recursively by following along their structure. Our approach has an advantage: it enables evaluating semantics directly alongside the diagram's generation. While other tools may be able to associate semantics with diagrams and draw them separately, this project allows executing both processes in parallel. Moreover, implementing custom semantics is straightforward and accessible to users.

4.2.1 The MonCatWrapper module

The previously defined `BrickWrapper` allows us to add `Drawable` elements to a brick diagram. These `Drawable` elements can join together using the semigroup operation, deform together with the wrapper, and later be stroked as a `Diagram B`.

However, we would also like to define additional semantic elements —ones that must be composed differently depending on whether it is a tensor or a composition. To accomplish this, we introduce the class `Compilable` and the datatype `MonCatWrapper`.

The class `Compilable b` represents the semantics of the diagram. Its method `baseCase` is analogous to the `strokeOutput` method, while its `tensorOp` and `composeOp` methods are intended to be used in `tensor` and `compose` when instantiating `FoldableDiagram`.

The datatype `MonCatWrapper a b` combines the capabilities of the `BrickWrapper` module —adding `Drawable a` elements— with an extra attribute for `Compilable b` elements: `data MonCatWrapper a b = MD { _wrapper :: Path V2 Double, _drawing :: a, _semantics :: b }`.

In implementing the various instances for `MonCatWrapper`, we closely follow the approach used for `BrickWrapper`, using the `Drawable a` component and leaving `Compilable b` until the end. The latter is used later when instantiating `FoldableDiagram`, as mentioned previously. This instance is displayed in Snippet 10.

Lastly, we may obtain `MonCatWrapper`'s components from the outside with functions `getSemantics`, `getDrawing`, and `getWrapper`.

In order to recover the functionality of `BrickWrapper` as a subset of `MonCatWrapper`, we define the empty tuple `()` as an instance of `Drawable` and `Compilable`. With this, one can create a type for just the brick diagram (`MonCatWrapper () ()`), with only `Drawable` elements (`MonCatWrapper a ()`) or only `Compilable` (`MonCatWrapper () b`).

The `MonCatWrapper` datatype therefore combines the flexible diagram augmentation of `BrickWrapper` with added semantics through the `Compilable` class. By keeping the brick's path separate and using `Drawable` for visual components, `MonCatWrapper` inherits `BrickWrapper`'s modularity. Meanwhile, `Compilable` enables diagram elements to capture semantic meanings during composition and tensoring. Together, these features allow supplementing diagrams with extra visual and semantic information as needed, without disrupting the core diagram structure and calculations. For this reason, the `MonCatWrapper` type makes it straightforward to build new diagram variants as desired.

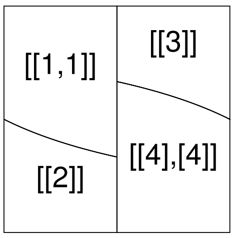
4.2.2 An example with Matrix Double

In section 2.1.2 we demonstrated how matrices are an example of a monoidal category. We can now readily implement them with the `MonCatWrapper` datatype.

First, we make the `Matrix Double` type —from the Data.Matrix module— an instance of `Compilable`, as shown in Snippet 11. To accomplish this, we need a method to parse a matrix from its leaf's labelling text for morphisms (via the `stringToMatrix` function) and another method to convert a permutation array into a permutation matrix for crossings (via the `permToMatrix` function). For the `baseCase` method of `Compilable`, the base cases for `Morphism` and `MorphismWNames` use the former function, while the base cases for `Crossing` and `CrossingWNames` use the latter.

The `composeOp` method is simply matrix multiplication `(*)`, while `tensorOp m1 m2` uses the function `joinBlocks (m1,z1,z2,m2)` to combine the input matrices `m1` and `m2` as diagonal blocks of the resulting matrix, with zero-filled matrices `z1` and `z2` padding the off-diagonal. This corresponds to the direct sum of both matrices.

By making `Matrix Double` an instance of Compilable in this way, we enable `MonCatWrapper` to incorporate matrix semantics into the diagram structure. However, we also want the brick diagram to display the morphism labels at the center of each



(a) Rendering a `MonCatDiagram`

$$\begin{bmatrix} 3.0 & 4.0 \\ 0.0 & 8.0 \end{bmatrix}$$

(b) Its compiled `Matrix Double`

Figure 9: Compiling a Matrix Double from its MonCatDiagram

region. For this, we must create a new type, `Labels`, that can print these texts and is an instance of `Drawable`. We include this code in Snippet 12 so that anyone may find it and use it as is, and as a way of showing one example of a complex custom `Drawable` instance, too.

With all of these components already defined, we can create the type `MatrixDiagram` as the wrapper of both previous types: `type MatrixDiagram = MonCatWrapper Labels (Matrix Double)`. This gives us a `FoldableDiagram` instance with both labels and matrix semantics.

Figure 9a shows an example of a brick diagram rendered as a `MatrixDiagram`, concatenating the brick wrapper and the `Labels` component. Meanwhile, Figure 9b displays the semantic `Matrix Double` meaning of this same diagram, which has been calculated automatically.

Let us calculate that this is correct in the following equation, where $\cdot$ is the ||| operator for composing, and $\oplus$ is the === operator for tensoring:

$$\begin{aligned} M &= \left(\begin{pmatrix} 1 & 1 \end{pmatrix} \oplus \begin{pmatrix} 2 \end{pmatrix}\right) \cdot \left(\begin{pmatrix} 3 \end{pmatrix} \oplus \begin{pmatrix} 4 \\ 4 \end{pmatrix}\right) \\ &= \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 \\ 0 & 4 \\ 0 & 4 \end{pmatrix} \\ &= \begin{pmatrix} 3 & 4 \\ 0 & 8 \end{pmatrix} \end{aligned}$$

In this way, `MatrixDiagram` demonstrates the power of `MonCatWrapper` to integrate brick diagrams, string diagrams —via `Drawable`— and any categorical semantics —through `Compilable`. By wrapping the brick path with labels and a matrix, we augment the basic brick algorithm with customisable elements, and semantics that can be tailored to different domains or applications.

4.3 Testing the code

The most difficult part of working with the Diagrams library is the enormously complex type system behind its functionality. This might make it challenging to find the exact class that a diagram must be an instance of, but it also means that, once implemented, the code is less prone to unintended mistakes. As the popular functional programming saying goes: "Once your code compiles it usually works" (HaskellWiki, 2023). Therefore, in order to review the correctness of this project's algorithms, we must rely on visually inspecting a wide range of test cases.

4.3.1 The test suite

The first JSON files were created manually until the basic commands showed a reasonable output. However, later in the project, a random set of 100 files were generated with a Python script as a full test suite to cover a larger scope of possibilities. They fulfil the JSON specifications, and they have a chance to display any input element in any configuration. The files are correctly parsed with `readInputDiagram`, but not `readInputDiagramWN` —so the name of the wires are not matched for equality, as discussed in section 3.3.1.

A test file in the project then draws each of these files in all possible ways. We have by now defined several different instances of `FoldableDiagram`. We will not account for `MonCatWrapper`, as this behaves similarly to `BrickWrapped`, but it is more difficult to generate and check for correctness of appropiate semantics. We will consider the first three instances for testing:

- The direct instance on `Path V2 Double`. It can be found in the module `StringDiagrams.Draw.NaiveDiagram`.

- `WiresDiagram`, the `Drawable` instance of `Path V2 Double` which has been wrapped with a `BrickWrapped`. It can be found in the module `StringDiagrams.Draw.WiresDiagram`.
- `LabelsDiagram`, the custom instance. It can be found in the module `StringDiagrams.Draw.LabelsDiagram`.

These Haskell modules are not part of this project's public methods, being only instantiating examples to be referenced. In part, this is because all instances in Haskell are implicitly imported and this would modify the Diagrams' original types. They are instead available as a piece of documentation.

There are six possible options to convert the instances into a renderable diagram:

- `(input # inputToOutput :: Path V2 Double) # strokeOutput`
- `(input # inputToOutput :: WiresDiagram) # strokeOutput`
- `(input # inputToOutput :: WiresDiagram) # strokeBrick`
- `(input # inputToOutput :: LabelsDiagram) # strokeOutput`
- `(input # inputToOutput :: LabelsDiagram) # strokeWires`
- `(input # inputToOutput :: LabelsDiagram) # strokeBrick`

The direct instance can only be rendered with `strokeOutput`. `WiresDiagram` can be used with `strokeOutput` to draw the wires, or `strokeBrick` to draw the brick diagram. `LabelsDiagram` can be used with `strokeOutput`, `strokeWires`, and `strokeBrick`, although the last one is the same one as for `WiresDiagram`, so we omit it. Instead, we have included the drawing of `LabelsDiagram`'s string diagram and brick diagram together. An example of one of these randomly generated diagrams is displayed in Figure 10.

4.3.2 Imperfections found

The mistakes found during these tests come in two distinct flavours: logic-based errors —which come from a faulty implementation of the algorithm, and have already

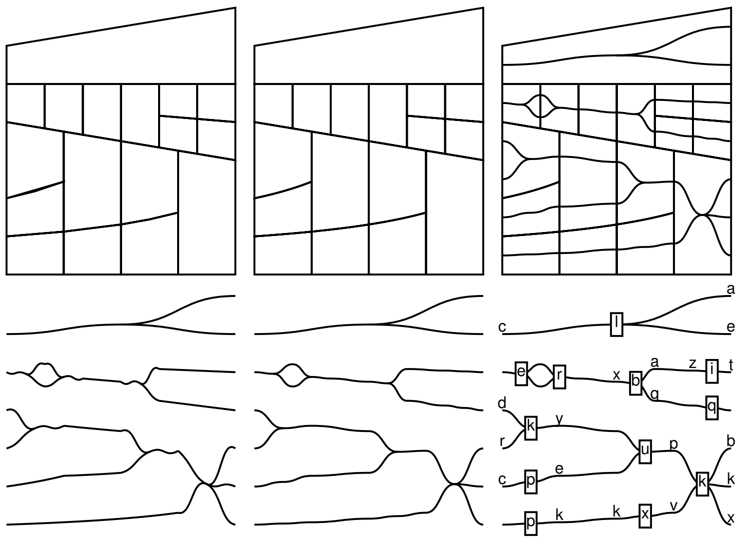


Figure 10: Randomly generated test example #21

been corrected— and visual imperfections, which come from a characteristic of the algorithm or a feature from the original library which is not easily avoidable.

One of the first errors discovered by the test suite has already been commented on and fixed in section 4.1.1: the distinction between juxtaposing by trace or by envelope causes === to malfunction. Other logic-based errors include not accounting for the width of a diagram, or miscalculating the centrepoint of a region.

There are also features of the algorithm that, although less than optimal, are fully intended and usually fixable. The main one is that the end shape of the diagram is a right trapezoid, an unsymmetrical shape that may not be wanted. This is why we provide the user with functions `rectangify`, `squarify`, and `isoscelify`, to turn it into a rectangle, a square, or an isosceles trapezoid. Also, sometimes the diagram ends up being too long or too wide for its contents; it may also have its boxes too small. Thus, after `inputToOutput` but before `strokeOutput`, the diagram can be deformed or scaled as needed. After `strokeOutput` one can also change its style characteristics, such as the line width or colour (Manual, 2023, 3.4: Texture).

Another intrinsic feature of the algorithm is that the brick diagrams are not necessarily made up of straight lines, due to the non-linearity of `pinch`. While the external shape is always a quadrilateral, its internal contents might include segments of varying curvature. This is expected and unavoidable. And, precisely because of this, the resulting string diagram inside the brick diagram is typically well-packed and harmonious.

Next, there are implementation technicalities that are not considerably noticeable, but their removal would benefit the overall appearance of the drawings:

- The font size of the labels is consistent across the diagram. Even more, when uploading an .svg image on LaTeX, the document's font format is applied onto them. This means that some labels will overflow off their boxes, or will overlap onto some wires if the text is too long. However, it also means that we can use $ for mathematical notation and it will be correctly rendered on the LaTeX page.
- There is no suitable way of removing the paths contained inside the boxes. There is a way of clipping `Diagram` types, and a user contributor's package on Boolean operations on paths (Manual, 2023, 3.5: Clipping; Boolean operations on paths). Nevertheless, automatically removing these paths fragments would mean that, when scaling the finished product horizontally, the wires and the boxes would separate and some space would appear between them. Our solution is to use a white background for the boxes, which unfortunately cannot be overridden for the moment.
- For the same reason, we cannot attach the segments' endpoints to fixed locations on the sides of the box. Doing so would also produce spaces between the wires and the boxes. This is why we have fixed them to the centrepoint of the region.
- When adding the wires' names on a leaf, we draw both its input and output labels. However, when composed, one's input and another's output labels coincide. This cannot be predicted in advance due to the recursive compartmentalisation of the algorithm. We solve this by filtering at the end —using `nubBy`— the located objects that share a common location.

Compartmentalising the recursion, which is a basic feature of this algorithm, can also bring harder challenges to overcome. Occasionally, not being able to look at the general appearance of the drawing beforehand can mean that the end shape looks in-

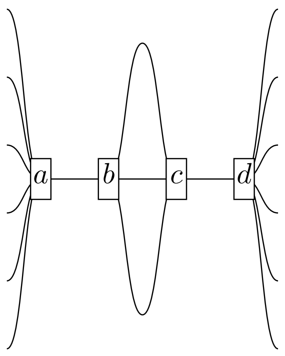


Figure 11: Inconsistently looking diagram

consistent at the end. One prominent case is when a tall diagram gets composed horizontally with a shorter one. This causes the shorter shape to look more voluminous than what would be necessary, as illustrated in Figure 11.

While it could be scaled in the x direction to smoothen the curves, this could potentially provoke excessive space in other parts of the diagram to its sides, instead. This problem is not easy to adjust, and not without modifying how much space the string diagrams occupy during their deformation, which should be globally calculated in any case.

Lastly, executing the test suite with `cabal test` takes approximately 12.2 seconds in the local system, which is 0.122 seconds per example to process on average. This slow performance is mainly due to the small parameter used in the `deform'` method in `BrickWrapper` (0.0001), which allows for the curves in the brick diagram to be very smooth. Removing it causes the curves to have bumps, especially for smaller diagrams. But it also brings down the time taken to process all 100 examples to around 5.3 seconds, less than half of the previous time.

4.4 Comparison to DisCoPy

DisCoPy (De Felice et al., 2021) is a Python library for rendering string diagrams that has similar capabilities to our software. However, there are some notable differences in DisCoPy's implementation and functionality, primarily due to the data structure used to store the diagrams.

4.4.1 Implementation: whiskering

There are multiple approaches to storing string diagrams, arising from different conceptual viewpoints. One common technique is to treat diagrams *geometrically*, as graphs of points and line segments on the plane. While straightforward, this graph-based approach loses mathematical meaning, such as the internal relationships that the subdiagrams encode.

In contrast, our algorithm stems from a purely *algebraic* definition: string diagrams are morphisms in a free monoidal category. The advantage to using the Diagrams library is that it fully handles the transition between this algebraic representation and the required geometrical one for visualisation. This allows us to preserve the diagram's mathematical meaning onto them at no cost.

DisCoPy takes a more *combinatorial* approach. They represent a diagram via its domain, its codomain, its collection of boxes, and each box's offset (De Felice et al., 2021). This representation is known as whiskering. The domain captures the diagram's inputs —equivalent to the list of wire names on the left side— while the codomain describes the outputs.

Every diagram can be sliced vertically into layers, with each layer containing a box, some wires above, and some below. The offset tracks what wires are above each box in this format. The input, the layers, and the output provide sufficient information to represent any diagram. Moreover, the addition of extra elements, such as braids or crossings, behaves exactly the same as a box does.

This layered representation is always possible, and is equivalent to the geometric view if one considers sliding the boxes back and forth between each other —that is, taking quotient modulo interchangers. Not taking it results in a representation of a premonoidal category instead.

According to Toumi (2020), one of the authors of this library, the drawing algorithm proved to be the part that took the most amount of work. This occurred because converting from the combinatorial description to the graphical format involves numerous layout decisions. In contrast, our algorithm depends entirely on the inherent tree structure to position diagrams, largely bypassing the need for the manual choices. With our data structure, the underlying mathematics naturally defines the layout, rather than relying on designing it discretionarily.

Moreover, we believe that the Haskell language is well-suited for encapsulating purely mathematical algorithms such as ours. While DisCoPy has chosen Python for its approachability —more now amid the rise of machine learning applications— Haskell's advanced type system and common use in applied category theory make it more adept at abstractly modelling complex systems.

The developers of DisCoPy have very recently expanded the capabilities of the DisCoPy library with their latest release, which incorporates a novel data structure for hypergraphs (Toumi et al., 2023). The library is expected to see ongoing growth, and expansion of its features, in the near future.

4.4.2 Drawing diagrams with DisCoPy

In DisCoPy, diagrams can get composed with one another with the method `then`, or the operator >>. If in a monoidal category, they can also get tensored with the method `tensor`, or the operator `@`. Morphisms are drawn as boxes with the class `Box`.

In Snippet 13 we see how to read a diagram from our JSON file format and draw it with DisCoPy. Its output image is displayed in Figure 12. Note that the contents are drawn top to bottom instead of left to right.

If the JSON object's type is `Compose` or `Tensor`, it only needs to call for the children objects recursively and join them with operators >> or `@`. For the base case of a morphism, we create a `Box` from its `"label"`, its wire inputs and its wire outputs. The wires have to be calculated from the `"dom"` and `"cod"` attributes, which are translated into the class `Ty` and tensored with each other.

We have chosen to process the same diagram than in 3a and 8c. These three figures are an example of each of the ways to render a string diagram: geometrically (with Tikz), algebraically (with our library), and combinatorially (with DisCoPy). As already mentioned, the easiest internal implementation is the algebraic one. However,

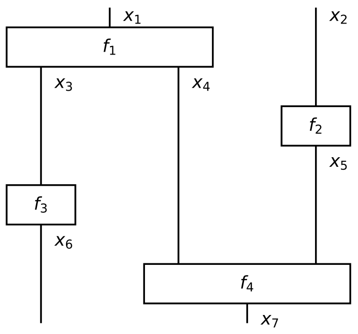


Figure 12: An example of a string diagram rendered by DisCoPy

DisCoPy has a very straightforward interface for the user, which our tool could take inspiration from.

On the other hand, we argue that our rendering is more aesthetically pleasing and resembles hand-drawn diagrams better. This might be noticed in our more complex example at Figure 10, which was randomly generated, but still harmonious. In our diagrams, the drawings are packed as compactly as possible based on the user's given configuration. As it has been mentioned before, this is because we deform the internal regions non-linearly and globally at each rendering step. Additionally, our algorithm renders tensored diagrams at the same x-coordinate, making it immediately clear which processes can be executed concurrently. DisCoPy does not implement its algorithm in these ways, so it provides less visual aid for researchers compared to our approach.

If the category is symmetric, and thus can contain crossings, the user can also construct its base case with the `Swap` class, imported from their `symmetric` module. In fact, in the latest release of the library (Toumi et al., 2023), the authors have included a module for each of the category extensions present in the survey from Selinger (2010), which we discussed in section 2.1.1.

4.4.3 Compiling diagrams with DisCoPy

DisCoPy allows for the compilation of string diagrams by applying a functor onto them —a functor can translate boxes and wires into semantic meanings. The diagram can then be evaluated by looping over its list of layers.

However, DisCoPy already implements some pre-built semantic modules for users to directly employ. It contains the category of Python functions, with tuples as tensors. Additionally, it implements the category of matrices with the Kronecker product. And, similarly to our work, it also incorporates the category of matrices with the direct sum.

We can reuse the previous code to parse the JSON file into a diagram. Instead of using a Box as the base case, we can directly read the matrix contained in its label. The `Matrix` class requires the flattened matrix and its dimensions as parameters.

The code for this is shown in Snippet 14. Its output for our semantic example is exactly the same as ours, which we calculated in Figure 9b: `Matrix[int64]([3, 4, 0, 8], dom=2, cod=2)`.

The primary distinction from our library is that DisCoPy requires an additional step of converting the categorical object into either the drawing structure or the semantic structure. Since it implements drawing and compiling as entirely separate methods, it is not immediate to infer that drawing is merely another form of compilation.

4.4.4 Other capabilities in DisCoPy

Our tool operates under the assumption that the string diagram in the JSON input file is properly configured. The Read module can validate some of these assumptions; however, we do not presume that the diagram belongs to any specific category. Additionally, there is no implementation for rewriting diagrams dependent on said categories.

Therefore, we focus our tool on rendering and compiling diagrams, which is the innovation we present. Rewriting diagrams can be better handled by separate, specialised tools, as noted in the literature review in section 2.2.

DisCoPy also represents one such tool. It can check for equality between two diagrams, and compute the optimal ‘normal form’ diagram for a given process. In fact, we could use DisCoPy to separately rewrite diagrams before processing them. Though, at some point, integrating native rewriting capabilities into our tool could prove useful.

DisCoPy also contains modules for working with more specific domains. For instance, one can use diagrams to study grammars in linguistics. Quantum processes can be depicted diagrammatically as well. In fact, the library was first developed for NLP methods on quantum hardware, so they are able to produce quantum circuits with it that can link to an external, domain-specific compiler: t|ket⟩ (Sivarajah et al., 2021).

This is not the only way in which it can interface with external tools. More recently, it has included integration with PyTorch, JAX, and tensornetwork for machine learning applications (Toumi et al., 2023).

While DisCoPy is a more general tool than what we present in this project, our novel approach of storing string diagrams can bring several improvements. First and foremost, in the visual rendering of diagrams and its packing consistency. It also radically simplifies the drawing algorithm, due to the declarative expressiveness of the Diagrams library, which Python lacks. The use of a tree structure may even improve time complexity of some algorithms, due to the bypass of a flat list of all boxes, and potentially leveraging the explicit parallelism in its construction.

5 Conclusions and recommendations

5.1 The StringDiagrams library

5.1.1 Publishing the package

The software tool developed for this project is hosted on GitHub at `https://github.com/celrm/stringdiagrams`, and it has the format of a Cabal package. This library is intended for future publication on Hackage, the central repository for open source Haskell packages.

Apart from releasing the library as a standalone package, another option would be to integrate the code into one of the existing Diagrams library packages. Specifically, it could be included in either the core `diagrams-lib` package —which contains Diagrams.TwoD modules related to two-dimensional diagrams— or the `diagrams-contrib` package, for user contributions that are too specialised for the main library. The advantage of publishing it as part of an existing package is that it would benefit from the established user base and visibility of those libraries. On the other hand, a standalone package allows for more control over releases and dependencies.

For integration with these two libraries, the code must be released under a BSD3 license. Therefore, our code is also released under this license. It allows reuse, modification, and distribution, even for proprietary software. However, redistributions of the software require maintaining the license attached to it, and the contributors' names cannot be used to promote derived products.

Another requirement for publishing the library is matching the documentation format standards, which should be similar to the rest of libraries. Specifically, documentation should be generated with the tool Haddock, which produces HTML files from Haskell source files annotated with comments. The Diagrams library includes a `diagrams-haddock` package that enables documentation to contain self-rendered images. For our library, including illustrative diagram images in the documentation would be essential.

5.1.2 Structure

The StringDiagrams Cabal project currently contains a library, an executable, and a test suite. The library exposes a StringDiagram module, exporting key functions, classes, and types from the core code (explained in section 3.3), as well as the wrappers described in sections 4.1 and 4.2. It also exposes separate modules for different instantiations of `FoldableDiagram` and its derived types: `NaiveDiagram`, `WiresDiagram`, `LabelsDiagram`, and `MatrixDiagram`, all in the `Draw` folder. These last four modules provide examples rather than usable library code, but users can still access these custom renderers.

Internal, non-exposed library modules include `Read`, `Draw`, `BrickWrapper`, and `MonCatWrapper`. As mentioned in section 4.4, mimicking DisCoPy's clear interface, which allows creating diagrams from any category, could be worthwhile. However, the current framework sufficiently demonstrates the innovation on the data structure and drawing algorithm.

In addition to the library, the Cabal project contains an executable that generates most of the images included in this text. This provides useful sample code for future users, who can use these visual examples as a foundation to start creating their own string diagrams.

The test suite forms the third and final component of the Cabal project, thoroughly detailed in section 4.3. The folder contains a Jupyter notebook with Python code that generates arbitrary JSON files, following the specification required for the library's Read module to parse. While the generation is not completely uniform, all possible case combinations have a chance of being produced. The test folder also includes the 100 test cases generated by the notebook. Additionally, a Haskell code file renders each test case in the 6 different configurations discussed in the testing analysis. By utilising a variety of rendering configurations, the test suite provides comprehensive coverage to validate the library's capabilities for diverse inputs.

5.2 Future recommendations

5.2.1 Implementation

This project showcases that a recursive data structure, where each node does not know about the rest of the tree —not even knowing its own child nodes—, can directly render diagrams thanks to the declarative nature of the Diagrams library and its operators === and |||. The key innovation to implementing it has been to maintain a right trapezoidal shape invariant on the brick diagram. Given this trapezoid shape constraint and the power of the two Diagrams operators, there are likely many more implementation possibilities than what is presented here.

The brick diagram wrapper shown in section 4.1.1 calculates dimensions and juxtaposes correctly because its Envelope holds the trapezoidal shape. An alternative approach could be to enforce this invariant shape on any diagram's Envelope, eliminating the need for a wrapping type. However, as mentioned in section 4.1, Envelope operations currently only apply to the type `QDiagram`, which lacks nonlinear deformability.

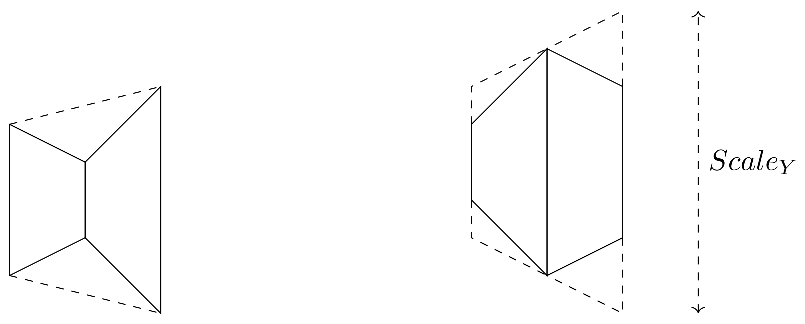
One potential way to avoid nonlinear deformations is to change how two right trapezoids are composed horizontally. In section 3.2.2 we introduce the Pinch deformation, which enables the middle vertices of the composed trapezoids to become collinear with the other top vertices. Rather than deforming the diagrams directly, a possibility is to fill their gaps with empty space.

If the middle vertices are lower than their expected place, the resulting pentagon simply needs an empty triangle placed at the top. If the middle vertices are higher, two empty triangles on the sides are necessary. This would enlarge the overall diagram, so a uniform Y-axis scaling would also be required. The empty space could be generated with the Diagrams command `phantom`, which takes up the space specified in its parameter, but visually produces no drawing.

A more homogeneously looking approach to incorporate empty space would be to make the diagram symmetric along its horizontal axis, transforming it into an isosceles trapezoid shape, and whereby space is added evenly to both the top and bottom portions. This operation, along with the conversion back into a right trapezoid, only requires shear transformations, which are linear. This symmetrical padding approach is illustrated in Figure 13, although the last step to convert the figure back into a right trapezoid is not reflected.

While avoiding the `Deformable` constraint in diagrams simplifies the code, this approach also sacrifices visual elegance. A trade-off emerges between code simplicity and aesthetic appearance. The full implications of this modification remain undetermined, as, for example, it may fix the visual inconsistencies noticed during testing, or introduce new unforeseen issues. For example, scaling down the diagrams could cram and overlap elements. Incorporating empty space padding rather than deforming diagrams would represent a significant change that requires further investigation to weigh its costs and benefits.

Another way to improve this library's implementation would be to use the more advanced `Arrow` type from the Diagrams library. Trying to draw arrows in a fully compositional manner is not possible, because some attributes need to be calculated from the global perspective of the entire diagram. For instance, arrowheads cannot be scaled



(a) The top middle vertices are lower (b) The top middle vertices are higher

Figure 13: An alternative to pinching, which involves adding empty space

together with the rest of an arrow. To address this, the Diagrams library initially provided the `ScaleInv` datatype (Rosenbluth, 2015). This issue is exactly analogous to our need for certain elements to remain unmodified locally and only be drawn at the end, such as the boxes of a morphism or text labels.

Additionally, scaling arrows have a risk of disconnecting them from one another. This is the problem encountered with the gaps between boxes and wires during testing, as discussed in section 4.3. As a potential solution, we could try to use arrows instead of paths when drawing the string diagram wires.

To accommodate some of these delayed attributes, the Diagrams team added the ability for `QDiagram` to have deferred leaves in their internal rendering trees. From its documentation of `DelayedLeaf`:

> “A leaf in a QDiagram tree is either a Prim, or a “delayed” QDiagram which expands to a real QDiagram once it learns the ”final context” in which it will be rendered. For example, in order to decide how to draw an arrow, we must know the precise transformation applied to it (since the arrow head and tail are scale-invariant).” (Manual, 2023)

It could be worthwhile to further explore how arrows are rendered and how to harness these advanced Diagrams features.

5.2.2 Capabilities

Although being able to rewrite string diagrams is vital for any tool operating on them, this project decouples the rendering portion of the algorithm from its rewriting portion. Rewriting diagrams, if ever implemented, would take place before reading the JSON files.

For example, our drawing algorithm could be invoked by other tools with different capabilities and interfaces in their own languages, like DisCoPy from section 4.4, where rendering has been said to be the most challenging aspect of the entire library.

There are also possibilities for interfacing with other technologies, both with other diagrammatic tools —through standardised file formats, such as the JSON files presented here— and with external computational back-ends. For instance, computing the Kronecker product of two large matrices can be very computationally expensive, so that work could be offloaded to a more specialised Python library. Similarly, a diagram representing an electrical circuit could be sent to Simulink for its evaluation. This kind of integration could be done for most of the domain applications mentioned in section 2.2. We have seen that DisCoPy already enables connections with tools for quantum mechanics and machine learning; it would be beneficial to implement these capabilities in this library as well.

Another helpful extension to this project could be to incorporate functors similar to those in DisCoPy. This would enable modifying the base cases of recursion without needing distinct classes or data types for each style. Enhancing ease of use in the user interface through features like this would provide great benefit.

Looking at the big picture, the ultimate long-term goal for this tool would be to have an interactive graphical interface, where users could focus on specific parts of a diagram —corresponding to particular subnodes in the recursion tree— and modify them by dragging, dropping, and clicking buttons. The modifications could come from rewriting the underlying file, while the drawing tool would be responsible for rendering updated diagrams with each change, as well as collecting user interactions and sending them to the rewriting algorithm.

One step toward achieving this vision would be to use some of the more advanced back-ends from the Diagrams library. Currently, the two browser-targeting back-ends are Canvas and HTML5. Running a generated program with these back-ends creates an interactive session at `http://localhost:3000/` (Manual, 2023, 8.6: The Canvas backend).

Integrating our tool with these back-ends opens the door to capturing user interaction through Diagrams' querying system. Every diagram has an associated query which assigns values from a monoid to its points. While the default query simply denotes interior versus exterior points, customised queries could identify more complex subdiagrams:

> "As another interesting example, consider using a set monoid to keep track of names or identifiers for the diagrams at a given point. This could be used, say, to identify which element(s) of a diagram have been selected by the user after receiving the coordinates of a mouse click." (Manual, 2023, 4.5: Using other monoids)

5.3 Conclusions

This dissertation set out to develop a new tool for rendering string diagrams using Haskell's Diagrams library. The primary research question was whether generating diagrams by folding over binary tree structures would be viable. The project has demonstrated that this approach is indeed feasible and has several advantages.

In summary, this project has made the following key contributions:

- Develop a recursive data structure that enables compositional rendering for string diagrams. We store them in binary space partition trees, preserving locality as much as possible, such that subdiagrams render independently before being placed.
- Maintain a right-trapezoidal shape invariant on their brick diagrams, so that they can be drawn directly through Diagrams' juxtaposing operators. This makes our code to closely match the theoretical pseudocode. While some global de-

formations are needed, they still maintain the recursive compartmentalisation. Moreover, this causes the diagram's image to look well-packed and harmonious.

- Design a modular system built on generalisable types and classes. This provides flexibility for users to customise styles and add semantics. An example showcases how to evaluate semantics in the monoidal category of matrices.
- Release the code as open-source with the license BDS3-Clause, so other researchers can use it or build other tools from it.
- Compare our approach to the DisCoPy library, acknowledging the trade-off between our simplicity in drawing, and their effective user interface, plus their availability of domain-specific integrations.

While our implementation successfully demonstrates the tree-based approach, future work can build upon these foundations in several ways:

- Use the `QDiagram` type instead of paths, which would necessitate avoiding non-linear deformations, but would enable operating solely on envelopes rather than using wrapping types.
- Incorporate more advanced Diagrams features like arrows and delayed rendering to improve the algorithm's implementation.
- Expand capabilities beyond drawing, such as adding rewriting rules or an interactive interface.

In conclusion, this dissertation has accomplished its main objectives. The proposed rendering technique was implemented and tested successfully. By storing diagrams recursively, we overcome limitations in existing string diagram renderers. With further development, it has the potential to become a valuable open-source tool for working with string diagrams across many application domains.

References

Abramsky, S. (1996), Retracing some paths in process algebra, *in* U. Montanari and V. Sassone, eds, ‘CONCUR ’96: Concurrency Theory’, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 1–17.
DOI: 10.1007/3-540-61604-7_44.

Abramsky, S. and Coecke, B. (2008), ‘Categorical quantum mechanics’.
DOI: 10.48550/arXiv.0808.1023.

Baez, J. C. and Erbele, J. (2015), ‘Categories in Control’.
DOI: 10.48550/arXiv.1405.6881.

Bolt, J., Coecke, B., Genovese, F., Lewis, M., Marsden, D. and Piedeleu, R. (2017), ‘Interacting Conceptual Spaces I : Grammatical Composition of Concepts’.
DOI: 10.48550/arXiv.1703.08314.

Budde, R., Kautz, K., Kuhlenkamp, K. and Züllighoven, H. (1992), ‘What is prototyping?’, *Information Technology & People* **6**(2/3), 89–95.
DOI: 10.1108/EUM0000000003546.

Coecke, B., Duncan, R., Kissinger, A. and Wang, Q. (2015), ‘Generalised Compositional Theories and Diagrammatic Reasoning’.
DOI: 10.48550/arXiv.1506.03632.

Coecke, B. and Kissinger, A. (2018), Picturing Quantum Processes: A First Course on Quantum Theory and Diagrammatic Reasoning, *in* P. Chapman, G. Stapleton, A. Moktefi, S. Perez-Kriz and F. Bellucci, eds, ‘Diagrammatic Representation and Inference’, Vol. 10871, Springer International Publishing, Cham, pp. 28–31.
DOI: 10.1007/978-3-319-91376-6_6.

Coecke, B., Sadrzadeh, M. and Clark, S. (2010), ‘Mathematical Foundations for a Compositional Distributional Model of Meaning’.
DOI: 10.48550/ARXIV.1003.4394.

Coecke, B. and Spekkens, R. W. (2012), ‘Picturing classical and quantum Bayesian inference’, *Synthese* **186**(3), 651–696.
DOI: 10.1007/s11229-011-9917-5.

De Berg, M., Van Kreveld, M., Overmars, M. and Schwarzkopf, O. (1997), Orthogonal Range Searching: Querying a Database, *in* ‘Computational Geometry’, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 93–117.
DOI: 10.1007/978-3-662-03427-9_5.

De Felice, G., Toumi, A. and Coecke, B. (2021), ‘DisCoPy: Monoidal Categories in Python’, *Electronic Proceedings in Theoretical Computer Science* **333**, 183–197.
DOI: 10.4204/EPTCS.333.13.

Delpeuch, A. (2020), ‘A Complete Language for Faceted Dataflow Programs’, *Electronic Proceedings in Theoretical Computer Science* **323**, 1–14.
DOI: 10.4204/EPTCS.323.1.

Floyd, C. (1984), A Systematic Look at Prototyping, *in* R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Züllighoven, eds, 'Approaches to Prototyping', Springer, Berlin, Heidelberg, pp. 1–18.
DOI: 10.1007/978-3-642-69796-8_1.

Fong, B. (2016), 'The Algebra of Open and Interconnected Systems'.
DOI: 10.48550/arXiv.1609.05382.

Fong, B., Spivak, D. and Tuyéras, R. (2019), Backprop as Functor: A compositional perspective on supervised learning, *in* '2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)', pp. 1–13.
DOI: 10.1109/LICS.2019.8785665.

Fritz, T. (2020), 'A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics', *Advances in Mathematics* **370**, 107239.
DOI: 10.1016/j.aim.2020.107239.

Ghani, N., Hedges, J., Winschel, V. and Zahn, P. (2018), Compositional Game Theory, *in* 'Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science', LICS '18, Association for Computing Machinery, New York, NY, USA, pp. 472–481.
DOI: 10.1145/3209108.3209165.

Halter, M., Patterson, E., Baas, A. and Fairbanks, J. (2020), 'Compositional Scientific Computing with Catlab and SemanticModels'.
DOI: 10.48550/ARXIV.2005.04831.

HaskellWiki (2023), 'Why Haskell just works'.
URL: *https://wiki.haskell.org/Why_Haskell_just_works*

Hedges, J. and Herold, J. (2019), 'Foundations of brick diagrams'.
DOI: 10.48550/arXiv.1908.10660.

Joyal, A. and Street, R. (1991), 'The geometry of tensor calculus, I', *Advances in Mathematics* **88**(1), 55–112.
DOI: 10.1016/0001-8708(91)90003-P.

Kelly, G. M. (1982), *Basic concepts of enriched category theory*, number 64 *in* 'London Mathematical Society lecture note series', Cambridge University Press, Cambridge ; New York.

Kissinger, A. and Van De Wetering, J. (2020), 'PyZX: Large Scale Automated Diagrammatic Reasoning', *Electronic Proceedings in Theoretical Computer Science* **318**, 229–241.
DOI: 10.4204/EPTCS.318.14.

Kissinger, A. and Zamdzhiev, V. (2015), Quantomatic: A Proof Assistant for Diagrammatic Reasoning, *in* A. P. Felty and A. Middeldorp, eds, 'Automated Deduction - CADE-25', Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 326–336.
DOI: 10.1007/978-3-319-21401-6_22.

Manual (2023), 'Haskell's Diagrams library'.
URL: *https://diagrams.github.io/doc/manual.html*

Miranda, E. R., Yeung, R., Pearson, A., Meichanetzidis, K. and Coecke, B. (2022), A Quantum Natural Language Processing Approach to Musical Intelligence, *in* E. R. Miranda, ed., 'Quantum Computer Music: Foundations, Methods and Advanced Concepts', Springer International Publishing, Cham, pp. 313–356.
DOI: 10.1007/978-3-031-13909-3_13.

Patterson, E. (2017), 'Knowledge Representation in Bicategories of Relations'.
DOI: 10.48550/arXiv.1706.00526.

Reutter, D. and Vicary, J. (2019), 'High-level methods for homotopy construction in associative n-categories'.
DOI: 10.48550/ARXIV.1902.03831.

Riley, M. (2018), 'Categories of Optics'.
DOI: 10.48550/arXiv.1809.00738.

Rosenbluth, J. (2015), 'Diagrams: Composition, Envelopes and Alignment'.
URL: *https://github.com/jeffreyrosenbluth/NYC-meetup*

Selinger, P. (2010), A Survey of Graphical Languages for Monoidal Categories, *in* B. Coecke, ed., 'New Structures for Physics', Vol. 813, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 289–355.
DOI: 10.1007/978-3-642-12821-9_4.

Signorelli, C. M., Wang, Q. and Coecke, B. (2021), 'Reasoning about conscious experience with axiomatic and graphical mathematics', *Consciousness and Cognition* **95**, 103168.
DOI: 10.1016/j.concog.2021.103168.

Sivarajah, S., Dilkes, S., Cowtan, A., Simmons, W., Edgington, A. and Duncan, R. (2021), 't|ket⟩: a retargetable compiler for NISQ devices', *Quantum Science and Technology* **6**(1), 014003.
DOI: 10.1088/2058-9565/ab8e92.

Team, T. C. D. (2022), 'The Coq Proof Assistant'.
DOI: 10.5281/ZENODO.7313584.

Toumi, A. (2020), 'DisCoPy: Monoidal Categories in Python'.
URL: *https://youtu.be/kPar2nQVFnY*

Toumi, A., Yeung, R., Poór, B. and de Felice, G. (2023), 'DisCoPy: the Hierarchy of Graphical Languages in Python', *Proceedings ACT 2023* .

Vagner, D., Spivak, D. I. and Lerman, E. (2015), 'Algebras of Open Dynamical Systems on the Operad of Wiring Diagrams'.
DOI: 10.48550/arXiv.1408.1598.

A Code for drawing diagrams

```haskell
instance FoldableDiagram (Path V2 Double) where
    strokeOutput = strokePath
    leaf l = let (al, ar) = leafArity l in
        unitSquare # alignBL # pinch (-al) # pinch ar

    tensor d1 d2 = alignB $ d1 # t1 <> d2 # t2 # snugT
        where (t1, t2) = tensorOps d1 d2
```

Snippet 7: Path V2 Double as a brick diagram (NaiveDiagram)

```haskell
instance (Drawable a) => FoldableDiagram (BrickWrapper a) where
    strokeOutput = strokeDrawing . (^.user)
    leaf l = let (al, ar) = leafArity l in
        BW (unitSquare # alignBL # pinch (-al) # pinch ar) (draw l)
```

Snippet 8: If Drawable a, then BrickWrapper a is FoldableDiagram

```haskell
instance Drawable (Path V2 Double) where
    strokeDrawing = strokePath
    draw (Morphism a _) = drawWires a
    ...

type WiresDiagram = BrickWrapper (Path V2 Double)
```

Snippet 9: Path V2 Double as a string diagram (WiresDiagram)

B Code for compiling diagrams

```
instance (Drawable a, Compilable b)
    => FoldableDiagram (MonCatWrapper a b) where
    strokeOutput d = strokeDrawing (d^.drawing)

    leaf l = MD (unitSquare # alignBL # pinch (-al) # pinch ar)
        (draw l) (baseCase l) where (al, ar) = leafArity l

    tensor d1 d2 =  (d1 # t1 === d2 # t2) # alignB
        # set semantics (tensorOp (d1^.semantics) (d2^.semantics))
        where (t1, t2) = tensorOps d1 d2

    compose d1 d2 =  (d1 # t1 ||| d2 # t2)
        # set semantics (composeOp (d1^.semantics) (d2^.semantics))
        where (t1, t2) = composeOps d1 d2
```

Snippet 10: If Drawable a and Compilable b, MonCatWrapper a b is FoldableDiagram

```
instance Compilable (Matrix Double) where
    baseCase (Morphism _ s) = stringToMatrix s
    baseCase (Crossing p) = permToMatrix p
    baseCase (MorphismWNames _ s) = stringToMatrix s
    baseCase (CrossingWNames _ p) = permToMatrix p

    tensorOp m1 m2 = joinBlocks (m1, z1, z2, m2)
        where z1 = Data.Matrix.zero (nrows m1) (ncols m2)
              z2 = Data.Matrix.zero (nrows m2) (ncols m1)
    composeOp = (*)
```

Snippet 11: Matrix Double is Compilable

```haskell
newtype Labels = Labels [Located (Diagram B)]
type instance N Labels = Double
type instance V Labels = V2
instance Semigroup Labels where
    (<>) (Labels a) (Labels b) = Labels (a <> b)
instance Deformable Labels Labels where
    deform' _ = deform
    deform t (Labels ls) = Labels $ map (deformLoc t) ls
        where deformLoc t' (Loc o s) = Loc (deform t' o) s
instance Drawable Labels where
    strokeDrawing (Labels ls) =
        mconcat . map (\(Loc o s) -> moveOriginTo (-o) s) $ ls
    draw (Morphism (al, ar) s) = let c = 0.5 ^& (0.25*al + 0.25*ar)
        in Labels [ Loc c $ text s # fontSizeG 0.25 ]
    draw l@(MorphismWNames _ s) = draw (Morphism (leafArity l) s)
    draw _ = Labels []
```

Snippet 12: Labels is Drawable

C Code from DisCoPy

```python
def build(d,f):
    if d["type"] == "MorphismWNames": return f(d)
    if d["type"] == "Compose":
        return build(d["diagram1"],f) >> build(d["diagram2"],f)
    if d["type"] == "Tensor":
        return build(d["diagram1"],f) @ build(d["diagram2"],f)

def box(d):
    wires = lambda l: reduce(lambda x,y: x @ y, map(Ty, d[l]))
    return Box(d["label"], wires("dom"), wires("cod"))

data = json.load(open('hh19.json'))
build(data, box).draw(figsize=(3, 3))
```

Snippet 13: Rendering a string diagram from our JSON format with DisCoPy

```python
def mat(d):
    m = eval(d["label"].strip())
    return Matrix([e for r in m for e in r], len(m), len(m[0]))

data = json.load(open('mat.json'))
build(data, mat)
```

Snippet 14: Converting a diagram into its matrix semantics with DisCoPy